Cryptographic protocols: formal and computational proofs

(All documents are allowed; duration: 3h)

February 26, 2020

For the ease of the correctors, please use different sheets for the two parts of the exam.

2 Analyzing Protocols and Implementations (1h30, 1/2 the mark)

A startup called *SuperSecure* proposes a new cryptographic protocol that smartphone apps can use to upload and download sensitive data from cloud-based servers. You have been tasked with analyzing whether this protocol is secure and correctly implementing the protocol in F*.

Protocol. The protocol is between an initiator I (a smartphone app) and a responder R (a server). I wishes to send a secret, mutually-authenticated request message m to R and R responds with a secret response message m' meant only for I. We assume that everyone knows the public keys of I and R (pk_I,pk_R) but only I and R know their own private keys (sk_I,sk_R).

The protocol uses a variation of the encrypt-then-sign pattern for requests: the initiator I generates a fresh encryption key ek, encrypts it under the public key of R (pk_R), encrypts the secret message m using ek to obtain a ciphertext c, and then signs c using the initiator's private key (sk_A). The response message m' is simply encrypted using ek. The request and response messages are tagged with constants 0 and 1 to distinguish them.

$$\begin{array}{l} A \longrightarrow B : A, \mathsf{penc}(ek, pk_B), \mathsf{enc}(0 \parallel m, ek), \mathsf{sign}(\mathsf{enc}(0 \parallel m, ek), sk_A) \\ B \longrightarrow A : \mathsf{enc}(1 \parallel m', ek) \end{array}$$

Hence, the protocol uses three cryptographic primitives: public-key encryption (penc, pdec), public-key signatures (sign, verify), and authenticated symmetric encryption (enc, dec). The types and functional lemmas (in F* syntax) for these primitives are given in Appendix A.

Security Goals. The two main security goals of the protocol can be stated as follows:

- **Confidentiality:** Any request message m or response message m' sent between I and R should only be known to I and R.
- **Authentication:** If R accepts a request m from I, then:
  - m must be known to I (*Sender Knowledge*),
  - I must have sent m (*Sender Authentication*),
  - R must be the intended recipient of m (*Receiver Authentication*).

  Similarly, if I accepts a response m' from R, then the three dual properties must hold, in addition to a new goal that says that m' must be correlated with m.
  - m' must be known to R (*Sender Knowledge*),
  - R must have sent m' (*Sender Authentication*),
  - I must be the intended recipient of m' (*Receiver Authentication*),
  - m' must have been sent in response to m (*Response Correlation*).

Threat Model. A typical deployment scenario for the protocol is where a smartphone Alice (A) plays the role of I and a server Bob (B) plays the role of the server, and we wish to protect messages sent between them. In addition, we assume that Alice is also willing to use this protocol to send requests to Mallory (M), and Bob is willing to respond to requests from Mallory (M). Mallory is controlled by the adversary (i.e. the adversary knows sk_M); she is free to play the roles of both I and R and may deviate from the protocol.

In addition to controlling Mallory, we assume (as usual) that the adversary controls the network, and hence can read, inject, and redirect messages sent on the public network. The adversary can also create any number of keys, and may use the full cryptographic API of Appendix A to construct and deconstruct messages. However, we assume that the attacker cannot bypass the cryptographic API to break the underlying crypto primitives, and he cannot simply guess the secret keys of A and B.

In this adversarial setting, we expect the security goals stated above should hold for all messages m, m' sent between A and B, as long as sk_A and sk_B remain secret.

2.1 Exercise 1

Does the protocol satisfy its two security goals (including all the authentication sub-goals)? List the security goals (Request Confidentiality, Response Confidentiality,. . .) and for each goal, write Yes/No answers, and informally explain why you think the goal is achieved or not.
If not, demonstrate an attack, and then fix the protocol so that it achieves its goals.

2.2 Exercise 2

Write an F* implementation for the protocol above.
The model will consist of:

- a function send_req for A that takes m, pk_B, and sk_A as input and returns the freshly generated encryption key ek and the (encrypted) message c to be sent to B;
- a function recv_req for B that takes c, pk_A, and sk_B as input and returns the key ek and request message m;
- a function send_resp for B that takes m, m', and ek as input and returns the (encrypted) message c' that will be sent to A;
- a function recv_resp for A that takes c' and ek as input and returns m'.

You can assume the F* declarations in Appendix A for the cryptographic primitives. If you need additional functions, declare them. The syntax you use for F* need not be perfect, but the logic of the code should be precise and clear.

2.3 Exercise 3

Appendix B presents a labeled cryptographic API. Write labeled types for the four functions above. State why your types guarantee the secrecy of m and m'.

2.4 Exercise 4

Rewrite the protocol code to use the labeled API. By appealing to the types in the labeled API, and by adding brief comments to each line of your code, argue why your code is well-typed.

2.5 Extra

Show how would you encode the authenticity goals by modifying the types of your four functions. Then, informally describe the steps that will be needed to prove that your code meets these authentication goals. (Hint: You will need to extend the labeled API of Appendix B with signature predicates and additional lemmas.)

A Unlabeled Cryptographic API

```
(* A type for byte arrays *)
val bytes: Type0
val zero: bytes (* The constant 0 *)
val one: bytes (* The constant 1 *)

val concat: bytes → bytes → bytes
val split: bytes → option (bytes * bytes)
val concat_split_lemma: b1:bytes → b2:bytes →
  Lemma (split (concat b1 b2) == Some (b1,b2))

(* Authenticated Symmetric Encryption *)
val sym_key: Type0
val sym_keygen: unit → ST sym_key
                      (requires (λ h0 → ⊤))
                      (ensures (λ h0 _h1 → h0 == h1))
val sym_enc: k:sym_key → p:bytes → c:bytes
val sym_dec: k:sym_key → c:bytes → option bytes
val sym_enc_dec_lemma: k:sym_key → p:bytes →
  Lemma (sym_dec k (sym_enc k p) == Some p)

(* Public Key Encryption − Asymmetric *)
val pub_key: Type0
val priv_key: Type0
val priv_keygen: unit → ST priv_key
                      (requires (λ h0 → ⊤))
                      (ensures (λ h0 _h1 → h0 == h1))
val priv_to_pub: priv_key → pub_key
val pke_enc: pk:pub_key → k:ae_key → c:bytes
val pke_dec: sk:priv_key → c:bytes → option ae_key
val pke_enc_dec_lemma: sk:priv_key → k:ae_key →
  Lemma (pke_dec sk (pke_enc (priv_to_pub sk) k) == Some k)

(* Signatures − Asymmetric *)
val sig_key: Type0
val verif_key: Type0
val sig_keygen: unit → ST sig_key
                      (requires (λ h0 → ⊤))
                      (ensures (λ h0 _h1 → h0 == h1))
val sig_to_verif: sig_key → verif_key
val sign: sk:sig_key → msg:bytes → sg:bytes
val verify: vk:verif_key → msg:bytes → sg:bytes → bool
val sign_verify_lemma: sk:sig_key → msg:bytes →
  Lemma (verify (sig_to_verif sk) msg (sign sk msg) == true)
```

B Labeled Cryptographic API

```
(* Principals: participants in protocols *)
let principal = string (* "A", "B", "M", etc. *)

(* Secrecy Labels: sets of principals *)
type label =
  | Public: label
  | Secret: list principal → label
let includes l1 l2 =
  match l1,l2 with
  | Public,_ → ⊤
  | Secret pl1, Secret pl2 → ∀p. List.Tot.mem p pl2 ⟹ List.Tot.mem p pl1
  | _,_ → ⊥

(* A type for *labeled* byte arrays *)
```

```
val lbytes: label → Type0
val coerce: l1:label → l2:label{includes l1 l2} → b1:lbytes l1 → b2:lbytes l2

(* zero and one are public *)
val zero: lbytes Public
val one: lbytes Public

(* Concatenation preserves labels *)
val concat: l:label → lbytes l → lbytes l → lbytes l
val split: l:label → lbytes l → option (lbytes l * lbytes l)
val concat_split_lemma: l:label → b1:lbytes l → b2:lbytes l →
  Lemma (split l (concat l b1 b2) == Some (b1,b2))

(* Authenticated Symmetric Encryption *)
val sym_key: l:label → Type0
val coerce_sym_key: l1:label → l2:label{includes l1 l2} → sym_key l1 → sym_key l2
val sym_keygen: l:label → ST (sym_key l)
                         (requires (λ h0 → ⊤))
                         (ensures (λ h0 _h1 → h0 == h1))
val sym_enc: kl:label → ml:label{includes ml kl} → k:sym_key kl → m:lbytes ml → c:lbytes Public
val sym_dec: kl:label → k:sym_key kl → c:lbytes Public → option (lbytes kl)

val sym_enc_dec_lemma: kl:label → ml:label → k:sym_key kl → m:lbytes ml →
  Lemma (requires (includes ml kl))
        (ensures (sym_dec kl k (sym_enc kl ml k m) == Some (coerce ml kl m)))

(* Public Key Encryption − Asymmetric *)
val pub_key: a:principal → Type0
val priv_key: a:principal → Type0
val priv_keygen: a:principal → ST (priv_key a)
                         (requires (λ h0 → ⊤))
                         (ensures (λ h0 _h1 → h0 == h1))
val priv_to_pub: a:principal → priv_key a → pub_key a
val pke_enc: r:principal → kl:label{includes kl (Secret [r])} →
             pk:pub_key r → k:ae_key kl → c:lbytes Public
val pke_dec: r:principal → sk:priv_key r → c:lbytes Public → option (ae_key (Secret [r]))
val pke_enc_dec_lemma: kl:label → r:principal → sk:priv_key r → k:ae_key kl →
  Lemma (requires (includes kl (Secret [r])))
        (ensures (pke_dec r sk (pke_enc r kl (priv_to_pub r sk) k) == Some (coerce_ae_key kl (Secret [r]) k)))

(* Signatures − Asymmetric *)
val sig_key: a:principal → Type0
val verif_key: a:principal → Type0
val sig_keygen: a:principal → ST (sig_key a)
                         (requires (λ h0 → ⊤))
                         (ensures (λ h0 _h1 → h0 == h1))
val sig_to_verif: a:principal → sig_key a → verif_key a
val sign: a:principal → ml:label → sk:sig_key a → msg:lbytes ml → sg:lbytes ml
val verify: a:principal → ml:label → vk:verif_key a → msg:lbytes ml → sg:lbytes ml → bool
val sign_verify_lemma: a:principal → ml:label → sk:sig_key a → msg:lbytes ml →
  Lemma (verify a ml (sig_to_verif a sk) msg (sign a ml sk msg) == true)
```