Computers and Electrical Engineering 36 (2010) 874–885

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Vulnerability analysis of networks to detect multiphase attacks using the actor-based language Rebeca

Hamid Reza Shahriari [a,*], Mohammad Sadegh Makarem [a], Marjan Sirjani [b,c], Rasool Jalili [a], Ali Movaghar [a]

[a] Department of Computer Engineering, Sharif University of Technology, Azadi Avenue, Tehran, Iran
[b] Department of Electrical and Computer Engineering, University of Tehran, Kargar Street, Tehran, Iran
[c] School of Computer Science, Institute for Studies in Theoretical Physics and Mathematics, Niavaran Square, Tehran, Iran

ARTICLE INFO

Article history:
Available online 26 June 2008

Keywords:
Security analysis
Vulnerability analysis
Actor
Model checking
Rebeca language

ABSTRACT

Increasing use of networks and their complexity make the task of security analysis more and more complicated. Accordingly, automatic verification approaches have received more attention recently. In this paper, we investigate applying of an actor-based language based on reactive objects for analyzing a network environment communicating via Transport Protocol Layer (TCP). The formal foundation of the language and available tools for model checking provide us with formal verification support. Having the model of a typical network including client and server, we show how an attacker may combine simple attacks to construct a complex multiphase attack. We use Rebeca language to model the network of hosts and its model checker to find counter-examples as violations of security of the system. Some simple attacks have been modeled in previous works in this area, here we detect these simple attacks in our model and then verify the model to find more complex attacks which may include simpler attacks as their steps. We choose Rebeca because of its powerful yet simple actor-based paradigm in modeling concurrent and distributed systems. As the real network environment is asynchronous and event-based, Rebeca can be utilized to specify and verify the asynchronous systems, including network protocols.

1. Introduction

As computer networks grow in size and complexity, their security analysis becomes more complicated. The evolution of computer networks on one hand and their distributed nature on the other hand, creates opportunities for insiders and outsiders to violate the system security. Many services are perfectly secure when offered in isolation, but when combined with other services, result in an exploitable vulnerability. For example, the file transfer protocol (ftp) and the hypertext transfer protocol (http) offered simultaneously in the same host, may allow the attacker to write in a web directory using the ftp service which causes the web server to execute a program written by the attacker.

Accordingly, security evaluation has become an important requirement in design and management of computer networks. When evaluating the security of a network, it is not enough to consider the single vulnerabilities without considering the other hosts, their relationships, and interactions as well as their network infrastructure. Many of the attacks exploit the global weaknesses in network introduced by interconnections. Nevertheless, the analysis of network security is a complex

* Corresponding author. Present address: Department of Computer Engineering and IT, Amir-Kabir University of Technology, Tehran, Iran. Tel.: + 98 21 64542716; fax: +98 21 66495521.
E-mail addresses: shahriari@aut.ac.ir (H.R. Shahriari), makarem@ce.sharif.edu (M.S. Makarem), msirjani@ut.ac.ir (M. Sirjani), jalili@sharif.edu (R. Jalili), movaghar@sharif.edu (A. Movaghar).

0045-7906/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2008.04.009

and error prone task by hand. Thus, the automatic analysis has been considered. Some people have modeled the network in order to analyze and detect different attacks [1–13]. They could analyze the network model to show some simple attacks. Because of the lack of expressive and simple modeling languages, the complex and distributed attacks have not been considered widely.

In this paper, we use a model based approach to show that how an attacker may use simple attacks to construct a complex attack and reach her/his goals, which were not possible using simple attack methods. We use an abstract model of a network in order to find the complex multiphased attack, named Mitnick attack. To the best of our knowledge, this attack has not been modeled.

Multiphase attacks usually are performed using interaction of different network agents. Such environment is well fitted in actor-based computation paradigm. We use Rebeca [14–16] to model a system consisting of a server, a client, an attacker and their TCP protocol stack layer.

Rebeca (*Reactive Objects Language*) is an actor-based language with a formal foundation, presented in [14–16]. A model in Rebeca consists of a set of reactive objects (called rebecs) which are concurrently executing and asynchronously communicating. Rebeca can be considered as a reference model for concurrent computation, based on an operational interpretation of the actor model [20–22]. It is also a platform for developing object-based concurrent systems in practice. Formal verification approaches are used to ensure correctness of concurrent and distributed systems. The Rebeca Verifier tool, as a front-end tool, translates Rebeca code into languages of existing model checkers, allowing verification of their properties[23,24]. There is also an ongoing project on developing a direct model checker for Rebeca using state space reduction techniques [25–27].

We choose Rebeca because of its powerful yet simple actor-based paradigm in modeling concurrent and distributed systems, and easy to use Java-like syntax for software engineers in modeling, and also the naturally decomposable model and independent modules which is exploited in formal verification and model checking as well as in modeling. The network environment is asynchronous and thus is well fitted in fully asynchronous model of Rebeca. Moreover, the object-oriented nature of Rebeca facilitates the modeling in comparison to other languages such as Promela [31].

The next section surveys the related works that have been done in this field; the third section briefly describes Rebeca. Section 4 presents the model, and its analysis is shown in Section 5 and finally we conclude in Section 6.

2. Related work

The works published on related topics include a set of works which focus on using model checking to verify and analyze the security of systems and other approaches to analyze network vulnerabilities. The CSP process algebra and its model checker FDR have been widely used to verify the security protocols [10]. It belongs to class of formalisms which combine programming languages and finite state machines. Shahriari and Jalili [1] used CSP to model and analyze the Transmission Control Protocol vulnerabilities in presence of a malicious attacker. They used model checker FDR to find some attack scenarios to TCP in broadcast network. They focused on simple attacks, such as connection reset and connection hijack. In [6] CSP is used to discover de-synchronization attacks on intrusion detection systems. Such attacks occur when the state of the intrusion detection system (IDS) becomes desynchronized from that of the system it aims to protect. In [7] the same authors showed that their analysis is data-independent.

Security analysis has been paid more attention recently in two aspects, the individual host and the network vulnerability analysis. Several tools are proposed for detecting individual host vulnerabilities. These include Nessus vulnerability scanner [28], which scans the hosts to detect vulnerabilities. Similar tools such as System Scanner by ISS [29], and CyberCop by Network Associates [30] scan hosts attempting to discover vulnerabilities in the host configuration. However, they do not attempt to investigate how a combination of configurations on the same host or among hosts on the same network can contribute to the vulnerabilities of a network.

The NetKuang system [11] tries to assess beyond host vulnerability. The system is an extension to its authors' previous work on building a rule-based expert system, named Kuang. They extended the Kuang's rule-set to include certain UNIX network security issues, which are undetectable when searching a single host. NetKuang uses a backtrack search algorithm to accomplish the identification of vulnerabilities.

Dacier and Deswarte [12] proposed the concept of privilege graphs. Each node in a privilege graph represents a set of privileges owned by the user, and edges represent vulnerabilities. Privilege graphs are then explored to construct an attack state graph, which represents different ways in which an intruder may reach a certain goal, such as root access on a host. Ritchey and Ammann [13] used model checking for vulnerability analysis of networks via the model checker SMV [17]. They could obtain one attack corresponding to an unsafe state. The experiment was restricted to specific vulnerabilities. However the model checking approach has been used in some other research to analyze network vulnerabilities by Sheyner et al. in [18]. The expressiveness of the language of the model checker has limited their model.

Ramakrishnan and Sekar [4] used a model checker to analyze a single host system with respect to combinations of unknown vulnerabilities. They presented an abstract model of a simple UNIX system. The key issue in their research is checking of infinite space model using model abstraction. However their approach was limited to the configuration vulnerabilities.

In [19] Bellovin has described some implementation independent flaws in Transmission Control Protocol (TCP). He also presented a variety of attacks based on these flaws. The flaws are specified informally and also the single step attacks have been regarded. In this paper we find multiphase attacks on TCP using actor-based model checking.

3. Rebeca modeling language

Rebeca [15,23] is an actor-based language, with independent reactive objects, communicating by asynchronous message passing, and using unlimited buffers for messages. The *actor* model was originally introduced by Hewitt [20] as an agent-based language. It was later developed by Agha [21,22] into a concurrent object-based model. The actor model is proposed as a model of concurrent computation in distributed and open systems. Objects are reactive and self-contained and are called *rebec*, standing for *re*active o*bjec*t. Computation takes place by message passing and execution of the corresponding methods of messages. Each message specifies a unique method to be invoked when the message is serviced. Each rebec has an unbounded buffer, called a queue, for arriving messages.

According to our experiences [1–3,34], Rebeca messaging mechanism and parameter passing is more natural and easier to use for modeling security protocols than Promela and CSP.

Each rebec is instantiated from a *class* and has a single thread of execution. We define a *model*, representing a set of rebecs, as a closed system. It is composed of rebecs, which are concurrently executed, and are interacting with each other. When a message is read from the queue, its method is invoked and the message is removed from the queue. Note that reading messages, thus, drives the computation of a rebec. Rebecs do not provide an explicit control over the message queue. We consider the execution of a method atomic. Sending a message within a method execution is not considered to be a transition, per se. This leads us to coarse grained transitions. Note that this coarse grained granularity of the interleaving of methods is compatible with the asynchronous nature of the communication of Rebeca, which does not contain suspending communication primitives (e.g. a possibly suspending receive state). It also reduces the state space and makes the model simpler.

3.1. Syntax

The detailed syntax for reactive classes (reactive-object templates), rebecs (reactive class instantiations), and models (parallel composition of rebecs) have been presented in Fig. 1. The syntax of a *reactive class* definition is similar to Java, except for the definition of *knownobjects*. The rebecs included in the *knownobjects* part of a reactive class definition, are those rebecs, which their message servers may be called by instances of this reactive class.

After declaring the known rebecs, a list of reactive class fields are declared in *statevars* part. Then the methods, which may themselves contain local variables, are defined as message servers. Variables are typed, and method declarations follow a

```
<model> ::= <reactiveclasses>
            <main>
<reactiveclasses> ::= {<reactiveclass>}+
<reactiveclass>   ::=
      reactiveclass <reactiveclassName>'('<queueLength>')' '{'
            <knownobjects>
            <statevars>
            <body>
      '}'
<knownobjects> ::=
      knownobjects '{'
            {<reactiveclassName> <varname>;}*
      '}'
<statevars> ::=   statevars '{'
                        {<var>;}*
                  '}'
<body>      ::=   {<method>}+
<method>    ::=
      msgsrv <methodName> '(' {<parameter>}* ')' '{'
            {<statement>;}*
      '}'
<parameter> ::= <var> | <var> ',' <parameter>
<var>       ::= <typeName> <varName>
<statement> ::=
      <mir> | <assignment> | <conditional> | <create>
<mir>       ::=
      <varname> '.' <methodName> '(' {<varname>}* ')' ';'
<create> ::=
      <varname> = new <reactiveclassName> '(' <knownobjectsBinding> ')'
<main> ::=
      main '{'
            {<rebec>;}+
      '}'
<rebec> ::=
      <reactiveclassName> <varname> '(' <knownobjectsBinding> ')'
```

Fig. 1. Reactive class, rebec and model definition syntax.

standard syntax. Unlike Java, methods have no return mechanism and therefore no return type. The core language for statements (*statement*) allows the remote method invocation requests (*mir*) which are sending messages, assignments (*assignment*), if-statements (*conditional*), object creation (*create*), and sequential composition.

In *mir*, after specifying the callee (receiver) id, the method name and actual parameters are included. This can be viewed as a message consists of the callee id, message id and the parameters passed to the callee. Although not mentioned explicitly in the message, the caller (sender) passes its rebec identity (self) to the callee (receiver). Caller and callee may be the same rebec, modeling local calls (sends to self).

Every reactive class definition has a method named *initial*. In the initial state of the system, each rebec has an *initial* message in its message queue, thus *initial* is the first method executed by each rebec. After defining the reactive classes, there is a keyword *main* followed by the definition of the Rebeca model which is defined as a finite collection of rebecs that are run in parallel. In declaring a rebec, the bindings to its known rebecs are specified in the list of *knownobjects*.

3.2. Rebeca verifier

Rebeca verifier [20,23] provides an integrated environment to create Rebeca models, specify properties, and translate models to the language of backend model checkers like Spin [31] and SMV [17]. Using the tool, a user can create, edit and debug Rebeca codes, such that the code can be successfully translated to one of the back-end model checker languages. The required properties can be expressed at Rebeca source code level, using temporal specification patterns. These properties can then be automatically translated to the specification language of the selected back-end model checker and the output code can be model checked by it.

We chose Spin as the back-end model checker. In Rebeca Verifier, the Promela code generator is used to produce Promela codes from Rebeca models. Each class in Rebeca is a *proctype* in Promela, and each rebec is a process. Each method of a rebec is mapped to an atomic block in the corresponding process in Promela. The message queues can easily be modeled by channels, according to the length specified by modeler. Within an infinite loop in a process, the message channel is read for the next message to be served. After receiving a message, the atomic block associated to that message will be executed. Processes (rebecs) are instantiated in the init process of Promela. Fig. 2 shows Rebeca language, theory and verification tool.

The Java-like syntax and object-oriented features of Rebeca facilitate the modeling of more complicated, while other languages which we have experienced (such as Promela and CSP) are not object-based and suffer from unconventional syntax and are hard to use. Moreover, the asynchronous nature of computer networks is well fitted in the fully asynchronous semantics of Rebeca message passing.

Since Rebeca syntax is similar to the conventional object-oriented programming languages, it is easy to learn for software engineers. Accordingly, understanding, debugging, modifying, and maintenance of model are straightforward.

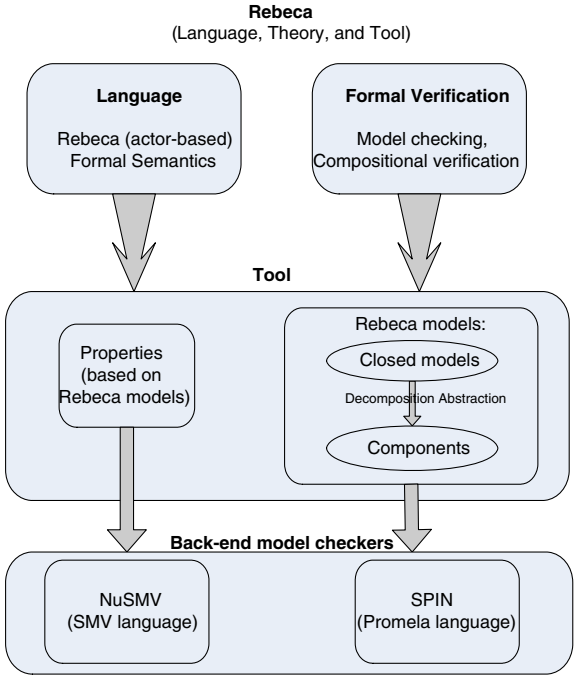

Fig. 2. Rebeca: language, theory, and Rebeca verifier tool.

4. The Mitnick attack

The Mitnick attack is a multiphase attack, which includes SYN-flood attack, TCP sequence number predication, and IP spoofing. This attack uses SYN-flood to deny the service of host, which has a trust relationship to another host. In Mitnick attack scenario, host trusts another host, and attacker tries to compromise this trust. Fig. 3 shows the attack steps. In this figure, Bob (on host B) has trusted to the Alice (on host A) address.

The following scenario describes the attack [32]:

1. Eve (the attacker) starts SYN-flood attack to prevent Alice from responding Bob.
2. Eve sends multiple TCP packets to the target Bob, in order to be able to predict the values of TCP sequence numbers generated by Bob.
3. Eve then pretends to be Alice, by spoofing Alice's IP address, and sends a *SYN* packet to Bob in order to establish a TCP session between Alice and Bob.
4. Bob responds with a *SYN-ACK* to Alice. Eve does not see this packet. Since Alice's input queue is full due to number of half open connections caused by the SYN-flood attack, she cannot send a *RST* (Reset) packet to Bob in response to the spurious *SYN* message.
5. Using the calculated TCP sequence number of Bob (recall that Eve did not see the *SYN-ACK* message sent from Bob to Alice) the attacker sends an *ACK* with the predicted TCP sequence number packet in response to the *SYN-ACK* packet sent to Alice.

Bob is now in a state where it believes that a TCP session has been established with a trusted Alice. Eve now has a one-way session with the target, Bob, and can issue commands to the target.

5. The model

In this section, the Mitnick attack environment is described and then a Rebeca model to describe and analyze it is presented.

5.1. Network model

The environment in which the problem is investigated is the same as our previous work [1] except that here we use a more general network and the environment is not limited to a LAN. Hosts and attacker may be placed in any location in the Internet. Therefore, attacker has not direct access to the network of hosts and cannot sniff packets. In this model, the environment includes set of some hosts communicating via TCP/IP protocol stack, but not restricted to a broadcast environment. Moreover, in this paper, we want to find attacks that are more complex by composing simple attacks. As Fig. 4 shows, hosts may be some clients who get service from servers in presence of a malicious attacker who tries to attack server and get service. The server trusts the client IP addresses and there are no other authentication mechanisms. For example in some services such as *Berkley Remote Shell* and *rexec*, authentication is based on client IP address [33].

In our model, we have some other assumptions without losing generality. In this network, all hosts are connected together via reliable network. In addition, we focus on connection establishment of TCP, and it has been assumed that

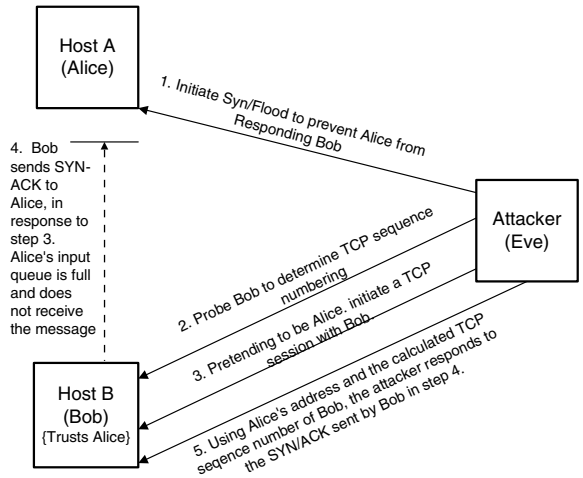

Fig. 3. Illustration of Mitnick attack [32].

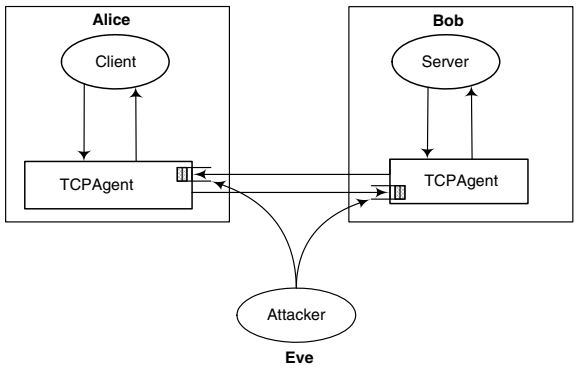

Fig. 4. Rebeca model of network. Each *TCP Agent* has a limited queue to represent TCP buffer.

the messages are sent in a single bulk packet. Thus, we have not modeled the error detection and flow control mechanisms such as acknowledgments.

5.2. Attacker point of view

Attacker has enough information to send packets to clients and server, i.e., he/she knows the client and server addresses. Since the attacker is outside of client/server local area network, he/she can not sniff their communications. In addition, we assumed that attacker is able to send spoofed packets, pretending to be client or server, i.e. the routers in middle of path from attacker to client/server are not able to detect IP spoofing. In our model, the attacker sends arbitrary packets to either client or host. We are interested to find special sequences of packets which yield a successful attack.

5.3. Modeling TCP

As stated, the TCP state machine is simplified and some aspects of TCP, which have no effect on the attack, have been abstracted. The TCP state machine is modeled as a rebec (reactive object) which demonstrates the TCP protocol state machine. It receives some messages either from other TCP agents or from its upper layer agent. The upper layer agents invoke some commands such as connection establishment or closing the connection and get the result. This component is used in any host. Thus, it is better to describe it as an independent module with predefined interface and reuse it in any other host. In our model, the attacker does not need the TCP agent, because he/she sends arbitrary packets to other hosts.

In this phase, time is not considered, and we have not modeled time aspects of TCP/IP such as timeouts. Therefore, the TCP state machine is simplified to involve main states including CLOSED, LISTEN, SYN-SENT, SYN-RECEIVED, and ESTABLISHED. Fig. 5 shows the simplified state machine. Because of the important role of TCP sequence number in many attacks, we have modeled the TCP sequence numbering, as well.

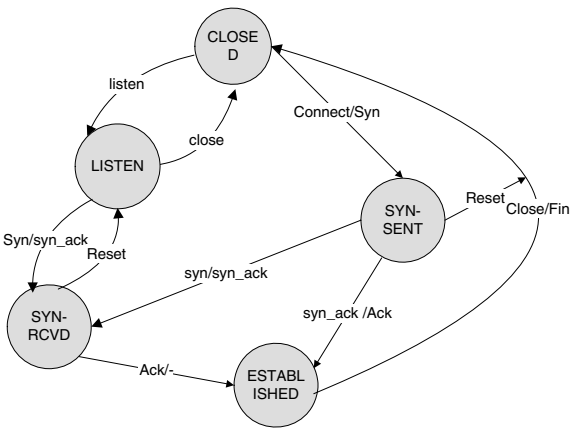

Fig. 5. Simplified TCP state machine.

5.4. Hosts description

As Fig. 4 shows, each host consists of two major parts: TCP/IP stack protocol and an application layer program. TCP/IP stack protocol abstracts the network layers as one component and hides the details of network layers form upper layer. It accepts upper layer command such as connect to other host, close the connection by handling the TCP handshaking and returns the result to application layer. The application layer program uses TCP layer to connect to other hosts and get/give service.

5.5. Rebeca model

The model consists of four rebecs: *Server*, *Client*, *TCPAgent* and *Attacker*. Each host uses a *TCPAgent* to handle TCP state machine except that *Attacker* may not obey TCP rules. *Server* and *Client* use their high-level interface with *TCPAgent* and do not involve TCP details. *TCPAgents* interacts with each other via *takePacket* messages, which model packet delivery in a real network.

The task of *Client* (Alice) is to connect to *Server* (Bob), send a high-level command to it, and then close the connection. Clients repeat this task iteratively. In the other side, *Server* is a passive host that serves the trusted clients by running their commands. It is important to note that client only sends safe commands to *Server*, but *Attacker* (Eve) do not. Attacker sends arbitrary packets to each host in network iteratively. The *selfLoop* message models this iterative task. Table 1 shows the Rebeca model in which some of the details of code have been removed.

6. Analysis of the model

To analyze the model, we use Rebeca Verifier to convert it to Promela and model check it using Spin [31]. To verify the model, we should specify some security goals. Generally these goals are extracted from security policy of an organization. As a result of verification some counter-examples may be achieved. Each counter-example determines a situation in which the specified properties are violated. First we check for simple security goals and try to obtain simple attack scenarios to violate them. As in our model the message passing between rebecs corresponds to the actions of network components, the attack scenarios can be directly derived from counter-examples.

We use the temporal logic to mention these goals as properties of the model. The first property to be checked is about service availability. Server always should be available and client should be able to get service. The following temporal formula specifies this property:

$$SrvAvailability_1 = \Box\ (clientError = false)$$

This means that client always connects to the server without any error (in our model, client periodically tries to connect to server) (where $\Box$ represents *always*). By checking the model, we can find some scenarios to violate the property. The summarized scenario in Rebeca is depicted as following:

6.1. Attack scenario 1:

ClientTCPAgent.connect (Server_IP)
ServerTCPAgent.takepacket (SYN)
Eve.Initial//*predicts the correct sequence number and sends RST packets in the slefloop*
Eve.selfloop
ClientTCPAgent.takepacket (RST)
Alice.ConnectionError

In this scenario attacker sends a *Reset* packet (RST) to Alice while she is handshaking with Bob, and causes the connection to be closed. The scenario can be rewritten in more conventional style:

Alice → Bob: *alice.bob.SYN*
Eve → Alice: *bob.alice.RST*

Attacker could also send the *Reset* packet to the server, Bob. This scenario is similar to the above scenario and is not presented here.

The second property to be checked is $SrvAvailability_{2:}$

$$\begin{aligned} SrvAvailability_2 = \Box(& \\ & (s\_state01\_TcpAgent[1] = SYN\_SENT) \\ & \Rightarrow \\ & \Diamond(s\_state01\_TcpAgent[0] = ESTABLISHED)) \end{aligned}$$

Table 1
Summarized Rebeca model of the client, host, TCP state machine and the attacker

```
Reactiveclass TcpAgent(4){
  knownobjects{
  Server Bob;
  Client Alice;
  Attacker Eve;
  TcpAgent anotherAgent;
  }
  statevars{
    byte lastSeqNoSent01;
    byte lastSeqNoSent2;
    byte lastSeqNoRcvd01;
    byte lastSeqNoRcvd2;
    byte state01;
    byte state2;
    byte myIP;
  }
  msgsrv initial(byte s_ip){//get initial state
    myIP=s_ip;
    state01=2;
    state2=2;
  }
  msgsrv connect(byte ip){
    //Connect to its ip
    }
  msgsrv takePacket(byte sIP, byte pType, byte senderSeqNo, byte ack, boolean
   command){
  //TCP State Machine implemented by a conditional structure
  }
  msgsrv close(byte ip){
    //Close the connection
  }
  msgsrv sendDataPacket(byte ip,boolean command){
    //Send data packets
  }
}
Reactiveclass Server(3){
  knownobjects {
    TcpAgent myAgent;
  }
  statevars{
    boolean safeCommand;
    boolean userConnected;
    boolean initialized;
  }

  msgsrv initial(){}
  msgsrv connected(byte ip){
  / inform Server that a new connection was established
  }
  msgsrv takeDataPacket(byte sIP,boolean command){
  //server new data packets are delivered to Server

  }

  msgsrv closed (byte sIP){
  //to inform Server that an existing connection was closed
  }
  msgsrv connectionError(byte ip){
  //Server is informed about connection errors
  }
}
Reactiveclass Client(3){
  knownobjects{
    TcpAgent myAgent;
  }
  statevars{boolean clientError;}
  msgsrv initial(){...}
  msgsrv connected(byte ip){
  //to inform Clinet that a new connection was established
  }
  msgsrv closed(byte ip){
  //to inform Client that an existing connection was closed
  }
  msgsrv takeDataPacket(byte sIP,boolean command){
  //new data packets are delivered to Client
  }
  msgsrv connectionError(byte ip){
  //Client is informed about connection errors,
  //This Message server sets clientError variable
  }
}
Reactiveclass Attacker(5){
  knownobjects{
    TcpAgent bobAgent;
    TcpAgent aliceAgent;
  }

  statevars{
    boolean newDestination;
    byte newSource;
    byte newPType;
    byte newSenderSeqNo;
    byte newAck;
    boolean newCommand;
  }
  msgsrv initial(){
    self.selfLoop();
  }
  msgsrv takePacket(byte sIP, byte pType, byte senderSeqNo,
  byte ack,boolean command)
  {
  //Packets are delivered to attacker through this message server
  }
  msgsrv selfLoop(){
  //Sending non-deterministically generated packets to other
hosts
  self.selfLoop();
  }
}
//Main body creates reactive objects
main{
  TcpAgent
serverTcpAgent(Bob,Alice,Eve,clientTcpAgent):(0);
  TcpAgent
clientTcpAgent(Bob,Alice,Eve,serverTcpAgent):(1);
  Server Bob(serverTcpAgent):();
  Client Alice(clientTcpAgent):();
  Attacker Eve(serverTcpAgent,clientTcpAgent):();
}
```

This means that if the client requests for a connection without receiving any error, the connection should be established finally, i.e. its TCP state should be *ESTABLISHED* (where $\Diamond$ represents *finally*). By checking the model for $SrvAvailability_2$ property, another counter-example was found. This counter-example is the scenario of SYN-flood attack. In this attack, attacker floods the victim by many *SYN* packets. Each *SYN* packet means a new connection request and causes a half open connection data to be stored in the victim. Thus, these data can consume the system resources and cause the new connection requests to be rejected. The following shows the scenario in Rebeca:

6.2. Attack scenario 2:

Eve.initial//Eve sends a SYN packet to Bob in each selfLoop
Eve.selfLoop
Eve.selfLoop
...
Eve.selfLoop
Alice.initial
SerevrTcpAgent.initial
ClientTcpAgent.initial
Bob.initial//*Sends a SYN to Alice, but Alice's queue is full and the SYN is dropped.*
TcpAgent.connect
ServerTcpAgent.takePacket
ServerTcpAgent.takePacket
ServerTcpAgent.takePacket
ServerTcpAgent.takePacket
Eve.selfLoop
...

As it is shown in the Scenario 2, the client waits for *SYN_ACK*, but he does not receive anything. It means that the server has been flooded and cannot reply to the client because it sends many *SYN* packets without following handshaking steps.

The last property to be checked is that safe commands are always executed on the server, i.e. attacker should not be able to execute his unsafe command on the server. The famous Mitnick attack goal is to violate this property and force the server to execute his dangerous command. The goal is described as follows:

$$Safety = \Box\ (safeCommand = true)$$

It means that always the safe commands should be executed on the server. In our model, trusted user sets *safeCommand* to *True* during each connection, but attacker executes its dangerous command. By checking the model to find counter-examples, we can find the following scenario to violate the *Safety* property:

6.3. Attack scenario 3 (Mitnick attack):

ServerTcpAgent.initial
ClientTcpAgent.initial
Bob.initial
Eve.initial
Alice.initial
//*Eve sends many SYN packets to Alice's TcpAgent*
Eve.selfLoop
...
Eve.selfLoop
//*Bob sends syn_ack to client, it will be dropped because client's queue is full:*
ServerTcpAgent.takePacket
//*Eve predicts sequence number and send ACK to serverTcpAgent, this prediction modeled using a nondeterministic choice.*
Eve.selfLoop
//ServerTcpAgent receives ACK and send connected message to Server
ServerTcpAgent.takePacket
Bob.connected
//*Eve sends a dataPacket that includes a malicious command to serverTcpAgent:*
Eve.selfLoop
ServerTcpAgent.takePacket //*the malicious command is delivered to Bob(Server)!*
Bob.takeDataPacket //*the server in now hacked by executing malicious command!*

The scenario can be rewritten in a more conventional style:

1. Attacker performs syn-flood attack:
   Attacker sends many *SYN* packets:
   Eve → Alice: *bob.alice.RST*
2. Server sends *SYN_ACK* to the client, it will be dropped because client's queue is full:

   Bob → Alice: *bob.alice.SYN_ACK*

3. Attacker connects to the server by predicting correct sequence number. This prediction is modeled as a nondeterministic assignment in Rebeca model:

   Eve → Bob: *Eve.bob.ACK*
4. Attacker sends a malicious command to the server to be executed:

   Eve → Bob: *Eve.bob.dangerousCommand*

This is a multiphase Mitnick attack, which gains some simple attacks to reach its goal. First, the attacker floods the client by SYN packet to prevent it to respond to the server. Then by spoofing client address, and predicting server TCP sequence number connects to the server and invokes its dangerous command.

As we do not have any attack dependent assumption, this approach can be generalized to find other attacks, either new or known. It is worthy to note that this independency is not perfect, because any model has some abstractions and assumptions that they confine model usage and may loss some useful information.

7. Conclusions and future works

In this paper, we presented a model of communicating hosts. In this model, some hosts get service from the server according to the server trust. An attacker tries to exploit this trust to get service from the server. We modeled the environment using an actor-based language Rebeca, and verified it to find attack scenarios. Rebeca helped us to have a detailed model of hosts and verify it to find complex attack scenarios. First, we found simple attacks to violate simple security properties. In the first attack, which is named connection reset, the attacker prevents client from connecting to the server by sending *Reset* packets to the client or server. In the second one, attacker exhausts the host resources by establishing many half-opened connections to it. It sends many *SYN* packets without following handshaking steps.

After that, the model has been checked to detect more complex attack scenario in which simple attacks are chained to construct a multiphase attack. This attack was the famous Mitnick attack. The attack chains some simple attacks such as SYN-flood, IP spoofing and TCP sequence number prediction.

We showed how an attacker may chain some simple attacks to launch a complex attack. In comparison to the earlier works [7,11–13,34], the Mitnick attack has been modeled and analyzed in previous works. Moreover, we have gained the formal language Rebeca, which is an actor-based language and actors interact via message passing. As the network environment is asynchronous, it is well fitted in fully asynchronous model of Rebeca. Additionally, the object-oriented nature of Rebeca facilitates the modeling in comparison to other languages such as Promela [31].

The main drawback of our approach is the simplicity of our model, as we have omitted the time in our model and also deleted some related states from TCP state machine. We continue working on it and try to overcome them. The other one may be the scalability of the approach to make the approach more applicable in real applications. As one of important features of Rebeca is compositional verification, we will try to gain this feature and verify larger models by composing the verified components.

Several issues can be considered for future works. In the first step the model can be checked against more security properties which have been extracted form the security policy. Thus, precise and applicable specification of security policy will be required. Another important extension to our model is considering time and modeling other states of TCP state machine. Consequently, we will be able to analyze more Denial of Service (DoS) attacks and their solutions. Moreover, our approach can be used for other protocols to find attack scenarios.

Acknowledgement

This research was in part supported by a Grant from IPM (No. CS1383-4-04).

References

[1] Shahriari HR, Jalili R. Using CSP to model and analyze transmission control vulnerabilities within the broadcast network. In: Proceedings of IEEE international networking and communication conference (INCC'2004); June 2004. p. 42–7.
[2] Zakeri R, Shahriari HR, Jalili R, Sadoddin R. Modeling TCP/IP networks topology for network vulnerability analysis. In: Proceedings of 2nd international symposium of telecommunications (IST2005); September 10–12. 2005. p. 653–8.
[3] Shahriari HR, Sadoddin R, Jalili R, Zakeri R, Omidian AR. Network vulnerability analysis through vulnerability take-grant model (VTG), In: Proceedings of *International Conference* 7th international conference on information and communications security (ICICS2005), LNCS, vol. 3783; 2005. p. 256–8.
[4] Ramakrishnan CR, Sekar R. Model based analysis of configuration vulnerabilities. J Comput Security 2002;10(1/2):189–209.
[5] Ryan P, Schneider S. Modeling and analysis of security protocols: a CSP approach. Addison-Wesley; 2001.
[6] Rohrmair G, Lowe G. Using CSP to detect insertion and evasion possibilities within the intrusion detection area. In: Proceedings of BCS workshop on formal aspects of security; 2002.
[7] Rohrmair G, Lowe G. Using data-independence in the analysis of intrusion detection systems. Theor Comput Sci 2005;340(1):82–101.
[8] Jajodia S, Noel S, O'Berry B. Topological analysis of network attack vulnerability. In: Kumar V, Srivastava J, Lazarevic A, editors. Managing cyber threats: issues, approaches and challenges. Kluwer Academic Publisher; 2003.
[9] Noel S, Robertson E, Jajodia S. Correlating intrusion events and building attack scenarios through attack graph distances. In: Proceedings of the *Annual Computer Security Applications Conference*, 20th annual computer security applications conference. Tucson, Arizona; December 2004. p. 350–9.
[10] Ryan P, Schneider S. Modeling and analysis of security protocols: a CSP approach. Addison-Wesley; 2001.

[11] Zerkle D, Levitt K. NetKuang –a multihost configuration vulnerability checker. In: Proceedings of the sixth USENIX UNIX security symposium. San Jose, CA; 1996. p. 195–204.
[12] Dacier M, Deswarte Y. Privilege graph: an extension to the typed access matrix model. In: Proceedings of third european symposium on research in computer security (ESORICS 94), Lecture notes in computer science, vol. 875. Brighton, UK: Springer-Verlag; 1994. p. 319–34.
[13] Ritchey RW, Ammann P. Using model checking to analyze network vulnerabilities. In: Proceedings of IEEE symposium on security and privacy; May 2001. p. 156–65.
[14] Sirjani M, Movaghar A. An actor-based model for formal modeling of reactive systems: Rebeca. Technical Report CS-TR-80-01. Tehran, Iran; 2001.
[15] Sirjani M, Movaghar A, Shali A, de Boer FS. Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae 2004;63(4):385–410.
[16] Sirjani M, Movaghar A, Mousavi M. Compositional verification of an object-based reactive system. In: Proceedings of the workshop on automated verification of critical systems (AVoCS'01). Oxford, UK; 2001. p. 114–8.
[17] SMV. SMV: A Symbolic Model Checker. Available from: <http://www.cs.cmu.edu/modelcheck/>.
[18] Sheyner O, Haines J, Jha S, Lippmann R, WingJ. Automated generation and analysis of attack graphs. In: Proceedings of IEEE symposium on security and privacy; 2002. p. 273–84.
[19] Bellovin SM. Security problems in the TCP/IP protocol suite. Comput Commun Rev 1989;19(2):32–48.
[20] Hewitt C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. PhD Thesis, MIT; 1971.
[21] Agha G. Actors: a model of concurrent computation in distributed systems. Cambridge, MA, USA: MIT Press; 1990.
[22] Agha G, Mason I, Smith S, Talcott C. A foundation for actor computation. J Funct Program 1997;7:1–72.
[23] Sirjani M, Movaghar A, Iravanchi H, Jaghoori M, Shali A. Model checking in Rebeca. In: Proceedings of parallel and distributed processing techniques and applications (PDPTA'03); June 2003. p. 1819–22.
[24] Sirjani M, Movaghar A, Shali A, de Boer F. Model checking, automated abstraction, and compositional verification of Rebeca models. J Universal Comput Sci ;11:1054–82.
[25] Sirjani M, Shali A, Jaghouri MM, Iravanchi H, Movaghar A. A front-end tool for automated abstraction and modular verification of actor-based models. In: Proceedings of fourth international conference on application of concurrency to system design (*ACSD*)2004. Hamilton, Canada: IEEE Computer Society; June 2004. p. 145–8.
[26] Jaghoori M, Movaghar A, Sirjani M. Modere: the model checking engine of Rebeca. In: Proceedings of ACM symposium on applied computing – software verification track; 2006. p. 1810–5.
[27] Jaghoori M, Sirjani M, Mousavi MR, Movaghar A. Efficient symmetry reduction for an actor-based model. In: Proceedings of 2nd international conference on distributed computing and internet technology. Lecture notes in computer science, vol. 3816; 2005. p. 494–507.
[28] Derasion R. [online] The Nessus attack scripting language reference guide; 2000. Available from: <http://www.nessus.org>.
[29] Internet Security Systems. [online] System scanner information. Available from: <http://www.iss.net>.
[30] Network Associates. [online] CyberCop Scanner Information. Available from: <http://www.nai.com/asp_set/products/tns/ccscanner_intro.asp>.
[31] Spin Model Checker. [online] Available from: <http://netlib.bell-labs.com/netlib/spin>.
[32] Undercoffer J, Pinkston J. Modeling computer attacks: a target-centric ontology for intrusion detection. In: Proceedings of 2002 CADIP research symposium; 2005.
[33] Bishop M. Computer security: the art and science. Addison-Wesley; 2003.
[34] Shahriari HR, Jalili R. Vulnerability take-grant (VTG): an efficient approach to analyze network vulnerabilities. Comput Security ;25:349–60.

Hamid Reza Shahriari is currently an Assistant Professor at the Department of Computer Engineering and Information Technology in Amir-kabir University of Technology in Tehran, Iran. He received his Ph.D. in computer science from Sharif University of Technology in 2007. His research interests include information security, vulnerability analysis and formal methods in security.

Mohammad Sadegh Makarem received his MS in Software Engineering from Department of Computer Engineering, Sharif University of Technology in 2005 and his BS in Computer Engineering from University of Tehran in 2003. His research interests include formal modeling and verification of component based systems and software architecture analysis. He is currently working on software development process and team software process.

Marjan Sirjani is an Assistant Professor at School of Electrical and Computer Engineering at University of Tehran, Iran. She is also a senior researcher at School of Computer Science at IPM. She received her Ph.D. in Computer Engineering from Sharif University of Technology, Tehran, in December 2004. Her fields of interests are applying formal methods in Software Engineering and System Design, Object-Oriented Modeling, Component-Based Modeling, Formal Verification, Abstraction and Compositional Verification.

Rasool Jalili received his Ph.D. in Computer Science from The University of Sydney, Australia in 1995. He then joined the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. He is now an Associate Professor, doing research in the areas of Distributed Computing and Information Security in his network security laboratory; nsc.sharif.edu.

Ali Movaghar is currently a Professor at the Department of Computer Engineering in Sharif University of Technology in Tehran, Iran. He received his Ph.D. in computer, information and control engineering from the University of Michigan in 1985. His research interests include the evaluation and verification of distributed real-time systems.