Part 1 Sample Solutions

There are many ways to implement Part 1. Here, we explain two secure designs.

Design 1

One approach is to maintain a directory of IDs and keys on the server. The client stores each file under a random ID. A directory listing is kept on the server (encrypted) that maps the filename to the corresponding ID and the keys used to encrypt and MAC that file.

The client needs to create and securely store the keys it will use to maintain the directory on the StorageServer. During client initialization, the client generates two symmetric keys, each 16 bytes long, using `get_random_bytes()`. The client gets its public key from the PublicKeyServer, encrypts the symmetric keys using its public key, and then signs the resulting ciphertext using its private key. It then puts the ciphertext and signature to the StorageServer under the ID `<username>/dir_keys`. In other words, we store $E_{K_A}(k_e, k_a), \text{Sign}_{K_A^{-1}}(E_{K_A}(k_e, k_a))$ on the storage server, where k_e, k_a are symmetric keys for encrypting and MAC'ing the directory listing.

The client can retrieve k_e, k_a by getting this data from the storage server, verifying the signature, and decrypting the ciphertext.

Apart from `<username>/dir_keys`, everything else on the storage server will be encrypted using authenticated encryption, and specifically, Encrypt-then-MAC. In particular, if we want to store data D at ID I, we'll store

$$AE_{k_1,k_2}(D, I) = E_{k_1}(D) || \text{MAC}_{k_2}(I, E_{k_1}(D)).$$

Here E_{k_1} represents AES-CBC encryption with AES key k_1 and with a random 16-byte IV (generated fresh with `get_random_bytes()` each time). The IV is prepended to the ciphertext (it's part of $E_{k_1}(D)$). The IV+ciphertext is MAC'd using SHA256-HMAC under key k_2, and the MAC tag is appended. Including the ID in the input to the MAC ensures that an adversary cannot swap around encrypted values between different IDs.

The directory listing contains a mapping that maps each filename to (r, k_1, k_2), where r is a 16-byte random ID where the file's contents are stored and k_1, k_2 are keys for encrypting the contents. The directory listing is stored at `<username>/directory`, encrypted under keys k_e, k_a (defined above) using authenticated encryption.

A file is stored at `<username>/files/`r (where r is the random ID retrieved from the directory listing), encrypted under keys k_1, k_2 (retrieved from the directory listing) using authenticated encryption. Each time the client uploads a new file, it picks new random 16-byte values r, k_1, k_2, adds an entry to the directory listing, encrypts the file contents, and stores them on the storage server.

We use the utility functions `to_json_string` and `from_json_string` as necessary to convert to and from strings (for encryption, putting to the StorageServer, etc.).

The client catches all exceptions thrown by `from_json_string`, and raises them again as `IntegrityError`. Any error here means that the result we got from the server wasn't valid JSON, which implies that something was modified by the server so we should raise an `IntegrityError` to signal this.

Design 2

The directory approach takes $O(n)$ space but also $O(n)$ time and bandwidth per update (where n is the number of files stored on the server). While you don't need to worry about performance, in practice one might want something with a lower cost per update. Design 2 is an alternative solution that achieves $O(1)$ updates.

Design 2 uses four keys: the public key K_A given to us, a symmetric encryption key k_e, a symmetric MAC key k_a, and a symmetric key k_n for name confidentiality. The symmetric keys k_e, k_a, k_n are stored on the server, encrypted and signed using the public key as in Design 1. All other data is stored encrypted using Encrypt-then-MAC, using k_e and k_a, respectively.

On the server we maintain two types of data: client information nodes, and data nodes.

Client information nodes store k_e, k_a, k_n in encrypted form. `information/<username>` stores $E_{K_A}(k_e, k_a, k_n)$, the encryption of the symmetric keys under our public key, along with a signature computed on this ciphertext. We always check that the signature verifies before trying to decrypt the ciphertext.

A *data node* stores the encrypted contents of a file, namely $E_{k_e}(v)$ where v is the contents of the file, along with a MAC over this ciphertext. In particular, the MAC is computed as $\text{MAC}_{k_a}(C, n)$ where C is the ciphertext $E_{k_e}(v)$, and n is the name of the file. Including the filename in the input to the MAC prevents a malicious storage server from swapping files between IDs (see the Test Case Explanations below for more on this attack).

The tricky part is: where do we store the data node on the storage server? In particular, what ID can we use, without compromising the confidentiality of the filename? In this design, we use the ID `<username>/`r where $r = F_{k_n}(\text{filename})$. The choice of the function F is discussed below, but one reasonable choice is $F_{k_n}(n) = \text{SHA256-HMAC}_{k_n}(n)$.

Encryption. We encrypt all data with AES in CBC mode. The symmetric keys k_e, k_a, k_n are chosen randomly with `get_random_bytes()` during client initialization and stored in the client information node. The IVs are always chosen fresh each time with `get_random_bytes()`, and then prepended to the ciphertext. We use SHA256-HMAC as our MAC. We always verify the MAC before decrypting the messages.

Choice of F. All that remains is to choose the function F. It needs to satisfy three properties: (1) it must be deterministic (the same filename always yields the same ID);[1] (2) it must depend on a key (otherwise dictionary attacks will reveal the filename, for some files); and (3) the value $F_k(n)$ should not reveal anything about the filename n. To achieve property (3), our approach will be to try to essentially mimic Design 1: we'll try to ensure that $F_k(n)$ looks like a random ID, that is random and independent for each different n.

There are several ways to achieve these properties. Here are a few constructions that work:

- $F_k(n) = \text{SHA256-HMAC}_k(n)$. In general, there's no guarantee that a MAC will necessarily provide confidentiality for its input—but some MACs do happen to provide that property. HMAC is one. In particular, HMAC is believed to be pseudorandom function (PRF), which means it behaves like a random function (a function that for each input, outputs a random bit-string that's uniformly distributed and independent of its output for all other inputs). Thus, $F_k(n)$ is like a random string for each filename n, so this effectively has the same properties as Design 1, where we pick a truly random ID for each filename.

- $F_k(n) = H(n||k)$. This also seems to achieve similar properties, if H is a secure cryptographic hash function (e.g., SHA256).

- $F_k(n) = \text{AES-ECB}(H(n))$, where H is a cryptographic hash. Why is this secure? Intuitively, because H is collision-free, each file will have a different value of $H(n)$. And, because AES is a pseudorandom permutation (see the lecture notes), if each input to the AES block cipher is different, then all of the outputs look like different random values, so they reveal nothing about the name n. Or, at a very intuitive level, the problem with ECB arises when you have two plaintext blocks (possibly in two different messages) that are equal; ECB leaks this fact. However, if the plaintext blocks come from a cryptographic hash function, the probability that two plaintext blocks are equal is extremely low.

- $F_k(n) = \text{AES-CBC}(H(n))$, where we use a constant (say, all-zeros) IV with AES-CBC mode, also works, for similar reasons.

These schemes are tricky. How could you have come up with them? The basic approach is to enumerate the properties we need, then try possible combinations of the building blocks we've given you to look for one that has all the necessary properties.

Some choices of F that don't work:

- $F_k(n) = H(n)$ is not secure: because it is an unkeyed function, anyone can compute the function. This allows dictionary attacks: if the attacker has a set of candidates for the filename, the attacker can hash all of those candidate names and see which hash digest matches the ID stored on the file server, thereby learning the filename.

[1] Even though deterministic schemes are vulnerable to chosen plaintext attacks, a deterministic function for hiding the filenames is not a problem for us within the requirements of the project. In particular, the security scenario described on page 6 of the instructions is still satisfied. The file contents, however, must be encrypted using a randomized encryption scheme.

Because human-chosen filenames will often be guessable (low entropy), an attacker can probably learn many filenames in this way.

- $F_k(n) = \text{AES-ECB}_k(n)$. This is not secure: it doesn't protect the confidentiality of the filename n. For instance, storing something under the filename $n_1 =$ `cs161 projects are super easy` and something else under the filename $n_2 =$ `cs161 projects are super hard` would cause the corresponding IDs to match in their first 16 bytes, which reveals that the names n_1, n_2 start with the same 16 bytes.
- $F_k(n) = \text{AES-CBC}_k(n)$, with a constant (e.g., all-zeros) IV. This has a similar problem as AES-ECB mode: if two filenames that start with the same 16 bytes, an adversary will be able to detect this, because the corresponding ciphertexts will start with the same 16 bytes.
- $F_k(n) = \text{RSA-Encrypt}(n)$ doesn't work, because encryption is non-deterministic.
- $F_k(n) = \text{Sign}(n)$ doesn't work, because the existence of the Verify() function allows dictionary attacks. An attacker who suspects the name might be n_0 can check his guess by seeing whether $\text{Verify}_{K_A}(n_0, id)$ is true or not; if Verify returns true, then the attacker knows his guess was correct. Thus, if an attacker can narrow down the set of possible filenames to a few million possibilities, the attacker can identify which one of those candidates is correct.

Sample Security Analysis

Here, we provide examples of attacks that our design protects against. You should use this section as a reference for the security analysis component of your design documents for Parts 2 and 3.

Attack 1. A malicious server might try to change the message after we have stored it, breaking the integrity property. We protect against this attack by MAC'ing or signing all data stored on the server.

Attack 2. An attacker might try to mount a dictionary attack on the name a file is stored under. This is not possible against our scheme (with $F_k(n) = \text{AES-ECB}(H(n))$) because we encrypt the hashed name with our symmetric key, so unless they know the symmetric key they cannot learn the hash.

Attack 3. A malicious server could try to switch the values stored at two names. That is, we may `upload(a,b)` and then `upload(c,d)`. The server might try to switch the values stored on the server around so that `download(a)` would return `d` or `download(c)` would return `b`. We defend against this by including the filename (along with the data) in the input to the MAC, and checking that it matches the expected name before authenticating and decrypting the data during `download`.

Part 1 Tests

Common Mistakes

Here is a selection of common mistakes made in Part 1.

Attempting to use asymmetric private key as symmetric key. The asymmetric private key is an RSA key object, which is a tuple of integers (n, e, d, p, q, u) (the modulus, public exponent, private exponent, factors of n, and the CRT coefficient $(1/p) \bmod q$)—it is only useful for the asymmetric methods. A symmetric key is just a random string of bytes. These should be generated using `get_random_bytes()`.

A related mistake was to generate symmetric keys by converting the private key to a string and hashing it. While this isn't necessarily insecure, it is a bit brittle—your symmetric keys are inherently tied to your private key, making them impossible to change. For this reason, cryptographers would probably consider this approach to be bad practice.

Keeping extra state in the client. Per the project spec, you aren't allowed to keep extra state in the client except as an optimization (such as a locally cached copy of state stored on the server). In the provided functionality tests, `t2` tested whether two instances of the client could download a file uploaded by the other, assuming both instances have the same username and RSA private key. Be careful to make all state recoverable from the server.

For instance, the idea is that Alice should be able to install a copy of the client on her work machine and on her home machine, and as long as she securely transfers her private key to both instances, everything should just work: both instances of the client should have access to all her files.

Length restrictions on asymmetric encryption. Directly encrypting the filename or file contents using asymmetric RSA encryption won't work. Both the filename and file contents can be arbitrarily long. However, RSA-OAEP (and many other asymmetric encryption schemes) are length-limited: they can't handle arbitrarily-long inputs. For the provided `asymmetric_encrypt()` function, the upper limit is 190 bytes.

Using hash(name) for the IDs of files. Because filenames may be predictable (low entropy), using hash(name) can leak the contents of the name, violating the confidentiality requirement. In particular, hash(name) allows dictionary attacks on the filename: an adversary who can enumerate a set of candidate filenames can determine which of those candidates is correct. Two possible fixes are described in the sample designs: IDs should instead of chosen randomly (Design 1), or be the HMAC of the name (Design 2).

Using encryption without authentication. Encrypting file contents, without also providing authentication to detect tampering, is insecure. It allows sophisticated chosen-ciphertext attacks. These attacks can potentially even allow an attacker to learn the contents of the file. If you're interested to learn more, you can read about "padding oracle" attacks.

A related flaw is to use a MAC, but fail to check it, or to decrypt before verifying the MAC. For instance, consider an implementation of download() that first decrypts the encrypted file, then checks the MAC and raises IntegrityError if the MAC tag is invalid. This might sound safe, but actually presents a danger. A malicious storage server could tamper with the ciphertext, then ask the client to download the file. An error will happen at some point, but if the adversary can distinguish whether the error happened during decryption (e.g., CryptoError due to invalid padding) vs. during MAC verification (e.g., IntegrityError thrown by your client), then padding oracle attacks may again become possible. To defend against this: compute a MAC on the ciphertext, and verify the validity of the MAC *before* decrypting the ciphertext.

Using hashes for integrity instead of MACs. Hashes are not a substitute for MACs. A problem with hashes is that they can be recomputed by a malicious storage server. The server can modify the data and then also modify the hash to match, making the modifications go undetected by the client. On a related note, it is difficult to design your own hash-based MAC. There are many attacks on schemes that aren't quite HMAC. You should use a provided MAC function instead of designing your own.

Generating symmetric keys as a hash of the filename. Some generated their symmetric keys for encrypting the file using $H(\text{username}||\text{filename})$. This suffers from the same problems as using $H(\text{filename})$ for the ID. Both usernames and filenames are low entropy and easy to guess, making the resulting symmetric keys easy to guess. Symmetric keys should be randomly generated using `get_random_bytes()`.

Using CTR mode but not setting an IV in the counter. CTR mode is insecure if you ever reuse a nonce, over all encryptions under the same key. The easiest way to set an IV is to generate random bytes and pass them as the prefix to `new_counter`. It is also possible to use the IV to set the initial value of the counter. You can prepend this IV to the ciphertext like you would in CBC mode encryption, and then use it to re-create the counter object during decryption. CTR mode without an IV is insecure; it is vulnerable to the same attacks as if you were to re-use the pad in a one-time pad scheme (the "two-time pad").

Using insecure primitives. DES and RC4 encryption are broken. Encrypted messages stored with these algorithms can be decrypted without the key. MD2, MD5, and SHA-1 are cryptographically broken hash functions and should not be used (and they shouldn't be used for HMAC-based schemes, either: though no attack is known that can forge MAC tags for these schemes, their use is discouraged). Use of these primitives is detected by tests **BrokenTestBadCrypto** and **WeakTestBadCrypto** (see below). Also, ECB mode should be avoided for file encryption—it leaks information about the file contents. Modes like CBC and CTR (which we've covered in class) provide full security if used correctly.

Storing data in plaintext. Storing the file contents or filename on the storage server in plaintext (in unencrypted form) is not secure. If it is sent to the storage server, the storage server can see it. Using the filename as the ID for get()/put() reveals the filename. Similarly, cryptographic keys should not be stored on the storage server unencrypted.

Test Case Explanations

Our autograder included numerous security tests. Here is an explanation of what each one is testing and some sample design flaws that might be responsible for failing that test case.

CheckDifferentialAttacks. Check if there are any values stored on the server that leak any bits of the message. We do this using a technique similar to differential cryptanalysis, which will catch any submissions that store the plaintext in a non-standard manner (e.g., compressed, pickled, ROT13d, etc).

CheckEncryptionMACDifferentKey. MAC keys and encryption keys should be different, and this test case checks just that.

CheckKeyNotStoredPlaintext. The point of creating an encryption key is to hide the content of a message. This test verifies that the encryption keys are never stored, in plaintext, on the server.

CheckPaddingOracle. A padding oracle attack is a chosen-ciphertext attack: if the attacker can convince the client to decrypt (tampered) ciphertexts chosen by the attacker, then a padding oracle attack allows the attacker to decrypt an encrypted message. In this attack, the attacker modifies the ciphertext in some cleverly chosen fashion, asks the client to decrypt it, and then observes the decryption process caused an invalid-padding error. If the attacker can observe whether such an error occurred, then this leaks partial information; after repeating this many times, an attacker can piece together all of these clues to deduce what the original message must have been. These attacks are very powerful and are subtle if you don't know about them.

Failure of this test case typically indicates that you used encryption without authentication, or that you decrypted some ciphertext without checking the MAC or before verifying the validity of the MAC. Defend against padding oracle attacks by always verifying the MAC *before* decrypting message contents.

CheckPlainttextMAC. MACs provide integrity, but they don't promise to provide confidentiality for their input. Therefore, you should generally avoid including plaintext data directly as part of the input to the MAC, if the MAC is not encrypted, as this might leak plaintext bits. Also, when computing a MAC solely on the file contents, the same file contents will always yield the same MAC.

CreateFakeKey. It is important to bind the name of a file to its contents. Otherwise, if a user creates files A and B, a malicious server could swap the names and have downloads for A return the content of B and vice versa.

RandomSwapValues. This test randomly permutes the IDs and values stored on the storage server. For instance, if you have stored {a:b, c:d} on the server, this test might change it to {a:d, c:b}. Then, it will try downloading files. If download() returns incorrect data, then the test case fails. Failing this test case tends to indicate that you are missing a MAC, or you have not bound the MAC'ed value to the filename it is associated with. The fix is generally to include the filename or ID in the input to the MAC, as we did in Design 1 and Design 2.

EncryptionHasNoncesBlackBox. Verifies that encrypted messages are correctly randomized, using IVs or nonces.

CBCIVsNotReused. Verifies that you don't reuse the same IV more than once for CBC mode encryption per key. Such reuse is insecure and can leak partial information about the plaintexts: the same contents will encrypt to the same ciphertext, and if two plaintexts start with the same 16 bytes, then an adversary can detect this (by noticing that the ciphertexts start with the same 16 bytes). Use a fresh, random IV for each CBC encryption you do. Generate the IV using `get_random_bytes()`.

KeysAreChosenFresh. Keys should be chosen randomly using `get_random_bytes()` and should not be generated via any other means. In particular, hashing the private key is poor practice and makes rotating keys difficult if the private key were ever to be compromised.

NameIsNotFromECB. IDs should not be generated by encrypting the filename with ECB mode or CBC/CTR mode with constant IV, as this leaks partial information. In particular, if two filenames start with the same 16 bytes, then the adversary can detect this (by noticing that the ciphertexts start with the same 16 bytes). Also, these modes allow dictionary attacks: an adversary who can convince you to upload files under a name of the attacker's choice will be able to exploit the properties of ECB/CBC/CTR to figure out the filename of your secret file, if the attacker can identify a list of candidate names for the secret file. Finally, CTR mode with a constant IV is especially bad: it is vulnerable to the attacks on pad reuse with the one-time pad.

NameIsNotFromHash. Verifies that IDs are not generated by hashing the filename. That has the same problems outlined above.

SmallCTRUsed. When using CTR mode, the counter should be at least 64 bits, to prevent repetition of the counter value.

CTRUsesUniqueValues. When using CTR mode, the counter values used should be unique. Give either a unique prefix, suffix, or initial value.

StringNotPlaintext. The filename should not directly appear in plaintext on the server. Similarly, the contents of the file should not appear in plaintext on the server.

StringNotPlaintextBoth. You fail this test if both the name and the contents of the file appear in plaintext on the server.

CanEncryptionBeOracle. This test checks that you are not doing anything funky with ciphertexts. We don't take off any points for it, but if you have passed all the functionality tests, it is a warning to double-check your design as the approach seems a bit suspicious. The test checks that it is possible to replace the specifics of `symmetric_encrypt()` with a function that returns random bytes. If your test fails, it means either (a) you fail one of the functionality tests, or (b) when we replace `symmetric_encrypt()` in this way, you fail a functionality test. If you pass all the functionality tests but fail this test, review your design to see what you are doing with the crypto. If you failed some of the functionality tests, you can ignore this test.

BrokenTestBadCrypto and WeakTestBadCrypto. You fail this test if you use insecure cryptographic algorithms. DES and RC4 are broken; you should not use those algorithms. Encrypted messages stored with these algorithms can be decrypted without the key. MD2, MD5, and SHA-1 are cryptographically weak hash functions and should not be used. They shouldn't be used for HMAC-based schemes, either: though no attack is known that can forge MAC tags for these schemes, their use to build a MAC is discouraged. Instead, use strong algorithms like AES, SHA256, and SHA256-HMAC.

NotIntegrityError. Something other than an `IntegrityError` was raised during testing. Our tests change data to unexpected values, and your code must handle those values by the spec.