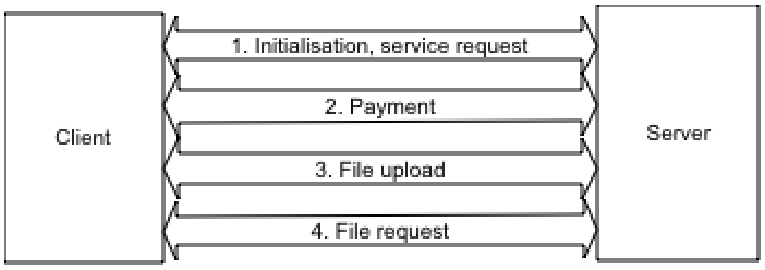
Cryptographic protocols.
”Secure storage” service protocol

Ilja Kuzovkin

January 24, 2011

The goal is to design the protocol for secure file hosting. Functionality should consist of: initialization of communication, payment for the service, uploading file and storing it on he server, generating the token by the server side and passing it to the client, returning corresponding file when it is queried by client.



We will go through those steps one-by-one as separate subroutines.
In the formal descriptions of the protocols we skip steps which are being done inside one machine, only the steps which include communication over the network are included into π-calculus and *common syntax* descriptions.

1 Initialization

1.1 Description

Here client should ask server for service, server will respond and some key exchange for further communication should be made. Shared key is needed for both parties to be sure further communication is going between them two.

1. Client sends his public key PK_C to the Server
2. Server signs it's public key PK_S and sends it to the client
3. Client verifies (using CA) that PK_S is indeed Server's public key
4. Client and Server exchange fresh symmetric key $K_{S,C}$ using Diffie–Hellman key exchange. During key establishment Client encrypts every message using PK_S and Server encrypts every message using PK_C
5. Client stores $K_{S,C}$
6. Server stores $K_{S,C}$ and associates it with Client's identity C

1.2 Representation in common syntax

```
C, S         : client, server
pkC, pkS     : asymmetric keys
kCS          : symmetric key
g, a, b, p   : number

1. C -> S: pkC
2. S -> C: pkS, {g, p}pkC (* CA part will go here *)
3. C -> S: {g^a mod p}pkS
4. S -> C: {g^b mod p}pkC (* at this point C and S can compute kCS *)
```

2 Payment

2.1 Description

Here Client is going to pass fee M_{coin} to the Server and Server has to reply as he gets it.

1. Client generates nonce N_C
2. Client sends $c_1 = E_{K_{S,C}}(N_C, C, M_{coin})$, and identity C to the Server
3. Server decrypts the message c_1 using $K_{S,C}$
4. Server checks if C from the c_1 is the same as C Client sent by plain text
5. Server checks if M_{coin} is correct fee (I mean correct amount of money)
6. If 4 and 5 are correct Server sends $c_2 = E_{K_{S,C}}(N_C)$ and is ready to accept the file.
7. Client decrypts c_2 and checks if N_C is the same. If it is, then Client can be sure that Server got the fee and is ready to accept the file.

2.2 Representation in common syntax

```
C, S    : client, server
kCS    : symmetric key
nC     : nonce
M_coin : number

1. C -> S: {nC, C, M_coin}kCS, C
2. S -> C: {nC}kCS
```

3 File upload

3.1 Description

Here Client will send file to the Secure Storage.

1. Client encrypts his file $c_3 = E_{PK_C}(file)$
2. Client send to the server $c_4 = E_{K_{S,C}}(c_3, C, N_C^2)$ and his identity C
3. Server decrypts c_4 using $K_{S,C}$
4. Server checks if C sent by plain text and C from c_4 are the same

5. Server computes hash of encryption of a file $h = hash(c_3)$. Hash function has to be one-way and 2nd preimage resistant (see 4.1).
6. Server stores h and $c3$ and now it has quadruple $(C, PK_C, h, c3)$ (which are: client identity, client public key, hash, encrypted file)
7. Server sends to Client client $c_5 = E_{K_{S,C}}(N_C^2, h)$
8. Client decodes c_5 and checks if N_C^2 is the same
9. If N_C^2 is correct Client now can use h as token for file retrieval

3.2 Representation in common syntax

```
C, S : client, server
file : file
pkC  : asymmetric keys
kCS  : symmetric keys
nC   : nonce
h    : hash

1. C -> S: {{file}pkC, C, nC}kCS, C
2. S -> C: {nC, h}kCS
```

4 File request and retrieval

4.1 Description

File request can be made by anyone, not only the user who uploaded the file. So basically token is public. It is secure if one-wayness of a hash function. Server will return encrypted file and if user knows the key, he will be able to decrypt.

1. User sends token h to the server
2. Server returns corresponding $c3$ (which is the encrypted file)

4.2 Representation in common syntax

```
C, S       : client, server
h          : hash
enc_file   : file encrypted with Client's public key

1. C -> S: h
2. S -> C: enc_file
```

We do not need to create a model for the ProVerif, because during this dialog we have nothing to hide. h is public hash, which everyone can use to retrieve the file and enc_file is encrypted file, so no one except Client can decrypt it.

5 Non-repudiation

5.1 Description

If the file returned by Server is not the Client expected to get, he can accuse Server in being dishonest. Do to so Client broadcasts h and c_3. Now everyone can compute h from c_3 using hash function, request same file from the Server and compare resulting files. Here 2nd preimage resistance requirement (which was mentioned in 3.1.5) arises.

6 ProVerif

6.1 Representation in π-calculus

You can also find this code in the file pi-secure-storate-v1

```
free c.
private free s.
private free Mcoin.
private free file.


(* Public key cryptography *)

fun pk/1.
fun encrypt/2.
reduc decrypt(encrypt(x,pk(y)),y) = x.

(* Signatures *)

fun sign/2.
reduc getmess(sign(m,k)) = m.
reduc checksign(sign(m,k), pk(k)) = m.

(* Shared key cryptography *)

fun enc/2.
reduc dec(enc(x,y),y) = x.

(* Diffie-Hellman functions *)

fun exp/2.
fun gr/1.
equation exp(x,gr(y)) = exp(y,gr(x)).

(* Test whether secrets are secret *)

query attacker:s.
query attacker:Mcoin.
query attacker:file.

query evinj:endFparam(x) ==> evinj:beginFparam(x).
query evinj:endSymKey(x) ==> evinj:beginSymKey(x).
query evinj:endMcoin(x) ==> evinj:beginMcoin(x).


(* The process *)

let processC =
(* 2 in *)
in(c, m1);
let g = decrypt(m1, skS) in
```

```
(* 3 out *)
new a;
let v1 = encrypt(exp(a, g), pkS) in
out(c, v1);

(* 4 in *)
in(c, m2);
let gb = decrypt(m2, skC) in
event endSymKey(exp(a, gb));

let k = exp(a, gb) in

(* send secret over channel *)
out(c, enc(s, k));

(* payment *)
new nA;
new C;
event beginMcoin(Mcoin);
out(c, (enc((nA, C, Mcoin), kCS), C));

(* upload *)
new nC;
event beginFparam(encrypt(file, pkC));
out(c, (enc((encrypt(file, pkC), C, nC), kCS), C)).


let processS =
(* 2 out *)
new g;
let v1 = encrypt(g, pkC) in
out(c, v1);

(* 3 in *)
in(c, m1);
let ga = decrypt(m1, skS) in

(* 4 out *)
new b;
event beginSymKey(exp(b, ga));
let v2 = encrypt(exp(b, g), pkC) in
out(c, v2);

let k = exp(b, ga) in

(* receive secret over channel *)
in(c, m2);
let s2 = dec(m2, k) in

(* payment *)
in(c, (m3, C1));
let (nA, C2, Mcoin) = dec(m3, kCS) in
```

```
if C1 = C2 then
event endMcoin(Mcoin);
out(c, enc(nA, kCS));

(* upload *)
in(c, (m4, C3));
let (enc_file, C4, nC) = dec(m4, kCS) in
if C3 = C4 then
event endFparam(enc_file);
new h;
out(c, enc((nC, h),kCS)).


process
(* 1 *)
new skC;
let pkC = pk(skC) in out(c, pkC);
new skS;
let pkS = pk(skS) in out(c, pkS);
new kCS;
((!processC) | (!processS))
```

6.2 ProVerif Results

6.2.1 v1

Based on the file *pi-secure-storage-v1*.
Secret values remain secret

```
RESULT not attacker:s[] is true.
RESULT not attacker:Mcoin[] is true.
RESULT not attacker:file[] is true.
```

2 out of 3 correspondence properties also hold

```
RESULT evinj:endMcoin(x_67) ==> evinj:beginMcoin(x_67) is true.
RESULT evinj:endFparam(x_1991) ==> evinj:beginFparam(x_1991) is true.
```

But ProVerif found an possibility to break correspondence property in the process of symmetric key exchange.

```
1. We assume as hypothesis that
attacker:encrypt(x_1819,pk(skS_16[])).

2. The message pk(skS_16[]) may be sent to the attacker at output {4}.
attacker:pk(skS_16[]).

3. We assume as hypothesis that
attacker:y_1816.

4. By 3, the attacker may know y_1816.
Using the function gr the attacker may obtain gr(y_1816).
attacker:gr(y_1816).
```

```
5. By 4, the attacker may know gr(y_1816).
By 2, the attacker may know pk(skS_16[]).
Using the function encrypt the attacker may obtain encrypt(gr(y_1816),pk(skS_16[])).
attacker:encrypt(gr(y_1816),pk(skS_16[])).

6. The message encrypt(gr(y_1816),pk(skS_16[])) that the attacker may have by 5
may be received at input {8}. The event beginSymKey(exp(y_1816,gr(b_1824)))
(with environment m1_20 = encrypt(gr(y_1816),pk(skS_16[])), @sid_170 = @sid_1817,
@occ10_907 = @occ_cst()) may be executed at {10}. So the message
encrypt(exp(b_1824,g[]),pk(skC_14[])) may be sent to the attacker at output {12}.
attacker:encrypt(exp(b_1824,g[]),pk(skC_14[])).

7. The message encrypt(x_1819,pk(skS_16[])) that the attacker may have by 1 may
be received at input {27}. The message encrypt(exp(b_1824,g[]),pk(skC_14[]))
that the attacker may have by 6 may be received at input {31}. So event
endSymKey(exp(a_1823,exp(b_1824,g[]))) may be executed at {33} in session endsid_1821.
end:endsid_1821,endSymKey(exp(a_1823,exp(b_1824,g[]))).


A more detailed output of the traces is available with
  param traceDisplay = long.

out(c, pk(skC_14_8)) at {2}

out(c, pk(skS_16_5)) at {4}

out(c, encrypt(g,pk(skC_14_8))) at {7} in copy a_4

in(c, encrypt(a_2,pk(skS_16_5))) at {27} in copy a_1

out(c, encrypt(exp(a_40_6,a_2),pk(skS_16_5))) at {30} in copy a_1

in(c, encrypt(gr(a_3),pk(skS_16_5))) at {8} in copy a_4

event(beginSymKey(exp(b_22_7,gr(a_3)))) at {10} in copy a_4

out(c, encrypt(exp(b_22_7,g),pk(skC_14_8))) at {12} in copy a_4

in(c, encrypt(exp(b_22_7,g),pk(skC_14_8))) at {31} in copy a_1

event(endSymKey(exp(a_40_6,exp(b_22_7,g)))) at {33} in copy a_1

The event endSymKey(exp(a_40_6,exp(b_22_7,g))) is executed in session a_1.
A trace has been found, assuming the following hypothesis :
    * attacker:encrypt(a_2[],pk(skS_16_5[]))

RESULT evinj:endSymKey(x_784) ==> evinj:beginSymKey(x_784) cannot be proved.
```

To fix that I added a nonce, which is being sent from the Client at the first step. This nonce is encrypted by Server's pkS. Server holds nonce in memory until key exchange is done and after that encrypts this nonce by a newly exchanged key kCS and sends it over to the Client. Client decrypts it with the same symmetric key kCS and checks if this nonce is the same as he sent in the beginning of the dialog.

6.2.2 v2

Based on file *pi-secure-storage-v2*.
As we can see from the output below our fix is working.

```
RESULT evinj:endMcoin(x_70) ==> evinj:beginMcoin(x_70) is true.
RESULT evinj:endSymKey(x_764) ==> evinj:beginSymKey(x_764) is true.
RESULT evinj:endFparam(x_1770) ==> evinj:beginFparam(x_1770) is true.
RESULT not attacker:file[] is true.
RESULT not attacker:Mcoin[] is true.
RESULT not attacker:s[] is true.
```

6.2.3 v3

Based on file *pi-secure-storage-v3*.
Because processes were not working as expected, I replaced all *reduc* constructions with *fun* and *equation* pairs. This helped and now both *endOfS* and *endOfC* are reachable. As we can see from the run results below none of out correspondence properties holds:

```
268: RESULT not attacker:endOfS[] is false.
322: RESULT not attacker:endOfC[] is false.
440: RESULT evinj:endMcoin(x_9854) ==> evinj:beginMcoin(x_9854) is false.
526: RESULT evinj:endSymKey(x_14920) ==> evinj:beginSymKey(x_14920) is false.
683: RESULT evinj:endFparam(x_26294) ==> evinj:beginFparam(x_26294) is false.
690: RESULT not attacker:file[] is true.
696: RESULT not attacker:Mcoin[] is true.
702: RESULT not attacker:s[] is true.
```