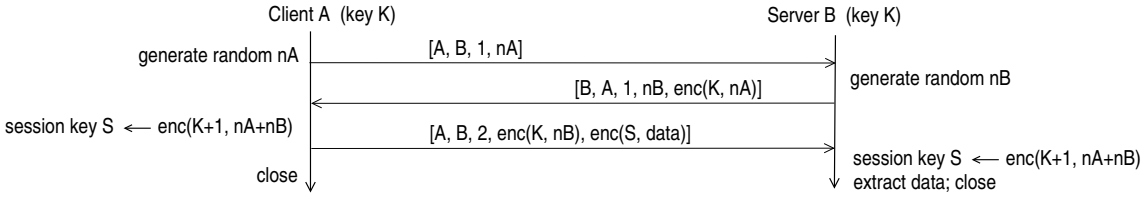
Modeling and Analyzing Authentication Protocols

A. Udaya Shankar
shankar@cs.umd.edu

~~March 1, 2011~~ March 7, 2011

1 Modeling protocols by distributed programs

Authentication protocols are typically described with message handshake diagrams, as illustrated below. But such descriptions can be rather incomplete. For example, the handshake below does not say whether client A talks to server B just once or repeatedly; it does not say what the attacker is allowed to do.



To avoid such ambiguities, we will specify authentication protocols by **distributed programs**, also known as a multi-threaded programs. The above protocol could be specified as follows.

```
Protocol(A, B) {
   chan ← [];                    // channel
   hst ← [];                     // connnection history
   K ← random();                 // shared key
   startSystem(B,Server(B,A,K)); // start server B
   startSystem(A,Client(A,B,K)); // start client A
   startSystem(Attacker());    // start Attacker
}
```

```
Attacker() {
  α; // everything attacker has read
  <read messages in chan>
  <send messages [A,B,...]>
  <receive messages [B,A,...]>
}
```

```
Client(A, B, K) {
   t ← startThread(client());
   return;

   function client() {
      while (true) {
         nA ← random();
         tx([A,B,1,nA]);
    1:   msg ← rx([B,A,1,.,.]);
         if (msg[4] = enc(K,nA)) {
            S ← enc(K+1,nA+nB);
            hst.append([A,S]);
            tx([A,B,2,enc(K,msg[3]),enc(S,data)]);
         }
      }
} }
```

```
Server(B, A, K) {
   t ← startThread(server());
   return;

   function server() {
      while (true) {
    1:   msg ← rx([A,B,1,.]);
         nB ← random();
         nA ← msg[3];
         tx([B,A,1,nB,enc(K,nA)]);
    2:   msg ← rx([A,B,2,.,.]);
         if (msg[3] = enc(K,nB)) {
            S ← enc(K+1,nA+nB);
            hst.append([B,S]);
            data ← enc(S,msg[4]);
         }
      }
} }
```

Program `Protocol` is the "root" program. The thread executing this program defines the following variables: `chan`, the message-passing channel; `hst`, history of connection establishments; and `K`, the master key shared between client and server. The thread then starts server system `B`; this includes starting a thread in `B` executing function `server` and saving the thread's id in variable `t` of `B`. At this point there are two threads, the original thread and the new thread in the server system. The original thread then starts a client system, after which there are three threads. The original thread then starts an attacker system. All we know about program `Attacker` are its input-output operations and a characterization (below) of what it cannot compute. So after `Protocol`'s main completes, there is a server system with its own thread, a client system with its own thread, and an attacker system with at least one thread.

The client system is in a never-ending loop. In each iteration it does the following: chooses a random `nA`; sends message `[A,B,1,nA]` on `chan`; receives a message `[B,A,1,x,y]` where `x` and `y` are arbitrary; if `y` equals `enc(K,nA)`, the client becomes open to `B` with session key `S` equal to `enc(K+1,nA+x)` and sends message `[A,B,2,enc(K,nB),enc(S,data)]`.

The server system is in a never-ending loop. In each iteration it does the following: receives a `[A,B,1,nA]` message; generates a random `nB`; sends message `[B,A,1,nB,enc(K,nA)]`; receives a `[A,B,2,x,y]` message where `x` and `y` are arbitrary; becomes open to `A` with session key `enc(K+1,nA+nB)` if `x` equals `enc(K,nB)`.

The attacker system can read all messages in `chan`. This is modeled by having function `tx(msg)` append `msg` to α, even those sent by the attacker. So `chan` is always a subset of α. The attacker system can also receive and send `[A,B,...]` messages.

Conventions

System and thread creation:

- `startSystem(Y(.))`, where `Y(.)` is a program: the thread executing this creates a new instantiation of `Y`, referred to as a **system**, executes the system's main code, and returns. The first parameter in `Y` is the created system's id.
- `startThread(f(.))`, where `f(.)` is a function in a program: the thread executing this creates a new thread executing `f(.)` and returns with the created thread's id.

Crypto:

- `random()`: random value.
- `enc(k,x)`: symmetric encryption of `x` using key `k`. (e.g., DES-CBC)
- `dec(k,x)`: symmetric decryption of `x` using key `k`.

Message-passing:

- `tx(msg)`: append message `msg` to `chan`. If the attacker can eavesdrop, this also appends `msg` to α.
- `rx()`: receive any message in `chan`; i.e., removes the message from `chan` and returns it. Blocks if there is no message in `chan`.
- `rx(.)`: argument is a message pattern, indicating the kind of message to receive. For example, `rx([A,B,1,,,])` only receives a `[A,B,1,x,y]` message where `x` and `y` can be any value; blocks if there is no such message in `chan`.

Sequences: A sequence is a list of values enclosed in square brackets, e.g., `[4,2,5,6]`. The entries are indexed starting from `0` at the head (at left); so in the previous sequence, `x[0]` is `4`, `x[1]` is `2`, and so on.

- `[]`: the empty sequence.
- `i..j`: sequence of integers between `i` and `j` (inclusive); e.g., `1..4` is `[1,2,3,4]` and `4..1` is `[]`.
- `x.size`: number of entries in sequence `x`.
- `x.keys`: sequence of indices of sequence `x`, i.e., `[0..x.size-1]`
- `x[i..j]`: sequence of `x[k]` for `k` in `i..j`. Note that it's empty if `i>j`.
- `x.append(y)`: append value `y` to the tail (at right) of sequence `x`.
- `x.remove()`: remove `x[0]` from sequence `x` and return it; undefined if `x` is empty.
- `x` $\subseteq$ `y`: true iff every entry in sequence `x` is in sequence `y`.

Messages: sequences; typically, the first two entries are the sender's id and the intended receiver's id.

---

2 Modeling the attacker

Modeling the attacker is not simple for two reasons. First, we don't know the program executed by the attacker. Second, the attacker knows much more than the values it reads. It also knows the programs executed by the users (client and server) and the parameters (other than keys) with which the programs were instantiated. Thus when the attacker reads a value, it knows the expressions that went into computing the value. Matching this information with information obtained earlier, the attacker may be able to compute expressions that users computed but did not send out, or would compute in the future, or would never compute. The only computations that are off-limits to the attacker are those that would overcome crypto, e.g., decrypt ciphertext without the key.

We illustrate using program `Protocol`. If the attacker reads `nA` and does not know `K` or `enc(K,nA)`, then it has no way of obtaining `enc(K,nA)`. If the attacker reads `[A,B,1,nA]` and `[B,A,1,nB,enc(K,nA)]`, it knows that the session key `S` to be used will equal `enc(K+1,nA+nB)`, but it cannot obtain the value. If the attacker knows `nA` and `nB` from a previous `A-B` connection, and `A` attempts a new connection with the same `nA`, the attacker knows the response, `enc(K,nA)`, and so it can get `A` to become open without `B` becoming open (but it still wouldn't know the session key).

Now to make all this precise for any authentication protocol program (and not just program `Protocol`). The basic idea is as follows. First, each value that is computed by a user is tagged by its "expression tree" (the tree of expressions that went into computing the value). Second, α contains expression trees, and not just values. Third, a value is not computable by the attacker if that value is not in α or is in α and can be reached from α's root node only by traversing an encryption operation without the key. Details follow.

Expression trees: Every value `x` computed in a user system is associated with an expression tree, denoted `x`↑. Each node in `x`↑ is a value-expression pair. The root node is `[x,'exp']`, where `exp` is the expression that yielded the value `x`. For every term `y` in `exp`, there is a child node `y`↑. Here are some examples:

`x←2;`	`x`↑ has root node `[2,'2']` and no child nodes.
`x←random();`	`x`↑ has root node `[x,'random()']` and no child nodes.
`x←a+b+c;`	`x`↑ has root node `[x,'a+b+c']` and child subtrees `a`↑, `b`↑ and `c`↑
`x←enc(K,[a,c]);`	`x`↑ has root node `[x,'enc(K,[a,c])']` and child subtrees `K`↑ and `[a,c]`↑.

What the attacker reads: When the attacker reads a value `x`, it actually reads `x`↑. So if the attacker has read messages `m1`, `m2` and `m3`, then α is the sequence `[m1↑,m2↑,m3↑]`. So `tx(msg)` appends `msg` to `chan` and, if the attacker can eavesdrop, `msg`↑ to α. For convenience, we assume α has a root node, whose children are the entries of α.

What the attacker cannot compute: For brevity, we introduce the following notation. For any value `x`, let ψ(`x`) stand for the statement "the attacker cannot compute `x` from α". The only tool we have (as of now) to establish ψ(`x`) is the following.

- ψ(`x`) holds for a given α if for every appearance of `x` in α, the path from α's root to `x` goes through a `enc(p,q)` node such that ψ(`p`) holds.

  There are some additional technical restrictions on `p` and `q`; these would typically hold unless the protocol is doing something pretty perverse.
  - `q` should not be `dec(p,p)` (for obvious reasons).
  - `q` should not be a "simple" function of `p`, e.g., `p` or `p+2`. (Apparently, this leaks information about `p`.)

This is a recursive definition, with the base case being that `x` is not in α.

3 Invariant assertions

Given an authentication protocol specified by a program, we need an unambiguous method to express desired properties of the program. Here are some examples of desired properties for program `Protocol`, stated informally. To

distinguish variables of different user systems, variable names are prefixed by the system id; e.g., `A.nA` refers to variable `nA` of `A`'s program, and `B.nA` refers to variable `nA` of `B`'s program.

A_1 : `A.nA` always equals `B.nA`, as long as both exist.
A_2 : The attacker never learns `K`.
A_3 : The attacker never learns a session key `S`.
A_4 : If `B` becomes open with a session key `S` then `A` has already become open with the same session key.

We will express such properties by **invariant assertions**, which are statements of the form *Inv* `P`, where `P` is a predicate (a boolean-valued expression) in the program's variables. *Inv* `P` says that at any time, the state of the program satisfies `P`, where the **state** is a snapshot of the values of its variables (i.e., `chan`, `hst`, α, `A`'s variables, `B`'s variables) and the control location of each thread. Given this, here is how the properties stated informally above can be expressed precisely:

A_1 : *Inv* `(exists(A.nA, B.nA) ⇒ A.nA = B.nA)`

A_2 : *Inv* ψ`(K)`

A_3 : *Inv* `(exists(A.S) ⇒` ψ`(A.S))`

A_4 : *Inv*
```
forall(i in hst.keys:
        [B,S] = hst[i] ⇒ ([A,S] in hst[0..i-1]))
```

Given a program with multiple threads, there are invariably many possible evolutions of the program, each corresponding to a different order in which threads execute. Another way of saying this is that if the program is in a state where several threads are active, then there can be several possible next states, each resulting from the execution of a statement by a different thread. (To illustrate, if program `Protocol` has just started the attacker system, then at this point thread, `A.t` can send a message `[A,B,1,nA]`, the attacker can send a message, but `B.t` cannot do anything until a `[B,A,1,.,.]` message shows up in `chan`.)

An **evolution** is a sequence of states and steps that the program can go through *starting from the initial state*. An assertion *Inv* `P` holds for the program iff it holds at every possible state of every possible evolution. Hence the following:

- To prove that assertion *Inv* `P` holds for the program, we have to show that it holds in every possible state of every evolution.
- To prove that assertion *Inv* `P` does not hold for the program, we have to come up with one evolution that ends in a state where `P` does not hold.

Atomicity

Not every statement execution has to be treated as a separate step of an evolution. We can treat a sequence of statements executed by a thread as an **atomic step** if no other thread can observe or affect an intermediate state in the sequence. For program `Protocol`, the following sequences can be treated as atomic steps:

- Initial step: consisting of `Protocol`'s main code, `B`'s main code and function `server` upto statement label `1`, and `A`'s main code and function `client` upto statement label `1`.
- Step `A.1`: an iteration of `client`'s loop, starting from the statement `1` and ending at statement `1`.
- Step `B.1`: the sequence of statements in `server` from `1` to `2`.
- Step `B.2`: the sequence of statements in `server` from `2` to `1`.

In future, we will identify the atomic steps of the program not as above, but by tagging locations in the program as **atomicity points**. Program `Protocol` has three atomicity points: at statement `1` of function `client` and at statements `1` and `2` of function `server`. The **initial state** is the state immediately after the initial step of the distributed program. The directed graph now has only those states where every thread is at an atomicity point. The size of the graph is usually unbounded (because the user and attacker programs are non-terminating and/or the program has unbounded parameters).

4 Analysis of program `Protocol`

Assertion A_1

Does assertion A_1 hold for `Protocol`? It's usually convenient to have a label for the predicate of an assertion one is analyzing. So we ask whether *Inv* B_1 holds for `Protocol`, where

B_1 : `(exists(A.nA, B.nA)` $\Rightarrow$ `A.nA = B.nA)`

Let's trace out the first few steps of `Protocol` assuming the attacker does nothing.

- Initial state:
  `chan = [[A,B,1,xA]];` α `= [[A,B,1,xA]↑]; (A.t at 1); A.nA = xA; (B.t at 1); not exists(B.nA).`
  B_1 holds in this state because `B.nA` does not exist.
- After thread `B.t` executes step `B.1`:
  `chan = [[B,A,1,xB,enc(K,xA)]];` α `= [[A,B,1,xA]↑, [B,A,1,xB,enc(K,xA)]↑]; (A.t at 1); A.nA = xA; (B.t at 2); B.nA = xA; B.nB = xB.`
  B_1 holds in this state because `B.nA = A.nA = xA`.
- After thread `A.t` executes step `A.1`:
  `chan = [[A,B,2,enc(K,xB),enc(S,data)], [A,B,1,yA]];`
  α `= [[A,B,1,xA]↑, [B,A,1,xB,enc(K,xA)]↑, [A,B,2,enc(K,xB),enc(S,data)]↑, [A,B,1,yA]↑];`
  `(A.t at 1); A.nA = yA; (B.t at 2); B.nA = xA; B.nB = xB.`
  B_1 does not hold in this state because `B.nA = xA`, `A.nA = yA`, and `yA` $\neq$ `xA` because `yA` is a new random number.

  Hence *Inv* B_1 does not hold for `Protocol`.

The amount of detail in the states of the above evolution is overkill. Something much briefer is adequate, for example:

- Initial state:
  `A.nA = xA; (B.t at 1).`
- After step `B.1`:
  `A.nA = xA; (B.t at 2); B.nA = xA.`
- After step `A.1`:
  `A.nA = yA` $\neq$ `xA; B.nA = xA.`

  B_1 does not hold.

Make sure your evolution starts from the initial state. It's no good reaching a faulty state if you start from an unreachable state.

Assertion A_2

Does assertion *Inv* B_2 hold for `Protocol`, where

B_2 : ψ(K)

It looks like this holds. So let's try to prove it. Let's look at an arbitrary state S of an arbitrary evolution of `Protocol`. We want to show that S satisfies the following: if `K` appears in $\alpha\uparrow$, then the path from the $\alpha\uparrow$'s root to the node with `K` goes through a `enc(p,q)` node where ψ(p) holds.

Initially, `K` is not in α. The client system sends out `K` in only two ways:

- `enc(K,nB)`, where `nB` is received from the channel.
- `enc(enc(K+1,nA+nB)),data)`, where `nB` is received from the channel, `nA` is randomly generated locally, and data does not depend on `K`.

In both cases, the value `nB` can come from the attacker. But as long as the attacker does not have a simple function of `K`, it cannot slip in an `nB` that will cause `A` to encrypt a simple function of `K` using `K`. (Note that this is really an induction over the steps of the program.)

A similar argument holds for expressions involving `K` that the server sends out.

Thus we can conclude that if `K` appears in $\alpha\uparrow$, then it appears in node of the form

- `enc(K,q)` node or
- `enc(enc(K+1,q),r)`

where `q` and `r` are not simple functions of `K`. So the attacker never learns `K`. Hence *Inv* B_2 holds.

Assertions A_3 and A_4

See homework 2.