Project #1 : Encrypted File System

Duc Viet Le
le52@purdue.edu

1 Design Explaination

1.1 Meta-data design

Block 0 contains all meta-data information. The metadata information includes password salt(passwd_salt), encryption salt (enc_salt), mac salt (mac_salt), password digest (passwordDigest), username (Username), encrypted message length. The structure of block 0 is in table 2

Field name	**Size**	**Description**
passwd_salt	16 bytes	Password Salt. It is used to compute the password digest.
enc_salt	16 bytes	Encryption salt. It is used to generate key for encryption
mac_salt	16 bytes	Mac salt. It is used to generate key for encryption
password_digest	32 bytes	Password Digest. $\mathsf{password\_digest} := \mathsf{SHA256}(\mathsf{password}\|\|\mathsf{passwd\_salt})$
user_name	128 bytes	Username
IV	16 bytes	Initialization Vector
file_length	4 bytes	Encrypted file length
Content	764 bytes	Encrypted content of a file system
Tag	32 bytes	Tag of the block from byte 0 till byte 991

Table 1: the structure of Block 0

Similarly, the structure of all other block other than block 0 is as follow:

Field name	**Size**	**Description**
IV	16 bytes	Initialization Vector
Content	976 bytes	Encrypted content of a file system
Tag	32 bytes	Tag of the block from byte 0 till byte 991

Table 2: the structure of Block n where $n > 0$

1.2 User Authentication

When a user tries to read/write to a file, it is required that user to provide the password (i.e password)that is used to encrypted and mac the file. After user provides the password, the password is prepended with the passwd_salt and passed to SHA256 hash function. The system will deny access if

$$(\mathsf{password\_digest} == \mathsf{HA256}(\mathsf{password}||\mathsf{passwd\_salt})) = \mathsf{false}$$

Otherwise, if the password is correct, the encryption key and mac key will be derived as follow:

$$\mathsf{enc\_key} := \mathsf{SHA256}(\mathsf{username}||\mathsf{password}||\mathsf{enc\_salt})[0..127]$$
$$\mathsf{mac\_key} := \mathsf{SHA256}(\mathsf{username}||\mathsf{password}||\mathsf{mac\_salt})[0..127]$$

Both encryption and mac keys are first 16 bytes of hash digest. User will use those keys to decrypt and verify integrity of blocks.

1.3 Encryption Design

We use CTR mode of encryption with Initial Vector. We use the function encript_AES() in ECB mode for one block as our pseudorandom function. Our construction of encryption algorithm is in figure 1

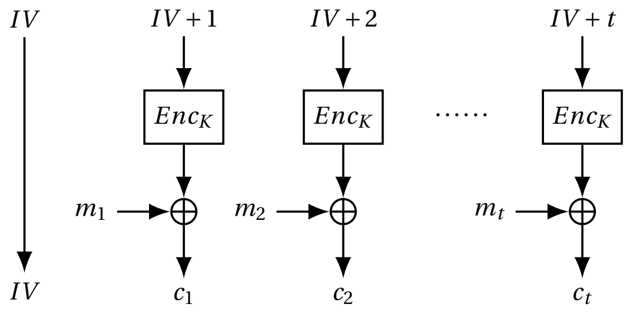

Figure 1: Encryption Algorithm

Similarly, the construction of decryption algorithm is as follow:

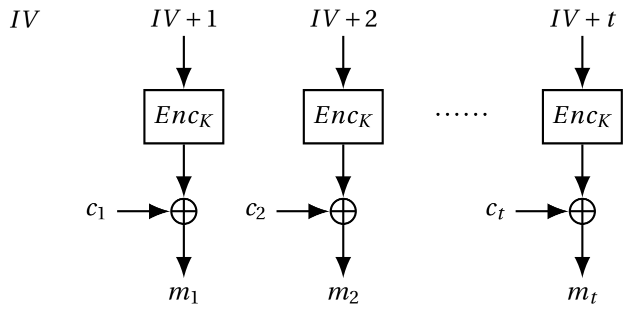

Figure 2: Decryption Algorithm

1.4 Length hiding

In order to hide the actual length of the file system, I pad the actual content of file with 0x00 so that the length of the plaintext passed to the encryption function is 764 byte for first block and 976 byte for all other block. In another word, I followed the suggestion in the hand out.

1.5 Message authentication code

I followed the Encrypt-then-MAC paradigm. I encrypt the content of the file first then compute the tag of the encrypted payload. This construction is proven to be secured against padding oracle attack agaisnt Mac-then-Encrypt construction. The MAC algorithm is constructed using SHA256(·) hash function. Given the mac_key, MAC is a combination of two algorithms described as follows:

- MAC_SHA256(mac_key, m) outputs:

$$\mathsf{tag} := \mathsf{SHA256}((\mathsf{k} \oplus \mathsf{opad}) || \mathsf{SHA256}((\mathsf{k} \oplus \mathsf{ipad}) || \mathsf{m}))$$

  where $\mathsf{ipad} = 0x363636...36, \mathsf{ipad} = 0x5c5c...5c$ and both has the same length with $\mathsf{mac_key}$ which is 16 bytes.

- MAC_SHA256_verify(mac_key, m, tag): return whether:

$$\mathsf{tag} \overset{?}{=} \mathsf{MAC\_SHA256}(\mathsf{mac\_key}, \mathsf{m})$$

1.6 Efficiency

For my design, I use one initialization vector and one tag for each block of size 1024 bytes. This implies that one can store at most 976 bytes of actual data for each block n where $n > 0$, and 764 bytes for block 0. There is a trade off between the number of IV and tag usage and speed efficiency. I chose to design the system this way because I think it is natural to treat each block as one plaintext and is easier to implement. Also, while treating a file system block as several plaintext blocks will improve speed efficiency, we still need to encrypt empty block because we pad plain text block with 0x00 to hide the length. Moreover, for both read and write access, i re-encrypt with new IV; this way introduces lots of overhead. However, I think for an adversary who has a view on the encrypted content will require more effort to differentiate between read and write access, and he only knows if it's a write access if new file is added.

Maximum Storage Efficiency Using less IV and Tags to improve storage efficiency but this design requires more time on updating. Intuitively, we only need one initial vector and one tag stored at block 0; however, if we want to update some blocks, we need to re-encrypt all other block again with new fresh IV. Otherwise, it will be a two-time pad, and the attacker will be able to decrypt our cipher text. Thus, we can store less initialization vector; however, everytime we read or update a portion, we need to re-encrypt and compute tag for large number of block.

Speed Efficiency Using more IVs and Tags to reduce the number of blocks that need to be updated. Similarly, by treating a block as several messages, one can efficiently update part of a block after a read or write. However, this design requires user to store more IV and Tags.

2 Pseudo-code

2.1 Create

takes as input filename, username, password

1. Check whether file exists or not
2. If file exists, verify username and password before removing file.

```
if(dir.exists())
  {
   ... //obtain meta data
   if (!verifyPasswordAndDigest(findUserName(file_name), passwordDigest, passwd_salt, password)) {
       throw new PasswordIncorrectException();
   }
  }

```

3. If file does not exists,
   - Create new file pad it with 0x00
   - Generate metadata

```
byte[] passwd_salt = secureRandomNumber(SALT_LENGTH);
byte[] enc_salt    = secureRandomNumber(SALT_LENGTH);
byte[] mac_salt    = secureRandomNumber(SALT_LENGTH);
file_length = intToByteArray(0);

```

   - Use metadata and password to derive encryption key and mac key

```
byte[] enc_key = DeriveEncryptionKey(password, enc_salt);
byte[] mac_key = DeriveMacKey(password, mac_salt);

```

   - Encrypt the padded content, compute tag, and save content.

```
byte[] cipher = encrypt_AES_CTR(ByteArrayConcatenation(file_length,contentToEncrypt.getBytes()), enc_key); // encrypt file length
payload = ByteArrayConcatenation(payload, cipher); // concatenate metadata with cipher
byte[] tag = MAC_SHA256(payload, mac_key);        // compute tag
save_to_file(payload, meta);                      // save

```

2.2 Length

Takes as input filename and password, outputs file length:

1. Verify whether file exists

```
if (!file.exists()) {
   System.out.println("[−] File doesn't exists");
   throw new Exception();
}
```

2. If file exists, verify login credential

```
if (!verifyPasswordAndDigest(findUserName(file_name), passwordDigest, passwd_salt, password))
{
   throw new PasswordIncorrectException();
}
```

3. Verify integrity of the block about to read

```
if (!integrityOfBlock(file_name,mac_key, 0)) {
   System.out.println("[−] Integrity Error: The metadata block has been modified");
   throw new Exception();
}
```

4. Use metadata and password to derive encryption key and mac key

```
byte[] enc_key = DeriveEncryptionKey(password, enc_salt);
byte[] mac_key = DeriveMacKey(password, mac_salt);
```

5. Decrypt, obtain the file length which is encoded as the first 4 bytes of the first block, re-encrypt with new IV, compute hmac and write back.

```
byte[] plain = decrypt_AES_CTR(cipher, enc_key);
// 4 obtain length
byte[] length = copyRange(plain,0,FILE_LENGTH);
//re−encrypt
byte[] newCipher = encrypt_AES_CTR(plain,enc_key);
// write back
for (int k = ENCRYPTED0_POSITION ; k < MAC_POSITION; k++ ) {
   firstBlock[k] = newCipher[k];
}
// recompute hmac
byte[] payload = copyRange(firstBlock, 0, MAC_POSITION);
byte[] tag = MAC_SHA256(payload, mac_key);
...
//write back
save_to_file(firstBlock, meta);
```

2.3 Read

Takes as input filename, starting_position, length, and password

1. Verify whether file exists

```
if (!file.exists()) {
   System.out.println("[−] File doesn't exists");
   throw new Exception();
}
```

2. If file exists, verify login credential.

```
if (!verifyPasswordAndDigest(findUserName(file_name), passwordDigest, passwd_salt, password))
{
   throw new PasswordIncorrectException();
}
```

3. Use metadata and password to derive encryption key and mac key

```
byte[] enc_key = DeriveEncryptionKey(password, enc_salt);
byte[] mac_key = DeriveMacKey(password, mac_salt);
```

4. Verify integrity of metadata block (block 0)

```
if (!integrityOfBlock(file_name,mac_key, 0)) {
    System.out.println("[−] Integrity Error: The metadata block has been modified");
    throw new Exception();
}
```

5. Determine starting block to read and ending block to read.

```
int start_block = startBlock(starting_position);
int end_block = startBlock(starting_position + len);
if (end_block > startBlock(file_length))
{
    System.out.println("[−] end_block does not exist");
    throw new Exception();
}
```

6. For each block, verify integrity of that block, then decrypt, obtain the data, reencrypt with new IV, recompute mac, and write back.

```
//verify integrity
if (!integrityOfBlock(file_name,mac_key,i))
{
    System.out.println("[−] Integrity Error: block has been tamper");
    throw new Exception();
}
// decrypt
byte[] plain = decrypt_AES_CTR(cipher, enc_key);
// read plain
for (int j = starting_position−(i−1)∗PLAINi_LENGTH−PLAIN0_LENGTH; j < numByteToWrite; j++) {
    content[rp]=plain[j];
    rp++;
    starting_position++;
}
//re−encrypt
byte[] newCipher = encrypt_AES_CTR(plain,enc_key);
...
// recompute hmac
byte[] payload = copyRange(firstBlock, 0, MAC_POSITION);
byte[] tag = MAC_SHA256(payload, mac_key);
...
//write back
save_to_file(firstBlock, meta);
```

2.4 Write

takes as input filename, starting_position, content, and password

1. Verify whether file exists

```
if (!file.exists()) {
    System.out.println("[−] File doesn't exists");
    throw new Exception();
}
```

2. If file exists, verify login credential.

```
if (!verifyPasswordAndDigest(findUserName(file_name), passwordDigest, passwd_salt, password))
{
    throw new PasswordIncorrectException();
}
```

3. Use metadata and password to derive encryption key and mac key

```
byte[] enc_key = DeriveEncryptionKey(password, enc_salt);
byte[] mac_key = DeriveMacKey(password, mac_salt);
```

4. Verify integrity of metadata block (block 0)

```
if (!integrityOfBlock(file_name,mac_key, 0)) {
    System.out.println("[−] Integrity Error: The metadata block has been modified");
    throw new Exception();
}
```

5. Determine starting block and ending block to write.

```
int start_block = startBlock(starting_position);
int end_block = startBlock(starting_position + len);
```

6. For each block, check whether if block exists or not. If block does not exist, create new empty block.

```
if(!meta.exists())
{
    //create a new block of empty string and hmac
    String toWrite = "";
    while(toWrite.length() < PLAINi_LENGTH)
    {
        toWrite += '\0';
    }
    byte[] toWriteCipher = encrypt_AES_CTR(toWrite.getBytes(), enc_key);
    byte[] tag = MAC_SHA256(toWriteCipher,mac_key);
    byte[] toWriteCipherTag = ByteArrayConcatenation(toWriteCipher,tag);
    save_to_file(toWriteCipherTag, meta);
}
```

7. Before any write operation verify integrity of the block, decrypt, update the data if necessary, reencrypt with new IV, recompute mac for each block.

```
//checking integrity
if (!integrityOfBlock(file_name,mac_key,i))
{
    System.out.println("[−] Integrity Error: block has been tamper");
    throw new Exception();
}
// obtain cipher text
byte[] cipher = get_ciphertext(file_name,i);
// decrypt
byte[] plain = decrypt_AES_CTR(cipher, enc_key);
// write to plain
for (int j = starting_position−(i−1)∗PLAINi_LENGTH−PLAIN0_LENGTH; j < numByteToWrite; j++)
{
    plain[j] = content[wp];
    wp++;
    starting_position++;
}
len=len−wp;
//re−encrypt
byte[] newCipher = encrypt_AES_CTR(plain,enc_key);
// write back
for (int k = ENCRYPTEDi_POSITION ; k < MAC_POSITION; k++ ) {
    firstBlock[k] = newCipher[k];
}
// recompute hmac
byte[] payload = copyRange(firstBlock, 0, MAC_POSITION);
byte[] tag = MAC_SHA256(payload, mac_key);
for (int m = MAC_POSITION; m < Config.BLOCK_SIZE; m++)
{
    firstBlock[m] = tag[m−MAC_POSITION];
}
save_to_file(firstBlock, meta);

```

8. finally, if filelength is increased, i will update the meta data by decrypting, writing back, re-encrypting, and recomputing hmac.

```
// update meta data
if (content.length + initalStartingPosition > file_length)
{
    int newLength = content.length + initalStartingPosition;
    System.out.println(newLength);
    byte[] newLengthbyte = intToByteArray(newLength);
    File meta = new File(root, "0");
    byte[] firstBlock = read_from_file(meta);
    // obtain cipher text
    byte[] cipher = get_ciphertext(file_name,0);
    // decrypt
    byte[] plain = decrypt_AES_CTR(cipher, enc_key);
    for (int n = 0; n < FILE_LENGTH; n++) {
        plain[n] = newLengthbyte[n];
    }
    //re−encrypt
    byte[] newCipher = encrypt_AES_CTR(plain,enc_key);
    // write back
    for (int k = ENCRYPTED0_POSITION ; k < MAC_POSITION; k++ ) {
        firstBlock[k] = newCipher[k−ENCRYPTED0_POSITION];
    }
    // recompute hmac
    byte[] payload = copyRange(firstBlock, 0, MAC_POSITION);
    byte[] tag = MAC_SHA256(payload, mac_key);
    for (int m = MAC_POSITION; m < Config.BLOCK_SIZE; m++)
    {
        firstBlock[m] = tag[m−MAC_POSITION];
    }
    save_to_file(firstBlock, meta);
}
```

2.5 Check Integrity

takes as input filename, password

1. Verify whether file exists

```
if (!file.exists()) {
    System.out.println("[−] File doesn't exists");
    throw new Exception();
}
```

2. If file exists, verify login credential.

```
if (!verifyPasswordAndDigest(findUserName(file_name), passwordDigest, passwd_salt, password))
{
    throw new PasswordIncorrectException();
}
```

3. Use metadata and password to derive mac key

```
byte[] mac_key = DeriveMacKey(password, mac_salt);
```

4. Verify integrity of metadata block (block 0)

```
if (!integrityOfBlock(file_name,mac_key, 0)) {
    System.out.println("[−] Integrity Error: The metadata block has been modified");
    throw new Exception();
}
```

5. Determine the number of blocks that need to verified

```
int file_length = length(file_name, password);
int numBlock = startBlock(file_length)+1;
```

6. Verify integrity of each block. Return false if one fail. Otherwise return true

```
for (int i = 1 ; i < numBlock ; i++) {
  if (!integrityOfBlock(file_name,mac_key,i))
  {
     return false;
  }
}
return true;
```

2.6 Cut

1. Verify whether file exists

```
if (!file.exists()) {
   System.out.println("[−] File doesn’t exists");
   throw new Exception();
}
```

2. If file exists, verify login credential.

```
if (!verifyPasswordAndDigest(findUserName(file_name), passwordDigest, passwd_salt, password))
{
   throw new PasswordIncorrectException();
}
```

3. Use metadata and password to derive encryption key and mac key

```
byte[] enc_key = DeriveEncryptionKey(password, enc_salt);
byte[] mac_key = DeriveMacKey(password, mac_salt);
```

4. Verify integrity of metadata block (block 0)

```
if (!integrityOfBlock(file_name,mac_key, 0)) {
   System.out.println("[−] Integrity Error: The metadata block has been modified");
   throw new Exception();
}
```

5. Determine starting block to cut and ending block to cut.

```
int end_block = startBlock(len);
int exist_block = startBlock(file_length);
```

6. Delete redundant blocks.

```
for (int cur = end_block+1; cur <= exist_block; cur++) {
  File file = new File(root, Integer.toString(cur));
  while (file.exists()) {
     file.delete();
     System.out.println("[+] Delete empty file "+ cur);
  }
}
```

7. Remove old content from the current last block by writing empty to the len position.

```
while(flushString.length() < PLAINi_LENGTH)
{
  flushString += ’\0’;
}
write(file_name,len, flushString.getBytes(), password);

```

8. Update metadata which is the file length. This implies reencrypt and recompute mac.

```
//update meta data
byte[] newLengthbyte = intToByteArray(len);
File meta = new File(root, "0");
byte[] firstBlock = read_from_file(meta);
// obtain cipher text
byte[] cipher = get_ciphertext(file_name,0);
// decrypt
byte[] plain = decrypt_AES_CTR(cipher, enc_key);
for (int n = 0; n < FILE_LENGTH; n++) {
   plain[n] = newLengthbyte[n];
}
//re-encrypt
byte[] newCipher = encrypt_AES_CTR(plain,enc_key);
// write back
for (int k = ENCRYPTED0_POSITION ; k < MAC_POSITION; k++ ) {
   firstBlock[k] = newCipher[k-ENCRYPTED0_POSITION];
}
// recompute hmac
byte[] payload = copyRange(firstBlock, 0, MAC_POSITION);
byte[] tag = MAC_SHA256(payload, mac_key);
for (int m = MAC_POSITION; m < Config.BLOCK_SIZE; m++)
{
   firstBlock[m] = tag[m-MAC_POSITION];
}
save_to_file(firstBlock, meta);

```

Note: The reason for re-encrypting and reMAC for reading is to hide access pattern. An adversary should not able to differentiate read and write access. He only knows if it's a write when the write adds more content to file.

3 Design Variations

1. Suppose that the only write operation that could occur is to append at the end of the file. How would you change your design to achieve the best efficiency (storage and speed) without affecting security?
   Answer: I will choose the design that maximum storage efficiency where only one initial vector is used, and I will not pad actual data with 0x00 to hide the length. The reason is that write operation is now append-only operation. We only need to obtain the IV and increment it to encrypt new write content of file. Also, if i pad it with 0x00 to hide the file without renew IV, then this will be a two time pad.

2. Suppose that we are concerned only with adversaries that steal the disks. That is, the adversary can read only one version of the the same file. How would you change your design to achieve the best efficiency?
   Answer:

   - I will choose the the design that uses a single initial vector. The reason is that despite that the vector is chosen uniformly at random, if one reuses one of them some where in the file, then the attacker can decrypt a portion cipher text. The higher the number of IVs are used in the encrypted file, the higher chance of reusing same IV. Therefore, using only one IV reduce such chance because attacker only get to see one version of the file.
   - Also, since the disk gets stolen, one may not need to use MAC at all.

3. Can you use the CBC mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?
   Answer:

   - Yes. CBC mode of encryption can achieve the same level of security as CTR mode. However, one will need a pseudorandom permutation function to construct encryption using CBC mode. In other word, in this project, we will need to use the decript_AES() function and encript_AES() at the same time while we only need to use encript_AES() for CTR mode. Also, since the encryption is linear, one may want to treat each file system block as smaller messages to improve encryption performance.

- From security point of view, I think using CBC mode is better than CTR mode in the case of misusing IV. In CTR mode, reusing IV implies that the attacker will learn lots about the plain text. However, for CBC mode, after encrypting few block, the output will be more diverge, and an adversary may not learn anything other than first few block.

4. Can you use the ECB mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?
   Answer: No. ECB mode of encryption is deterministic and stateless. Therefore, it's not IND-CPA-secure. Unless we generate different keys to encrypt each 128-bit block which is highly inefficient.

4 Paper Reading

Question 1 Consider adversary A that has access to $\mathsf{scrypt}(\cdot)$

1. Creates userid of length 7, userid = abcdefg, log-in and receives $T_7 = \mathsf{scrypt}(\mathsf{abcdefg}||K) = \mathsf{scrypt}(\mathsf{abcdefgk_1})$ where k_1 is the first byte of K.
2. For each possible character r, A tries and verify whether $T_7 \overset{?}{=} \mathsf{scrypt}(\mathsf{abcdefgr})$. If it is equal, A learn that $k_1 = r$ with high probability.
3. Creates userid of length 6, userid = abcdef log-in and receives $T_6 = \mathsf{scrypt}(\mathsf{abcdef}||K) = \mathsf{scrypt}(\mathsf{abcdefgk_1k_2})$ where k_2 is the second byte of K
4. Repeat step 2, with knowledge of k_1 for each possible character r, verify $T_6 \overset{?}{=} \mathsf{scrypt}(\mathsf{abcdefk_1r})$
5. Repeat until he learns $k_3, k_4, ..., k_\ell$

For a verification like in step 2, attacker requires at most 128 call to $\mathsf{scrypt}(\cdot)$. Since the length of K is ℓ, the attacker require at most 128ℓ to learn K