File transfer protocol

Ph4r05 & Miroc

March 20, 2014

1 GetKey protocol

Basic message format and the protocol:

$A \rightarrow S$:	B	getKeyRequest
$A \leftarrow S$:	$RSA^e(A_{pub}, K_1), IV_1, AES^e_{K_1,IV_1}(dh\_group\_id, g^x, \langle nonce_1 \rangle, sig_1)$	getKeyResponse
$A \rightarrow S$:	$B, hash(\langle nonce_1 \rangle)$	getPart2Request
$A \leftarrow S$:	$\langle nonce_2 \rangle, RSA^e(A_{pub}, K_2), IV_2, AES^e_{K_2,IV_2}(sig_2)$	getPart2Response

1.1 Legend

A	party that wants to obtain DH key of the user B.
B	party of which DH key is being obtained.
S	DH key storage server.
$\langle x \rangle$	Means x is Base-64 encoded.
$Apub$	public RSA key of the querying party.
RSA	asymmetric encryption (RSA/ECB/OAEPWithSHA1AndMGF1Padding).
$hash$	sha256.
$sig1$	$sig(hash(version\|\|B\|\|hash(B_{crt})\|\|A\|\|hash(A_{crt})\|\|dh\_group\_id\|\|g^x\|\|\langle nonce_1 \rangle))$.
$sig2$	$sig(hash(version\|\|B\|\|hash(B_{crt})\|\|A\|\|hash(A_{crt})\|\|dh\_group\_id\|\|g^x\|\|\langle nonce_1 \rangle\|\|\langle nonce_2 \rangle))$.

1.2 Implementation details

- $(dh\_group\_id, g^x, \langle nonce_1 \rangle, sig_1)$ is implemented using ProtocolBuffers, message `GetDHKeyResponseBodySCip`.
- $RSA^e(A_{pub}, K_1), IV_1, AES^e_{K_1,IV_1}(X)$ is implemented using ProtocolBuffers, message `HybridEncryption`.
- In order to generate `HibrydEncryption` message for given X use call `HybridCipher.encrypt(sig2, sipCert.getPublicKey(), rand);`
- Signature `sig1` and `sig2` is generated using ProtocolBuffers, message `DhkeySig`.
- Signature is created by class `DHKeySignatureGenerator`.
- Whole logic for generating DHkey as described by this protocol (from the key creator standpoint) is implemented in class `DHKeyHelper`, method `generateDHKey`.
- Note: nonce2 serves mainly as the key/file unique identifier later on.

1.3 Design rationale

Main design goals of this protocol:

- Perfect forward secrecy for the file transfer (DH key exchange, used only once).

- User opens a fixed number of file upload slots for a particular user by generating a fixed number of a keys. Simple Anti-Spam solution. Number of the generated keys can be user-dependent and set adaptively by the end user.
- Freshness using nonce1, nonce2. Nonce1 used to authenticate recipient (able to decrypt 2. protocol message), freshness, used in key generation. Nonce2 for freshness, used in key generation, serves as an unique identifier for key and file being encrypted using this key.
- Server does not have to be trusted. Just simple storage for keys.
- Only destination recipient can see DH public key, verify signatures on the keys, etc... Keys are uploaded to the server using authenticated channel, but here we try to avoid non-repudiation property of digital signatures.
- Anti-DoS features, key needs to be decrypted by a valid recipient in order to be marked unusable on the server. Nobody else can deplete pool of the stored keys for a given user since only recipient knows nonce1 - binding to the particular recipient certificate in time of generating the DH key.
- User has to finish the whole protocol in order to obtain working key (needs nonce2, user confirms he is able to decrypt the key), then is key marked as used on the server and won't be used again.
- Client is responsible for obtaining a valid key. If connection error occur in 1.,2. or 3. protocol message, key on the server are not affected (not marked as used), as if the whole protocol would not happen. If connection error happens in 4. protocol message, user won't receive nonce2 and key cannot be used. Server marks key used anyway and new has to be used (no ACKs for simplicity). But protocol should work over reliable TLS channel, so no such error should happen under normal circumstances.

1.3.1 Potential extensions

- Variable granularity for user key pools. Now we have 1 pool per user. Another approach is to have pool for local users (users sharing same domain - direct TLS auth is possible w.r.t. local user table), and common pool of keys for rest of the users from different domains. In cause of a depletion, group/key pool can be splited (up to the current extreme: 1 pool per user). Problem: authenticity of the key, DoS depletion. Can have multiple RSA blocks (per user, encrypting same symmetric key) and only one symmetric enc block (encrypted key).
- Multiple layers of DH keys - in case of DH key depletion (or new user?).
  - Personal DH key. Generated for 1 month or longer period. Stored on server.
  - Temporary DH key. Generated for 1 week max.
  - Ephemeral DH key. 1 unique key per user per usage (as is now).

  In case of a depletion of ephemeral keys, use only personal & temporary DH keys, somehow bound to the target user (how?). Goal: still preserve perfect forward secrecy. If ephemeral keys exist, user has to use them (how to force it?).

2 File transfer protocol

At this point user has finished GetKey protocol and generates symmetric encryption key from DH pub. Structure of the protocol was inspired by the OTR AKE (sign-mac).

$$
\begin{array}{rlll}
B: & salt_B & = \text{generate random salt} & \\
B: & nonce_B & = \text{generate random nonce} & \\
B: & y & = \text{generate random DH key} & \\
B: & c & = g^{xy} \bmod p & \\
B: & salt_1 & = hash(salt_B \oplus nonce_1) & \\
B: & c_i & = PBKDF2(\langle c \rangle \,||\, \text{"pass-}c_i\text{"}, hash(hash^i(salt_1)||nonce_2), 1024, 256) & \\
B: & M_B & = MAC_{c_1}(version||B||hash(B_{crt})||A||hash(A_{crt})||dh\_group\_id||g^x||g^y||g^{xy}|| & \\
 & & \quad ||\langle nonce_1 \rangle||\langle nonce_2 \rangle||nonce_B) & \\
B: & X_B & = B, nonce_B, sig(M_B) & \\
B: & E_m & = AES^e_{c_5,iv_1}(file\_meta) & \\
B: & E_p & = AES^e_{c_7,iv_2}(file\_pack) & \\
B: & F_m & = iv_1, MAC_{c_6}(iv_1, \langle nonce_2 \rangle, E_m), E_m & \\
B: & F_p & = iv_2, MAC_{c_8}(iv_2, \langle nonce_2 \rangle, E_p), E_p & \\
B: & U_{key} & = salt_B, g^y, AES^e_{c_2}(X_B), MAC_{c_3}\left(AES^e_{c_2}(X_B)\right) & \\
B \rightarrow S: & version, \langle nonce_2 \rangle, A, U_{key}, F_m, hash(F_m), F_p, hash(F_p) & & \text{REST uploadFile}
\end{array}
$$

2.1 Design rationale

Design goals:

- Add potential deniability of the encryption (or avoid non-repundation of the digital signature to some extent). At least in a weak sense (recipient could forge the message somehow). Digital signatures are still needed in order to verify the key exchange.
- Generating multiple symmetric keys, for a) encryption, b) MAC. Using different generating pathways for the passwords in order to avoid undesired relations between each of them.
- Server does not understand the format, simple storage of a binary data.
- Key exchange parameters binding to the identities of the both parties using MAC (binds also nonces to the key exchange), no MiTM.
- Highly randomized nature of the protocol, using saltb for generating the keys (local freshness from the Bs point of view).
- NonceB randomizes signed MAC, (could serve as an argument for potential deniability?)
- Hashes of F_m, F_p are used mainly to verify integrity after file transfer.

2.1.1 Potential extensions

- OTR-AKE inspired: strong-deniability is caused by publishing MAC keys after the session is over (anybody could forge MAC keys then).

2.2 Implementation notes

- Please refer to the reference implementation of the protocol in `DHKeyHelper` class (method, comments).
- SOAP is not suitable for this protocol (large file upload), use REST (over HTTPS) interface on the server using Google Protocol Buffers.
  - Port: 8443
  - Connection type: `TLS`

- Request method: `POST`
- Data format: `multipart/form-data`
- URI: `/rest/upload`

Request parameters:

1. `@RequestParam("version") int version`
2. `@RequestParam("nonce2") String nonce2` (Base64)
3. `@RequestParam("user") String user`
4. `@RequestParam("dhpub") String dhpub` (Base64(U_{key}))
5. `@RequestParam("hashmeta") String hashmeta` (Base64(SHA-256(x)))
6. `@RequestParam("hashpack") String hashpack` (Base64(SHA-256(x)))
7. `@RequestParam("metafile") MultipartFile metafile`
8. `@RequestParam("packfile") MultipartFile packfile`

- Standard form of $nonces_1 nonce_2$ is BASE64, this is used in MACs, as denoted. Exception: $nonce_1$ is used in binary form in generating $salt_1$.
- For generating MAC M_B is used Protocol Buffers message `UploadFileToMac`.
- For generating MAC on files, method `generateFTFileMac` can be used.
- Field U_{key} is represented using Protocol Buffers message `UploadFileKey`. The format is exactly the same, using type `byte` for each attribute.
- Hashing function used to hash files is `SHA-256`. Hash of the file is stored as `byte[]` as returned from `MessageDigest` object in Protocol Buffers message.
- For encryption AES-CBC is used, with IV initialized by `SecureRandom`. AES-GCM would be better, but we don't want authenticated encryption since possible extension counts with publishing MAC keys to get strong deniability, what is not possible since encryption keys have to remain secret.
- AES-CBC for X_B uses `AESCipher` implementation, thus IV is generated in the implementation, format of the message is: 12 B zero bits (salt for unused PBKDF2), 16 B IV, ciphertext.
- Format of a F_m, F_p is exactly the same as in the description of the protocol. Protocol Buffers message used for this element is `UploadFileEncryptionInfo` but actual ciphertext is excluded from the message and is appended in the bytestream due to reason Protocol Buffers does not handle well large structures according to the documentation.

  The order of elements is chosen in this way because: a) length of the ciphertext is not known during streamed encryption, cannot be written prior ciphertext block, thus in decryption nothing can be after ciphertext block, b) MAC can be computed after encryption is done, due to reason a) this cannot be performed in a streamed fashion in one pass.
- For increasing security and improving user experience, file upload separated to 2 files, *meta file* and *archive file*.
  - *Meta file* is supposed to be small, can be downloaded as soon as new file event arrives to the recipient. Should contain basic information about the uploaded files (file names, sizes, thumbnails in case of images). Main idea: do not download big archive files automatically, but still enable to show some details about files to the user so he can decide whether to download the files from the server or not.
  - *Archive file* is a ZIP (GZIP) archive containing files. Can contain multiple files/attachments – possible extension. In a basic use case it contains only a single file. Archive should be flat (contain only files in the root), same as attachments in the email work.

Google Protocol Buffers format of the meta file:

```
// Detailed info about some file from the archive.
message MetaFileDetail {
  optional string fileName=1;
  optional string extension=2;
  optional string mimeType=3;
  optional uint64 fileSize=4;
  optional bytes hash=5;
  optional bytes thumb=6; // JPEG image, thumb representation of the file.
}

message MetaFile {
  optional uint64 timestamp=1;
  optional bytes archiveHash=2; // Hash of the archive counterpart (binds archive to meta).
  optional bytes archiveSig=3;
  optional uint32 numberOfFiles=4;
  optional string title=5;
  optional string description=6;

  repeated MetaFileDetail files=7;  // List of enclosed files in the archive.
}
```

- Use method `initFTHolder` to initialize keys for the protocol. Function generates needed elements up to U_{key} (generates DHkeys, c_i, M_B, X_B). Method returns `FTHolder` that stores protocol variables.
- To generate archive and meta files from `FTHolder`, method `ftSetFilesToSend` can be used. It completizes `FTHolder` state by adding necessary values for file transfer. Files are encrypted and stored (E_m, E_p) on external storage, paths to the files are stored in `FTHolder`.
- Mime file stores thumbnail for a given file in a field `thumb`. It contains `JPEG` thumbnail generated by a sending side if the file to be send is supported format (currently generate thumbnails only for JPG/PNG/GIF/BMP, in future may generate thumb from movies, PDFs, DOCs – but problematic without cloud backend - would increase application size, cannot use cloud due to security problems – documents are highly confidential). Size of the thumbnail image should be below
- Server returns protocol buffer message.

3 File download

In order to download a file from file server 2 requests are needed, one SOAP call to obtain keys and metadata, another one to obtain particular binary file from the server.

3.1 REST

REST interface is used for file download (SOAP is not suitable for large files).

- Method: `GET`
- URI: `/rest/download/{nonce2}/{filetype}`
- Accepts `Range` header so continued download is possible if previous attempt failed somewhere in the middle.
- TODO: returns JSON, refactor to return protocol buffers.

3.2 SOAP

Request to obtain stored files: `ftGetStoredFilesRequest`. Can be used in several ways:

1. set `getAll` to 1, then list of all stored files is returned.
2. set list of users to `users`, list of files uploaded by given users is returned.
3. set list on nonce2s to `nonceList`, list of files with given nonce2 is returned.

Request body:

```
<xs:complexType>
    <xs:sequence>
        <xs:element name="getAll" type="xs:boolean" default="false"/>
        <xs:element name="users" type="hr:sipList" minOccurs ="0" maxOccurs="1"/>
        <xs:element name="nonceList" type="hr:ftNonceList" minOccurs ="0" maxOccurs="1"/>

        <xs:element name="version" type="xs:int" minOccurs ="0" maxOccurs="1"/>
        <xs:element name="auxVersion" type="xs:int" minOccurs ="0" maxOccurs="1"/>
        <xs:element name="auxJSON" type="xs:string" minOccurs ="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>
```

Response body:

```
<xs:complexType>
    <xs:sequence>
        <xs:element name="errCode" type="xs:int"/>
        <xs:element name="storedFile" type="hr:ftStoredFileList" minOccurs ="0" maxOccurs="1" />
    </xs:sequence>
</xs:complexType>
```

Stored file:

```
<xs:sequence>
    <xs:element name="sender" type="hr:userSIP"/>
    <xs:element name="sentDate" type="xs:dateTime"/>
    <xs:element name="nonce2" type="hr:ftNonce"/>
    <xs:element name="hashMeta" type="xs:string"/>
    <xs:element name="hashPack" type="xs:string"/>
    <xs:element name="sizeMeta" type="xs:long" />
    <xs:element name="sizePack" type="xs:long" />
    <xs:element name="key" type="hr:binaryPayload"/>
    <xs:element name="protocolVersion" type="xs:int" minOccurs ="0" maxOccurs="1"/>
</xs:sequence>
```

- Field `key` is serialized U_{key}.
- Method `FTHolder processFileTransfer(DHOffline data, byte[] ukey)` can be used to recontruct `FTHolder` from the key response. It is needed to load corresponding DHKey from the database, using field `nonce2`.
- File is downloaded using HTTPS GET with method `downloadFile(FTHolder holder, KeyStore ks, String password, int fileIdx, boolean allowReDownload)`. IV and MAC is parsed, stored to the holder, ciphertext is stored to a new file, its filename is in holder. This method supports resume of a download.
- Then file has to be decrypted by method `boolean decryptFile(FTHolder holder, int fileIdx)`.
- And in case of meta file, parsed `MetaFile reconstructMetaFile(FTHolder holder)`.