Specification and Security-Analysis of an Application using the SHVT Tutorial

Carsten Rudolph, Jürgen Repp
Fraunhofer Institute for Secure Information Technology
E-Mail: Carsten.Rudolph@sit.fraunhofer.de,Juergen.Repp@sit.fraunhofer.de

March 4, 2008

1 Introduction

Objectives of this tutorial:

- Introduction to the specification of an application using inheritance mechanism for roles and transition patterns.
- Explanation of mechanisms for specification refinement.
- Step-by-step tutorial for analysis of a simple example protocol.

This tutorial requires the SHVT and the project and preamble files for the example [1].
Additional information: SHVT Tutorial, SHVT-Handbook (online help)

Scenario:
A simple application where a client requests secret data from a server is specified. The tutorial comprises the following steps:

1. Specification of the application and security goals using basic definitions for the network layer.
2. Implementation of an attacker who tries to get secret information.
3. Adapting the model to the real environment of the application (WIFI, Intranet, SSL, usage of certificates) and specification of the security mechanisms used to ensure the security requirements defined in step 1.

[1] contact shvt@sit.fraunhofer.de for more information

4. After the protocol refinement it is checked whether the security goals are achieved.

2 Specification of the Application

2.1 First Specification using transition patterns

The Standard preambles with data types (*sets.vsp*) and functions (*functions.vsp*) are available for specification of cryptographic protocols. In this tutorial we assume that these standard definitions are used. In this case, the protocol specification consists of three parts:

- Part 1
  Declaration of roles
  Initial state of state components

- Part 2
  Transition patterns specify protocol steps
- Part 3
  Binding of roles with agents' names

2.2 Roles and initial state

Role notion supports specification of actions for subjects acting in a certain role. All actions (elementary automaton) executed in a certain role are assigned to this role in the definition of an transition pattern. Also state components can be assigned to a certain role. This can be compared with class definitions of programming languages. Local state components correspond to local class variables and transition pattern to methods of this class. Differently to class definitions in programming languages the instantiation of roles has to be defined statically in the preamble file. No dynamic creation of roles is possible. In programming languages these mechanisms are used to encapsulate functionality and data. For state transition pattern access to local state components is not encapsulated. Access to local state components by transition pattern, not assigned to the corresponding role, can be modeled. So there are no restrictions when attack models have to be specified.

For every instantiation the corresponding elementary automaton and the state components will be generated from the definitions of transition pattern and state pattern assigned to this role. It is possible to refine the behavior of a transition pattern depending on varying instantiations of the role.

Two roles are defined: `Client` and `Server`. The command `def_role` is used for declaration of roles.

Listing 1: Defintion of roles

```
def_role Client from Network
   { secret : Messages_seq := ::,
     state  : Messages := 'start',
     server : Agents_seq := Server };


def_role Server from Network
   { secret : Messages_seq := ['secret'],
     state  : Messages := 'init' };
```

For each role we specify two local state components `state`, and `secret`. The initial values of these state components are also specified in the role declaration. `Client` knows the identifiers of all agents acting as an server, in order to be able to start protocol runs with these agents. The servers are stored in in the local state component `server` (the name `Client_server` has to be used for this state component in transition patterns). Role names in initial states and transition patterns are automatically replaced with actual agents' names during the generation of analysis scenarios. The used inheritance mechanism (`from Network`) will be explained in 2.5.

2.3 Transition pattern

Protocol steps are specified by transition patterns. Each pattern specifies one single step. The patterns specify conditions for state transitions and changes of the states of state components of the particular agent and of shared state components (e.g. `Network`). All other state components remain unchanged .

```
Syntax:
def_trans_pattern role_a pattern_label
(x1,x2,x3,...,xn)
allocations,
predicates,
actions;
```

This generates a transition pattern `pattern_label` and assigns it to a role `role_a`. The next line declares local variables. These variables are local to the transition pattern. The rest of the pattern contains allocations, predicates and actions on state components (adding or removing elements). A state transition can occur in a particular state, if in this state an interpretation of the local variables exists

such that all allocations are defined, all predicates true and all actions result in a valid subsequent states in accordance with the domains of state components.

2.4 Some important operators

Table 1 shows the most important operators to be used in transition patterns. A_State denotes a state component, c a constant value and $mset$ a multiset. x_u is a previously unbound variable and x_b is already bound by a previous line in the pattern. Apart from the operation given here, other operators can be used as described in the SHVT handbook, section on preambles.

operator	example	description
<<	x_u << A_State	x_u is allocated with one element of A_State (non-deterministic) and deleted in A_State. If A_State is empty, no operation will occur.
	x_b << A_State	If $x_b \in$ A_State, x_b is deleted in A_State.
	c << A_State	If $c \in$ A_State, c is deleted in A_State.
	$mset$ << A_State	If $mset \subseteq$ A_State, this subset is deleted in A_State.
>>	x_b >> A_State	x_b is inserted in A_State
	$mset$ >> A_State	$mset$ is inserted in A_State
	c >> *A_state*	c is inserted in A_State
?	x_u *?* A_State	x_u is allocated with one element of A_State (non-deterministic)
	x_b *?* A_State	if $x_b \in$ A_State, this boolean term is true, otherwise it's false.
	c *?* A_State	if $c \in$ A_State, this boolean term is true, otherwise it's false.
	x_u *?* mset	x_u is allocated with one element of mset (non-deterministic)
	x_b *?* mset	if $x_b \in$ mset, this boolean term is true, otherwise it's false.
	c *?* mset	if $c \in$ mset, this boolean term is true, otherwise it's false.
∼?	x_u ∼? A_State	predicate is false
	x_b ∼? A_State	if not $x_b \in$ A_State, this boolean term is true, otherwise it's false.
	c ∼? A_State	if not $c \in$ A_State, this boolean term is true, otherwise it's false.
:=	x_u := *term*	x_u is allocated with *term*
	x_u := A_State	x_u is allocated with the sequence stored in state A_State .
	A_State := *term*	A_State is allocated with the value of *term* converted to a multiset.

Table 1: Table of important operators

2.5 Abstract Roles

Abstract roles are roles for which no instantiation of state components and transition pattern is performed. Other roles can inherit these roles. Transition pattern and state components of the abstract role are instantiated for these pattern if they are not also declared as abstract roles.
The roles `Client` and the `Server` inherit possible state components, transition

pattern and macros from the abstract role `Network` . No state components are inherited from `Network` .

The shared state component `Network` , used by the abtract role `Network` , is empty in the initial state:

```
def_state Network: net_elem_seq := ::, Network_send, Network_rec;
```

For integration of attackers, `Network` can be split into two state components `Network_send` and `Network_rec` . The command "Split" in the context menu of the preamble definition is used to activate split of state components.
The abstract role `Network` also inherits an abstract role (`Basic`):

Listing 2: Basic network access

```
def_role Basic abstract
   {}
   { send (to,m,net)
     (Basic,to,m) >> net }
   { receive (from,m,net)
     (from,Basic,m) << net }
   { listen (from,to,m,net)
     (from,to,m) << net }
   { relay (from,to,m,net)
     (from,to,m) >> net };

def_role Network abstract from Basic
   {}
   { send(to,m)
     send(to,m,Network)}
   { relay(from,to,m)
     relay(from,to,m,Network)}
   { receive (from,m)
     receive(from,m,Network) }
   { listen (from,to,m)
     listen(from,to,m,Network) };
```

No local state components are declared for the roles `Basic` and `Network` (after the `def_role` ... line). The role Basic provides four macros for accessing state component representing the network interface:

```
send,receive,relay,listen
```

The abstract role `Network` also provides these four macros. In this case the name of the state component representing the network interface is not used as a parameter. In all cases the state component `Network` is used. So it is possible to use the same mechanisms for different network parts by declaring different roles (e.g. DMZ, Intranet etc.). It's not possible to use one of the four macros defined by the role `Basic` directly in transition pattern whose roles' inherit from `Network`

. The first macro found in the inheritance hirachy is used. Initial values of state components of sup roles with equal names are overwritten.
The inheritance hirachy for the roles `Client` and `Server` defined in section 2.2 looks as follows:

2.6 Simple Example

Using the macros defined with the abstract role `Network` our simple Example can be specified by the following transition pattern:

Listing 3: Application

```
def_trans_pattern Client send_get_secret
  ()
  Client_state = 'start',
  Client_state := 'wait',
  send(Client_server,['get_secret']) ;

def_trans_pattern Client rec_secret
  (secret)
  Client_state = 'wait',
  Client_state := 'start',
  receive(Client_server,['data',secret]),
  when variable_bound(secret) = 'true' {
     (Client,conf,[secret]) >> Goals };

def_trans_pattern Server cmd_get_secret
  (from,m)
  listen(from,'Server',m),
  head(m)='get_secret',
  secret ? secret,
  send(from,['data',secret]),
  (Server,conf,[secret]) >> Goals;
```

Several agents could be assigned to the role `Client` and the role `Server` using the construct `def_pattern_bind` . In our example we only will use the default binding where the name of the agent is equal to the role name.

This example can be found in the demo directory of the SHVT. In order to load the example and to edit the specification the following steps have to be carried out using the SHVT:

1. **start SHVT**

- The tool is started using the start script sshvt in the installation directory.

2. **load project**

   - In the SHVT main menu start the Project Manager.
   - In the project manager window use File>Open to load the project file. The file is called ExampleApplication.prj and is in the subdirectory ExampleApplication.
   - The project is shown as a tree. There are four group nodes StandardPreambles, Network, Basic_Example,and Extended_Example. The subtree StandardPreambles contains nodes for standard preamble definitions (data types and functions). To open our example click middle mouse on Basic_Example.pre in group Basic_Example.

The group Network comprises all definitions related to network access and network structure. The already mentioned role `Network` , used for basic specification of network access, is also included in this group. Two variants of the example application are included in the project tree. Basic_Example includes the preamble implementing the application described in this section. Since macros of the abstract role `Network` are used in the transition pattern of Basic_Example.pre it will be necessary to perform a macro expansion for these transition patterns to interpret analysis results. For expansion of the transition pattern Client `send_get_secret` use the command Misc > Expand Transition Pattern in the preamble editor menu bar and select this pattern [2]:

Listing 4: Macro expansion

```
def_trans_pattern Client send_get_secret
  ()
   Client_state='start',
   Client_state := 'wait',
   (Client,Client_server,['get_secret']) >> Network_send;
```

Here the macro call `receive(Client_server,['data',secret])` is expanded to `(Client,Client_server,['get_secret']) >> Network_send` . If more complex inheritance mechanisms and macros are used it is essential for interpretation of analysis results to check macro expansion.

Now we can check our model by computing the reachability graph (graph of all states with the coresponding transitions). Execute the command Analysis in the context menu of group Basic_Example and click on the command Start Exhaustive Analysis in the right pane of the analysis window which is opened.

[2] only the transition pattern with a name beginning with token at the current cursor position are listed

```
Analysis
Start: 25.1.2008 12:43:07
Stop:  25.1.2008 12:43:07

 Reachability Graph SerEx (5)

 3 States computed.
 (1 DeadState)
 (0 Pseudo State Transitions)
 (3 State Transitions)
```

The reachability graph can be drawn using the command Draw Graph in the context menu of `Reachability Graph SerEx (5)` :

Only three state state transitions occur. The initial state `M-1` can by displayed by clicking (clicking mouse-l on compile) `Reachability Graph SerEx (5)` , the last state `M-4` with no successor state can be displayed by clicking mouse-l on `(1 DeadState)` :

```
 M-1
 Client_send_get_secret M-2
 Client_secret:   <::>
 Client_server:   <Server>
 Client_state:    <start>
 Goals:           <::>
 Network:         <::>
 Server_secret:   <[secret]>
 Server_state:    <init>


 M-3 Client_rec_secret
 M-4+
 +++ dead +++

 Client_secret:   <[secret]>
 Client_server:   <Server>
 Client_state:    <wait>
 Goals:           <(Client,conf,[secret]).(Server,conf,[secret])>
 Network:         <::>
 Server_secret:   <[secret]>
 Server_state:    <init>
```

In the last state the client did store the secret received from the server on `Client_secret` . The state component `Goals` has the value `<(Client,conf,[secret]).(Server,conf,[secret])>` . This state component is used to model certain security requirements. In our case the value 'secret' has to be confidential and only client and server should have access to this data. `Goal` is

set in the related protocol steps. The real implementation of the protocol will not include these statements. They are used for analysis purposes only. Because only the agents `Client` and `Server` exist in our model analysis of the security requirements is useless. In the next step a possible attacker is added to the model. This attacker is sniffing the network traffic. Despite the fact that it's clear that the attacker will be successful (no security mechanisms are used) it makes sense to check whether the attacks will be found before adding the necessary security mechanisms to the model, to avoid errors in the attacker model.

Analysis with attacker and automatic search for successful attacks.
To activate the attacker the following steps have to be executed:

- In the project manager choose Options in the menu for Basic.pre in group Network. Activate Split by choosing Yes.
- Activate compile (click for attacker_functions.pre in group Standard_Preambles (left mouse click) and Attacker_sniff_Network.pre in group Basic_Example. This attacker sniffs the network and tries to decode messages using all data available for him. Also generic attacker models where the attacker also tries to manipulate the protocol are available, but not used in this tutorial.
- Choose Set Break in the Analysis window (in the menu on the right side). A break condition has been saved with the project. This condition is automatically loaded. This break condition stops the computation of the reachability graph as soon as the attacker has access to data marked as confidential in the state component `Goals` . Activate the break condition by clicking Set Break.
- Start the computation of the reachability graph by clicking Start Exhaustive Analysis in the command pane on the right side.
- The computation stops and a state is shown in the output pane. In this state the attacker has sniffed the confidential secret from `Network_send` and would put the sniffed message to `Network_receive` if the analysis is continued.
- Choose Compute one Way to Root in the object menu for M-6 (while doing this, a frame is drawn around the complete state. The tool automatically computes one possible path to the initial state. This path shows which steps lead to the successful attack. This path can be explored and displayed step by step.

2.7 Extended Example

In the next step security mechanisms and the system environment will be added to the model:

- Access to the Server is secured by using SSL. Only access by clients with valid certificates will be accepted.
- An access control server (ACS) located in the intranet will distribute these certificates.
- WIFI is used by the client to access the resources. The keys for WIFI encryption are included in the initial state of the system.

Therefore new abstract roles are defined and added to group Network:

Intranet Uses the basic mechanisms for network access. For roles which inherit `Intranet` instead of `Network` . The macros send, receive, ... use the state component Intranet instead of Network.

WIFI Provides mechanisms for network access for roles which inherit WIFI instead of Network. The macros `send` , `receive` , etc. use the state component `Air` instead of `Network` and the network traffic is encrypted using symmetric keys.

PKI Provides macros and abstract roles for checking certificates, and distribution of certificates.

SSL The abstract roles for usage of SSL are defined in this preamble (The SSL handshake is modeled very simplified).

The actual protocol specification with role declarations and transition patterns is included in the group Extended_Example in the project. This folder also contains the preamble Gateway.pre with the transition pattern for relaying data from Intranet to WIFI and vice versa, and the attacker models.

The new transition pattern for initial distribution of certificates by the access control server `ACS` is added to the specification of the application:

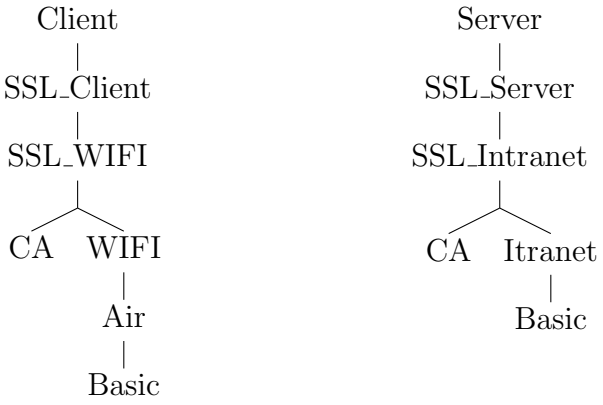
Listing 5: ACS Server

```
def_role ACS from Intranet, CA_Root
  { client_keys : net_elem_seq := (ACS, Client,1111) };

def_trans_pattern ACS send_cert
  (client, key_val)
  ('ACS', client, key_val) << ACS_client_keys,
  send_cert(client, key_val);
```

The role `ACS` which inherits the role `Intranet` and `CA_Root` sends certificates to all clients whose keys are stored in the state component `ACS_client_keys` . The `ACS` is located in the intranet. The actual protocol specification has not been changed, except of changing the inheritance hirachy:

```
def_role Client from SSL_Client
      ....
def_role Server from SSL_Server
      .....
```



The macro expansion for the transition pattern `Client_send_get_secret` looks quite different compared to the expansion of this pattern presented in listing 4:

Listing 6: Macro expansion

```
def_trans_pattern Client send_get_secret
   (SSL_Client_send_key_1,SSL_Client_send_cert_1,
    WIFI_internal_relay_net_2,WIFI_internal_relay_key_2)
    Client_state='start',
    Client_state := 'wait',
    SSL_Client_send_key_1 := sfilter_tag('Server',Client_ssl_keys),
    if SSL_Client_send_key_1=:: {
       ('Server',['get_secret']) >> Client_ssl_messages,
       SSL_Client_send_cert_1 ? Client_ssl_cert,
       (WIFI_internal_relay_net_2,'sym',WIFI_internal_relay_key_2)
           ? Client_wifi_keys,
       (Client,'Server',
        encrypt(WIFI_internal_relay_key_2,
                ['SSL_connect_request',SSL_Client_send_cert_1]))
          >> Air,
    else {
       (WIFI_internal_relay_net_2,'sym',WIFI_internal_relay_key_2)
           ? Client_wifi_keys,
       (Client,'Server',encrypt(WIFI_internal_relay_key_2,
        encrypt(SSL_Client_send_key_1,['get_secret']))) >> Air };
```

In the next step the attacker model for network sniffing is added in the same way as described in section 2.6. Since the state component `Network` is not used

in this model now for instance `Air` and `Intranet` could be observed. An attacker for sniffing the intranet is included in the project:

- In the project manager choose Options in the menu for Intranet.pre in group Network. Activate Split by choosing Yes.
- Activate compile for attacker_functions.pre in group Standard_Preambles (left mouse click) and Attacker_sniff_Intranet.pre in group Extended_Example.
- Execute the command Analysis in the context menu of Extended_Example.pre.
- Choose Set Break in the Analysis window (in the menu on the right side). The same break condition can be used, because only the security requirements stored in the state component `Goals` are checked. The break condition is independent from the attacker type.
- Start the computation of the reachability graph by clicking Start Exhaustive Analysis in the command pane on the right side.

Also in this case a break occurs. State `M-21` is displayed. The secret value 'secret' is stored unencrypted in the state component `Attacker_State` , which is used to store the knowledge of the attacker. The secret is computed by the transition `Attacker_read` from the value of `Intranet_send` :

```
(Server,Client,[encrypt,((Client,Server),sym,8888),[data,secret]])
```

To decode the encrypted message the attacker must have the knowledge of the symmetric key `((Client,Server),sym,8888)` used by client and server. The transition where the attacker acquires this knowledge can be found using the search function of SHVT:

- Determine on path from the initial state to state `M-21` , where the break condition is fullfilled. (Command: Compute one Way to Root in the context menu of the state).
- Search the state where the knowledge of the symmetric key was acquired using the predicate

  ```
  attack(('Server','conf',(('Client','Server'),'sym',8888)).
         ('Client','conf',(('Client','Server'),'sym',8888)),
         Attacker_State,Global)='true';
  ```

  using the command Search in the context menu of the path from the initial state.

A subpath where the key is computed in the last transition of the path is computed. The last search predicate looks similar to the predicate used in the break condition:

```
attack(Goals,Attacker_State,Global)='true';
```

Goals is replaced by

```
('Server','conf',(('Client','Server'),'sym',8888)).
('Client','conf',(('Client','Server'),'sym',8888))
```

In the case of the break condition `Goals` had the value: `(Server,conf,[secret])` , which did state that `secret` has to be treated confidential. Only the server should know `secret` . The client did not yet receive secret, therefore the corresponding value is not stored in `Goals` at this time. The constant which replaces `Goals` does express that the used symmetric key has to be treated confidentially and only client and server should know the key. States where this condition is violated are searched in our path. The result of the search query is a path where this condition is violated in the last state. In the predecessor state of this last state `Intranet_send` has the value:

```
<(Server,Client,
  [SSL_connect_response,
          [sign,(CA_Identity,priv,2222),(Server,pub,2222)],
          [crypt,(Client,pub,1111),((Client,Server),sym,8888)]])>
```

The symmetric key is encrypted, but the attacker seems to be able to decrypt this part of the message.
The next step where the attacker gets the knowledge to do the decryption can be found in the same way.