Attack Generation From System Models

Sameer K.C.

Kongens Lyngby 2011
IMM-MSC-2011-44

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

In a real world system such as organizational buildings, it is often hard to find the culprit who breaches the security at a particular location in the system. Formal methods are of little help because analyses and formalizations are available for software systems but not for real world systems. There are some approaches available such as threat modelling that try to provide the formalisation of the real-world domain, but still are far from the rigid techniques available in security research.

The situation gets even worse in case of insider threats. Insiders have better access, trust and intimate knowledge of surveillance and access control mechanisms of the system. Therefore, an insider can do much more harm to a system and its assets, and, even worse, an insider attack can be very difficult to trace.

With the help of static analysis techniques we can analyse an abstracted system model that allows for easy modelling of real-world systems. This abstraction makes the real world system an analysable model with an underlying semantics that will help us to carry out different analysis on the system. We can in turn define a modelling language that can be the basis for detecting attack threats at various locations in a system.

This thesis work focuses on generating potential attacks in a real world system by applying static analysis techniques to a system model, i.e., identifying which actions may be performed by whom, at which locations, accessing which data. In this work we developed a tool, written in Java, which is used to generate attacks at specified point in the system, i.e., what kind of attacks can happen at what locations and by what actors.

Acknowledgements

I sincerely thank Christian W. Probst for being my main thesis supervisor. I had taken courses such as *Analysis Technique* and *Robust Software System* with him before. He is also the author of *An Extensible Analysable System Model*, which provides the theoretical foundation for this thesis and for which the tool proposed in the thesis is being developed. His immense knowledge of the subject, ideas, suggestions and excellent supervision made this thesis happen. I am thankful for all the efforts and time he put towards me during the thesis period amidst his busy schedules.

I would also like to express my gratitude to my home supervisor professor Tuomas Aura from Alto University, Finland for the suggestions and helps he provided during the thesis period.

I also extend my thankfulness to all the online forums and communities that provided help in acquiring information regarding thesis work.

Last but not least I wish to use this opportunity to express gratitude and love to my friends and family for their support, motivation and for everything.

Lyngby, July 2011 Sameer K.C.

Contents

Chapter 1

Introduction

In today's world information plays vital role. The information is core to any business as well as the society. The world is seamlessly connected to offer information or access data resources physically and virtually at any location on the globe. Day to day operations from surfing the internet to big economical transactions, all of these tasks depend in some way on the confidentiality, integrity and availability of the information. Any compromise in the information may lead to disastrous consequences.

Information has become so vital in today's world that it has led to information theft. A wide range of attacks happen everyday in the software world to gain unauthorized access in order to get the information. Therefore, a wide range of access control mechanisms have been developed in years as a mean to restrict the access of data. These access control do well most of the time when the intruder is mostly outsider. But problems arise when the intruder is insider. Insider threat is one of the toughest challenge for security people because the insider has knowledge and some access rights within the organization. General idea is to secure the information by making the access control tight. But sometimes these measure do not serve their purpose, that is in case of such a insider attack, investigators often have to fall back on log file analysis to find out the possible attackers.

1.1 Insider Threat

Of all the attacks an organization security policies can handle insider attack is the most dangerous one. The insider problem has garnered the interest of many researches and agencies (Anderson and Brackney [2004]). Insiders have already the advantage of knowing the system well and can thus take advantage of the loop holes in the security. In this way an insider attack can pose a bigger threat than outside attacks and can result in catastrophic damages. Until recently, there has been relatively little focused research into developing models, automated tools, and techniques for analysing and solving the insider problem. We still depend on log file audits to deal with such a serious threat (Anderson and Brackney [2004]).

1.2 Real World

The biggest problem of insider threat lies in the real world such as organization buildings and infrastructures such as human actors, folder, keys, printouts etc. There are many analyses and formalization approaches that deals with software systems while there is little work done for the real system. This may be because software systems often are rigorous whereas same is not true in the real world scenario. Work such as "Threat Modelling" (Swiderski and Snyder [2004]) have been used to formalize the real world systems but lack the rigid techniques and formalization available in formal methods.

Probst, Hansen, and Nielson [2007] have proposed a solution to counteract insider problem in a real world system model. At first a formal model of systems is developed that represents real-world scenarios. These high-level models are then mapped to acKlaim, a process algebra with support for access control, that is used to study and analyse properties of the modelled systems. The analysis of processes identifies which actions may be performed by whom, at which locations, accessing which data. This allows to compute a superset of audit results before an incident occurs.

1.3 Thesis Work

The thesis focuses on developing a tool that is incorporates Probst and Hansen [2008] solution and tries to provide a tool base to carry out such analyses. The tool based on "EXASYM" (short hand for Extensible Analysable System Model discussed in Probst and Hansen [2008]) will be used to calculate a superset of attacks that can be caused by an attacker at a given location in the specified system. Thus the tool can be viewed as mechanism to generate possible sets of attacks at a particular location or data in the given system specification.

1.4 Related Work

The design and implementation of our tool is influenced by a number of research works. In this section we will briefly look upon those influences.

KLAIM, Nicola et al. [1998], abbreviated for *Kernel Language for Agents Interaction and Mobility*) is a process calculus that describes mobile agents and interaction. Some of the several other dialects of the KLAIM family are μKlaim by Gorla and Pugliese [2003], OpenKlaim by Bettini et al. [2002] and acKlaim by Probst et al. [2007]. Our tool is based on the concept of acKlaim. acKlaim is an extension to μKlaim calculus with addition of access control mechanisms and a reference monitor semantics (inspired from Hansen et al. [2006]) to ensure the compliance with the access policies in the system. Like other KLAIMs, acKlaim also consists of three layers: nets, processes, and actions. The addition in acKlaim to other Klaim is that processes are annotated with a name, in order to model actors moving in a system, and a set of keys to model the capabilities they have.

Portunes by Dimkov et al. [2011] is another language that deals with insider problem. Portunes is also based on the KLAIM family. Porturnes uses the containment relation as describe by Dragovic and Crowcroft [2005]. Portunes describes the system as layer of containments whereas our method breaks the system infrastructure into components such as locations, actors, data and key. In Portunes, there are three layers: *spatial*, *object* and *digital*. The *spatial layer* describes structure of organization like rooms, halls and elevators. The *object layer* includes objects in the organization, such as people, computers and keys. The *digital layer* presents the data of interest. The idea behind creating these layers is to allow actions that happens only in one single layer (for e.g. copying, reading is only done for data) or between the specific layers (a person can move data, but data cannot move a person). Our approach doesn't have any containment relation however we have defined set of actions such as move action is for locations not for data, read action is for data not for location etc. As in our approach, **Portunes** also provides semantics to the system by presenting system specification as a graph based and providing annotations or meanings to the edges and nodes in the graph. Also, **Portunes** like our method uses access control policies to describe security mechanisms.

1.5 RoadMap

The thesis is structured as below:

Chapter 2 gives the background theory on which the development of our tool is based. In this chapter we will see theory about abstract system, system components, modelling language grammar and modelling analyses.

Chapter 3 covers the details of analysis and design pattern of our tool. In this chapter we will see the steps and considerations taken for the implementation of the tool. We will discuss the principles on which we base our tool and also present different analyses that our tool can do. Similarly, we will exhibit how our tool is designed to incorporate future extensions and enhancements of the tool.

Chapter 4 provides a detail on the implementation of the theory so far discussed in earlier chapters. In this chapter we will discuss the tool we developed which can be used to generate attacks in the system model. The chapter includes the detailed functionalities of the tool and explain different external libraries used in our tool.

In **Chpater 5** we will evaluate the functionalities of our tool. We will set some basic tasks to be performed by the tool, evaluate the outcome and check whether the tool outputs as expected.

Finally, **Chapter 6** provides conclusion of our work. Also, functionalities that we have thought of but haven't implemented yet due to time limitations are briefly described here.

Chapter 2

Background Theory

In this chapter we will cover the background theory on our research work.

2.1 Insider Problem

The IT industry has revolutionized our world. The use of IT infrastructures in day to day businesses like banking, communication, transportation, healthcare etc. shows our increasing dependency on computer systems. With all these increasing dependencies, the computer network world is being filled with massive loads of data every day. Some data may be public or worthless, but some data are of high importance, clandestine and worth a lot. The safety of these data is the concern of anyone who owns the data as there are growing number of cyber attacks to steal the data. This has resulted in launch of a variety of security tools such as IDS(Intrusion Detection System), anti virus software, firewalls etc., which helps to detect the threat and counter measure it wherever possible. Many studies have been done to know the origins of these attacks. The CSI 2009 survey [Com, 2009] states that 30% of the respondents found that malicious activities (such as pornography, pirated software, etc.) is caused by insiders and 25% of the respondents felt that over 60 percent of their financial losses were due to non-malicious actions by insiders. The intention of an insider may not be necessarily malicious but may cause a loss. So, if the same insider has a malicious intent then losses can be hazardous.

With the feel of need to address the problem of *Insiders* there are lots of studies going on in research communities. In Bishop [2005], Matt Bishop defines insider as:

Definition 2.1 An insider with respect to rules R is a user who may take an action that would violate some set of rules R in the security policy, were the user not trusted. The insider is trusted to take the action only. when appropriate, as determined by the insider's discretion.

The definition is relative to the set of rules. An example would clarify the definition. Three users A,B and C can read, read/write and read/write/edit an article. So, in this way we have

```
Rule 1(R1): {read}
Rule 2(R2): {read,write}
Rule 3(R3): {read,write,edit}
```

Any person who does not satisfy rule **R1** is *less insider* than any one who satisfies **R2** or **R3**. Similarly, a person who satisfies rule **R3** is classified as *more insider* than the one who can only satisfy **R2**. So the one who meets rule **R3** automatically meets **R2** and **R1** and so on, forming a linear hierarchy. So, the insider term is more relative and it will be more suitable to call an entity an *insider with respect to the set of rules R* or if the restriction rules are inclusive then one entity is insider relative to another entity.

Bishop further completes the definition of the insider as:

Definition 2.2 The insider threat is the threat that an insider may abuse his discretion by taking actions that would violate the security policy when such actions are not warranted. The insider problem is the problem of dealing with the insider threat.

Traditional methods of policy-based enforcements will not work effectively to deal with insider problems as these policy-based enforcements are based on granting access on trust basis which insiders by definition violate. The notion of insider and insider problem in this work is based on the definition that Bishop proposed.

2.2 Modelling Systems

In this section we will see how we define a system model that represents the real world system and holds its properties. This system model will be analysable and later on we will see how we apply analyses techniques on this system model. This sections is based on Probst and Hansen [2008]

2.2.1 General System Model

A real world system that we are talking about such as buildings has some basic properties. Every building has locations for example server room, reception and then these locations are connected. Then there are actors or the people who can move around these locations. These actors have certain access grants to perform legitimate actions inside the organizational space. These actions can include moving from one location to another as well as data operations such as read and write.

Figure 2.1 from Probst et al. serves as an example to illustrate a higher level overview of a real building. Throughout this thesis we will be consulting this example. As can be seen from the figure, a real world building has physical locations (entrance, hallway, server room with printer, user office and janitor workshop), data network connection between computers at user office and server room and a printer is connected to the computer in server room. Entrance access policy is face recognition system while in other rooms there is cipher lock access policy whereas a physical key is needed to access the janitor's room. The people in the system are a user and a janitor.

2.2.2 Analysable System Model

In the previous section we have seen a real building scenario. Now in this section we will see the abstract model of the general model. Abstraction is done in order to figure out the components that the system is comprised of.

Components can be location components such as the server room and the user office, data components such as keys and real data, mobile components such as processes and actors. Data is located at a location. Keys are either assigned to actors or stored at a location. Locations are connected with edges to provide movement for the actors.

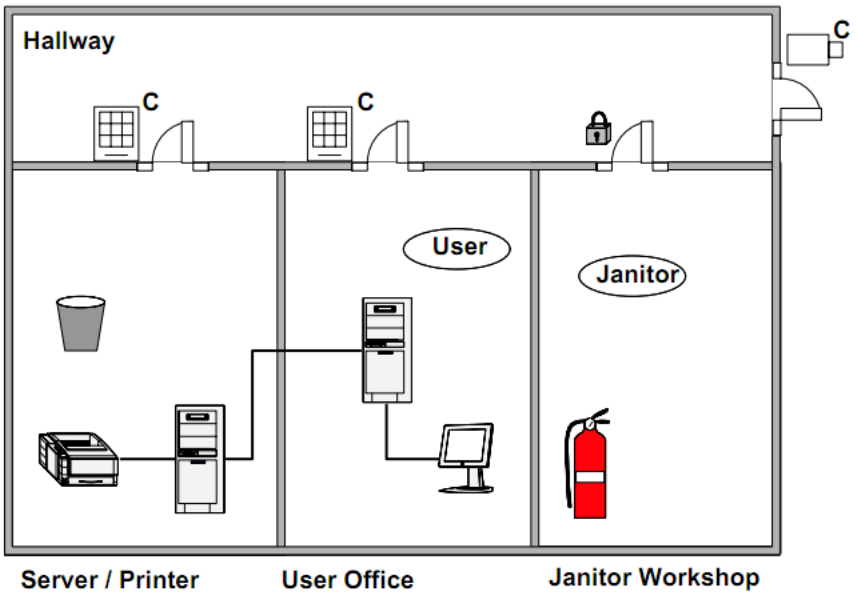


Figure 2.1: Example system connected with different locations (rooms) and virtually connected with data networks. Icons on doors specify access control mechanisms e.g. the entrance with face recognition and janitor room with tradition key-lock.

Now we will briefly explain the components that the system consists of:

2.2.2.1 Infrastructure

The infrastructure consists of a set of locations and connections. Locations can be modelled as a node in the graph. Also the locks at the doors can be seen as locations since one need to pass these locks to reach to the room. In general, where data can be located and where access can be restricted is modelled as a node.

There can be multiple connection paths between two locations. Also the connection can be directed or undirected as in case of real system. For instance in Figure 2.2 the connection between HALL and SRV is Hall $\rightarrow$ CL_{SRV} $\rightarrow$ SRV since to go to sever one need to pass the authentication at CL_{SRV} node which represents the lock of the server room. But there is a direct way SRV $\rightarrow$ HALL because one does not need any authentication to get out of the room if one is

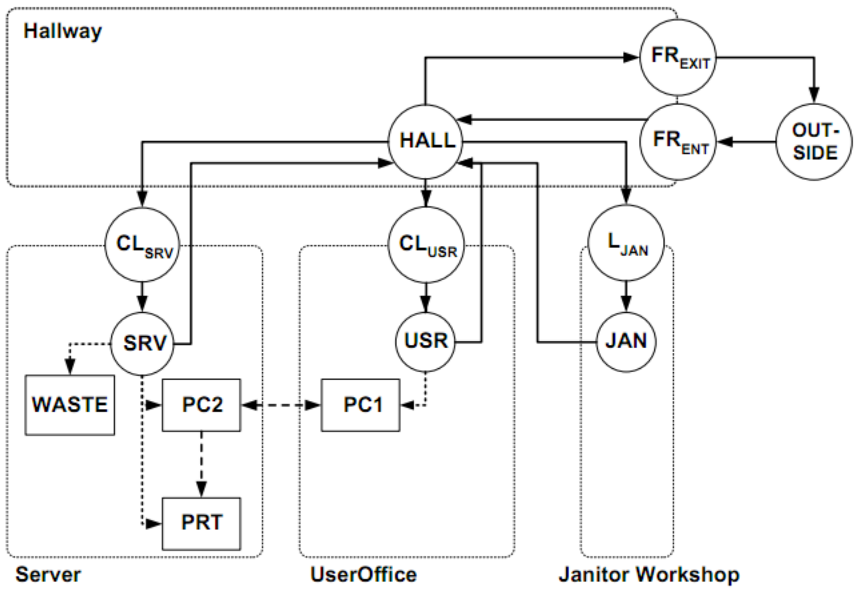


Figure 2.2: Abstraction for the example system from Figure 2.1 The different kinds of arrows indicate how connections can be accessed. The solid lines, e.g., are accessible by actors modelling persons, the dashed lines by processes executing on the network. The dotted lines are special in that they express possible actions of actors.

already inside it. Also the connection can be virtual as in case of data network. The dashed line between PC1 and PC2 in the figure, for example, depicts the virtual connection between two PCs. The connection flow is based on the connection model existing in the real system.

The node labelled "Outside" is outside the interest of our example model. Whatever lies in the *Outside* does not concern the modelling of the building system in the example. These nodes such as *Outside* nodes are collapsed nodes. Collapsed nodes allow to focus on specific area of system to be modelled. Later on one can replace the collapsed nodes with an extended system model to include previously ignored areas of the system.

Domains are used to group the locations. All the physical locations, for example, can be under domain "Locations" while all the data nodes such as PC1, PC2 and printer can be under domain "Data". The idea of grouping locations under a domain is to separate the interaction knowledge between different groups of locations, i.e., we may not want to have any connection between two domains in the system.

2.2.2.2 Actors

Actors are defined as the entities that can move from one location to another in the infrastructure. Actors can be people like users or janitors or can be processes such as data operations between two computers. Since locations have restriction policies, to access a location actor need some access rights. We call this *actor's capabilities*. An actor's capabilities can be itself (in case of face/print recognition) or it can be cipher keys/physical keys.

2.2.2.3 Data

Data are the objects actors work with. Data can be any set of information located at any location or possessed by any actor. For instance any information stored in computers PC1 and PC2 can be regarded as data. We can also associate certain knowledge gained by any actor as data. For instance, if an actor A knows the cipher key of one location then we can regard that knowledge as data and associate it with that actor.

2.2.2.4 Actions

Actors are supplied with a set of actions. Actions can be in/out, i.e., destructive read and output of data; move, i.e., migrate from one location to another; eval, i.e., spawn a process and read, i.e., non destructive read. These actions model the behaviour of the system.

So the system model can be defined as (Probst and Hansen [2009]):

Definition 2.3 A system model consists of all the components just introduced. Using locations, actors, data, and actions, it allows to capture the most important aspects of systems and insider threats who the user is, what the user does and knows, and where the user does it. While very simple in nature, this model is both powerful enough to model real-world scenarios, and at the same time flexible enough to be easily extendable.

2.2.3 System Extensions

Access control, encryption/decryption and logging are the extensions that we can add to the system model. These extensions are briefly mentioned below.

2.2.3.1 Access Control

To model the access control mechanism, a location has a set of access policies. These access policies guarantee that no unauthentic access will be possible for the given location. We call it *restrictions* of the location. Similarly, an actor is provided with a set of access grants which we call *capabilities.* For instance, a cipher key can be the capability of actor.

To get access to a location, restrictions on the location should match the capabilities of the actor. For instance, the boxes in Figure 2.3 represents the access control of that location. To go to the user room or the server room the restriction is C_U:m. Here C_U is the key and **m** is the move action. J:m and U:m at the entrance means actors J and U are needed themselves, in this case for face recognition system at the entrance.

2.2.3.2 Encryption/Decryption

The data encryption and decryption is done via keys. For e.g. in Figure 2.3 C_U:m in the box says key C_U is needed to access the location. We can think the cipher lock as a kind of data and the key can encrypt (lock) and decrypt (unlock) it.

In the similar manner to access control we can bind data with some key and only the matching key can decrypt the data. This encryption can be symmetric or asymmetric. Also a data can be encrypted with a set of keys. An empty set represents unencrypted data.

2.2.3.3 Logging

As can be seen from the boxes in Figure 2.3, some actions are over-barred, such as m or $\bar{m}$. The difference between these two is that the m represents an unlogged action whereas the later represents a logged action.

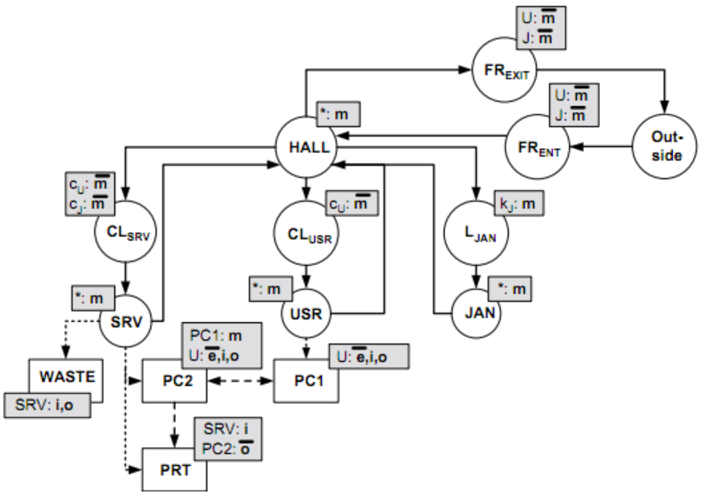


Figure 2.3: The abstracted example system from Figure 2.2, extended with policy annotations. There are two actors, janitor J and user U, who, e.g., have different access rights to the user office and the server room. Note the difference between accessing the user office or the server room with a cipher lock (logged) as opposed to the janitor workshop with a key (not logged).

The idea of logging is to provide log files for future edits. When an action is marked as logged then the logging component of the system will mark the reason and place of logging, i.e., who performed the action with what credentials (keys, biometrics, etc.) and where.

2.3 Modelling Language

In the previous section we define an analysable system model that represents the real world system. In doing so, we presented a description of system components like actors, data, actions as well as extensions like access control, logging and encryption/decryption. In this section we will present a modelling language that will be used to implement the desired analysis on the abstracted system model. The language is syntactically close to the abstract model specification.

In the following sections we will see how we specify our system components in the language.

2.3.1 Language Grammar

We are now going to define the grammar for our language. The grammar will define the syntax for our modelling language. The following subsections will focus on each part of our language grammar.

2.3.1.1 Spec

As we discussed earlier in section 2.2.2, a system or infrastructure is comprised mainly of 4 things: locations, connections, actors and data. Listing 2.1 shows our specification being defined as *spec* containing set of locations, connections, actors and data. These sets should be mentioned in the order defined by the *spec*.

```
spec :
 'locations' '{' ( ( domain | located | location ) ';' )* '}'
 'connections' '{' connection* '}'
 'actors'      '{' actor* '}'
 'data'        '{' data* '}'
 ;
```

Listing 2.1: Syntax for the system specification

2.3.1.2 Location and Policies

Location is one of the components of the system specification. A location is always under a domain name. A domain holds a set of locations and should have a unique name. Listing 2.2 shows that a location is given a name and a set of attributes and policies.

- *LOCID* is the name of the loation.
- Attributes can be optional. An attribute can be a domain name or a location name.
- A policy consists of a property and a corresponding set of actions. Property can be *NAMEID*, i.e., actor's name; *LOCID*, i.e., location name; *KEYID*, i.e., key name; and a wild card character "*". This wild card character signifies that any property is allowed. Actions can be either logged or unlogged. Available actions are move(m), eval(e), in(i), read(r), out(o) and the wild card (any action is allowed) character *.

```
location : LOCID [locattributes*]{policy*};

locattributes : domain | located  ;

domain  : 'domain' '=' DOMID;

located : 'location' '=' LOCID;

locationPolicies : policy* ;  //list of policies

policy : property ':' actions';'

property: NAMEID | LOCID | KEYID  | '*';

actions : unloggedAction | loggedAction ;

unloggedAction  : 'i' | 'o' | 'm' | 'r' | 'e' | '*';

loggedAction : 'log_' unloggedAction;
```

Listing 2.2: Syntax for the location and policies

2.3.1.3 Connection, Actors and Data

A connection is represented as *location name (source edge): set of other locations(destination edges).* The location names mentioned in the connection should be mentioned or defined in the *locations* component explained before.

Actors are expressed as actor name followed by the keys they possess. A known key is any key that actor has knowledge/remembrance of such as key codes and cipher codes. Owned key is the key that actor owns, for instance, physical key or biometric features.

Data are given a name, a set of policies just like the location policies defined above and an optional location name. A policy with wild cards, i.e., *:* means that data has no restriction policies and anyone can access it. Where as a policy like key_u:r tells that key_u is needed to read the data. Location name states the location where data can be found.

All domain, location, data, actor and key names should be unique in order for system specification to be well formed. For this we define these id's with the appropriate prefixes. A key name, for example, will always start with *KEY_* and a data name will always start with *DAT_*. This can be seen in Listing 2.3.

```
connection: LOCID ':' LOCID ( ',' LOCID )* ;

actor: NAMEID {(known_keys)* (owned_keys)* ;

data: DATAID {policy*}[location?];

NAMEID: 'ACT_' ID;

LOCID : 'LOC_' ID;

KEYID : 'KEY_' ID;

DOMID : 'DOM_' ID;

DATAID: 'DAT_' ID;
```

Listing 2.3: Syntax for the Connection Actors and Data

2.3.2 Language Example

In the previous section, we defined the syntax for the system specification. This section will present an example using the grammar mentioned above. Listing 2.4 will serve as the base example that we will consulting from here on. It represents the system shown in Figure 2.3

```
locations
{

// locations in the building A

   domain = DOM_LOC_BuildA; // domain name

   LOC_outside    { *: *; };
   LOC_reception  {*: *; };
   LOC_fhallway   { ACT_U: log_m; ACT_J: log_m; };
   LOC_hallway    { *: *; };
   LOC_csrv       { KEY_u: m;};
   LOC_srv        { *: *; };
   LOC_cusr       { KEY_u: m; };
   LOC_usr        { KEY_u: m; };
   LOC_ljan       { KEY_j: log_m;};
   LOC_jan        { *: *; };

// locations in the network
   domain = DOM_NET_BuildA;
   LOC_PCrec [location = LOC_reception] {*:*;};
   LOC_PCusr [location = LOC_usr] {*:*;};
   LOC_PCsrv [location = LOC_srv] {*:*;};
}
connections
{
  LOC_outside : LOC_reception, LOC_fhallway ;
  LOC_reception : LOC_outside, LOC_fhallway ;
  LOC_fhallway : LOC_hallway, LOC_reception, LOC_A ;
  LOC_hallway : LOC_csrv, LOC_cusr, LOC_ljan, LOC_fhallway ;
  LOC_csrv : LOC_srv;
  LOC_cusr : LOC_usr;
  LOC_ljan : LOC_jan;
  LOC_srv : LOC_hallway, LOC_PCsrv;
  LOC_usr : LOC_hallway, LOC_PCusr;
  LOC_jan : LOC_hallway;

  // connection for dom_network
  LOC_PCrec : LOC_PCusr, LOC_PCsrv;
  LOC_PCusr : LOC_PCrec, LOC_PCsrv;
  LOC_PCsrv : LOC_PCrec, LOC_PCusr;

}
actors
{
  ACT_U
  {
    known_keys = {KEY_u};
  };
  ACT_J
  {
    owned_keys = {KEY_j};
  };
}
data
{
  DAT_rec{KEY_u: r; KEY_j: r;}[LOC_PCrec];
  DAT_srv{KEY_u: r;}[LOC_PCsrv];
  DAT_usr{KEY_u: r;}[LOC_PCusr];
}

key{
  KEY_u [LOC_jan];
  KEY_j ;
}
```

Listing 2.4: Modelling Language Example exhibiting the implementation of grammar

locations is a set of locations. Each set of locations are under a domain name. In the example provided there are two domains: *DOM_LOC_BuildA* (contains the physical locations) and *DOM_NET_BuildA*(contains the data locations).

Locations are provided with policies. For instance: *LOC_fhallway* { *ACT_U: log_m; ACT_J: log_m;* } will read as location named *LOC_fhallway* has the restriction policy where ACT_U and ACT_J are allowed to move(m) and the action move is logged.

connections is a set of connections. Left hand side of ":" is the source locations and right hand of ":" is the set of destination locations. For example, *LOC_srv : LOC_hallway, LOC_PCsrv;* specifies that there is a path from *LOC_srv* to *LOC_hallway* and *LOC_PCsrv*. Note that this connection is directed, i.e., *LOC_A: LOC_B* does not mean that there is a connection from *LOC_B* to *LOC_A*.

Actors are defined with their actor name and set of keys they possess. In the example, actor U (actor user) has a known key KEY_u whereas actor J (actor janitor) has an owned key KEY_j.

These keys are then defined to specify whether they are stored in any location or not. For example *KEY_u[LOC_jan]* states that KEY_u can be found at the location *LOC_jan*.

Data are given with their data ids, set of policies and location where the data is located. Example *DAT_rec*{*KEY_u: r; KEY_j: r;*}*[LOC_PCrec]* reads as data with data id *DAT_rec* can be found at location *LOC_PCrec* and can be read by *KEY_u* and *KEY_j*.

2.4 Analysing Models

Previously we defined a language that provides the syntax for the specification of our system model. By now we have build an organized way to define our system and its components. In this section, we will focus on the analysis aspect of our system that is to say how we can simulate real world behaviours in our system model.

2.4.1 Static Analysis

One of the general approaches of counteracting system attacks is to keep the analysis of previous attacks. If an action is taken then it is searched in the previous attacks collection and if matched it is recognized as attack. This seems helpful when the attacks pattern does not change too often.

On the other hand, Static Analysis (Nielson et al. [1999]) can deal with such dynamic behaviours as it tries to identify the system properties that holds for every single configuration. All possible states can be calculated from an initial system configuration. With the help of static analysis we can simulate the desired behaviour of the system before its actual implementation. This gives the benefit of identifying vulnerable points in the system and take necessary security measures to solve the issue even before the system is implemented. Existing approaches of security measures can benefit when paired with static analysis techniques (Probst and Hansen [2008]).

2.4.1.1 Log Equivalent Actions

There is a need to address the locations and actions that are indistinguishable from the viewpoint of an observer or analysis in our case. This means that if an analysis shows that an actor can be in a location l then he might just as well be in any equivalent location or might have performed any actions in between. The idea to use log equivalent is to speed up the reachability analysis as an actor can be in any equivalent locations so it becomes easier to compute the transitive, reflexive hull of the current location with an assumption that actor can be in any of these locations or might have performed any actions in between them. Also, it is helpful in performing LTRA (as will be discussed in section 2.4.1.3). In LTRA, two actions or locations are equivalent if moving from one location to another or performing the action actor does not cause any log entry. It helps to find out what might have happened between the two logged events. Algorithm 1 shows the pseudo code for determining log-equivalent. For each actor all the locations are checked where actor can be located and then it is checked whether the actor can perform any action on locations. In case of LTRA, only actions that does not cause log entry are considered. The algorithm stops when no further changes occur.

2.4.1.2 Reachability Analysis

The question that first arises while trying to model the behaviour of the system would be : *can actor A go from location X to location Y?*. The reachability of one destination location from another source location in our system depends on the matching of restrictions of location nodes in path between X to Y and capabilities of actor. Simply to say if A has required credentials needed at all the nodes in the route of X to Y then A can go from X to Y, else not. Algorithm 2 shows the pseudo code to find whether an actor A can move from location X to Y

Algorithm 1 Algrorithm to simulate log equivalent actions

```
equivalent()
$changed$ = true
while $changed$ do
    for all actors $n$ do
        for all locations $l$ that n might be located at do
            for all locations $l'$ reachable from $l$ in one step do
                simulate all actions that $n$ can perform on $l'$ (without causing a log entry in case of LTRA)
                for each action set $changed$ if $n$ at location $l$ learns a new data item
            end for
        end for
    end for
end while
```

Algorithm 2 Algrorithm to find if actor A can go from Location X to Y

```
for all available sets of routes $R*$ from location X to Y do
    for all location node $L$ in a single route $R$ do
        find restrictions on $L$
        find capabilities of actor $A$
        if $A$ capabilities bypasses restrictions on $L$ then
            proceed to next location node in $R$
        else
            cannot go any further in this route $R$
        end if
    end for
end for
return success status
```

Algorithm 2 returns a boolean status stating the success or failure of move action of actor A from location X to Y. Algorithm 3, which is our modification of Algorithm 2, is used to find the reason that does not allow actor to perform action on one particular location node. We then try to resolve the reason so that actor can perform its desired action on that particular node. For example, at node N actor A needs key K then algorithm tries to find how to get key k so that A can pass N.

In Algorithm 3 from line 12-25, it is shown that how an insider attacker would look for gaining unauthorized access. In line 12 a reason can be actor (in case of biometrics such as face recognition) so the attacker needs to social engineer that particular actor. Line 13 states a reason can be key also. So the attacker

Algorithm 3 Algrorithm to generate attacks when actor A traverses from Location X to Y

```
1:  for all available sets of routes $R*$ from location X to Y do
2:      for all location node $L$ in a single route $R$ do
3:          find restrictions on $L$
4:          find capabilities of actor $A$
5:          if $A$ capabilities bypasses restrictions on $L$ then
6:              proceed to next location node in $R$
7:              if the reason is logged then
8:                  log the reason,actor,location in log sequence
9:              end if
10:         else
11:             cannot go any further in this route $R$
12:             if reason is $ACTOR$ then
13:                 social engineer the $ACTOR$
14:             else if reason is $KEY$ then
15:                 if $KEY$ associated with Location then
16:                     find the location at which $KEY$ is located
17:                     repeat the process from line 1 but now X will be current node $L$
                        and Y will be location node of key
18:                     if return is success then
19:                         actor got the key
20:                     else
21:                         actor could not get the key
22:                     end if
23:                 else if $KEY$ associated with $ACTOR$ then
24:                     social engineer $ACTOR$ to get the key
25:                 end if
26:             end if
27:         end if
28:     end for
29: end for
30: return success status
```

needs to find if the key is located somewhere or is in possession of some other actor. If the key is with any actor then attacker needs to social engineer the actor again. But if the key is in any location then attacker tries to traverse that key's location from the current node in order to obtain the key. The algorithm is in the worst case scenario since we presume an attacker has all the knowledge of keys location, actors and their keys. This may lead to over approximation of number of attacks but it will always contain the subset of actual attacks.

2.4.1.3 Log Trace Reachability Analysis (LTRA)

As we have discussed earlier one of the common and frequently used methods in cyber crime investigation is tracing log files to find who caused the event. There are times when what happened between two logged events becomes more important question than the logged events itself. *LTRA* tries to answer this question. It takes a system model specification described in 2.4 and a log file as input. *LTRA* then tries to simulate all the actions for the actors present in the system such that the logged events can be generated. The result of interest are all the set of actors, actions and possible paths that caused the logged events.

Algorithm 4 describes the LTRA. It takes a log as input. At first initialization is done i.e. all actors are placed at their initial location (outside the system in our case), set with initial key sets (may be empty) and all locations are initialized with potentially empty initial data set. Each log entry from the log sequence is taken one by one. In each iteration, the algorithm first calls log equivalent method from the current location. Then it checks whether there is exactly one actor that can cause the entry and update the data structure accordingly. The algorithm repeats until all log equivalent actions are simulated.

2.5 Summary

In this chapter, we discussed about the insider problem in the real world system and the way to deal with it. For this we reviewed the theory of conversion of real world system into an abstracted system. We looked into the details of abstracted system components i.e. locations, actors, data, keys. Similarly, we also looked into modelling grammar language and how to use it to represent the abstracted system components. We also reviewed the analysis algorithms like reachability analysis and LTRA to show how these analyses helps in counter measuring the insider problem.

Algorithm 4 Algrorithm for LTRA

```
Require: system specification and log sequence
 1: /* initialization */
 2: place all actors at their initial location
 3: initialize all actors and locations with initial key set
 4: /* iterate over log sequence */
 5: while log sequence not empty do
 6:    {/}* perform log equivalent actions */
 7:    equivalent()
 8:    /* take next logged action */
 9:    next(reason, from, to, action) from log
10:    if reason is an actor then
11:       from is the exact location of the actor reason
12:       remove that actor from all other locations
13:       the only possible actor is reason
14:    else if reason is a key then
15:       possible actors are all actors who might be at from and know the key
          reason
16:       if only one actor at from knows the key reason then
17:          remove that actor from all other locations
18:       end if
19:    else if reason is a location then
20:       potential actors are all actors who might be at from
21:       if only one actor is located at from then
22:          remove that actor from all other locations
23:       end if
24:    end if
25:    for all potential actors n do
26:       simulate effect of $n$ performing action action
27:    end for
28: end while
29: /* perform log equivalent actions */
30: equivalent()
```

Chapter 3

Analysis and Design

In this chapter, we will cover analysis and design patterns to deal with the insider problem discussed in previous chapter. The aim of this chapter is to provide analysis and design of a framework that we based on the **EXASYM**(Extensible Analysable System Model) theory by Probst and Hansen [2008]. The tool is designed to generate attacks at specified points or locations in the system so that one can find out the vulnerabilities in the system before hand.

3.1 Framework

The task we are interested in is to be able to present a real world system into an abstracted analysable model on which analyses algorithms can be run. When we get a real world system, like in Figure 2.1, we want to be able to convert this real world scenario in some abstracted form. For this purpose we defined infrastructure and components like actors, data, connections in Chapter 2. This abstracted system can then be fed to insider analyses to get the threats result.

Figure 3.1 shows our basic understanding of building such a framework. First of all a real world system is mapped to an abstracted system. This abstracted system is analysable. The mapping from a real world system to an abstracted system is done with the help of modelling language we described earlier in section 2.3. Once we get the abstracted and the analysable system then we can run the analyses algorithms on the abstracted system. The result we receive will be the list of possible attacks.

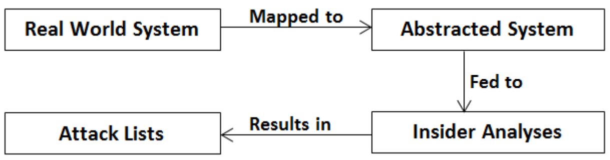


Figure 3.1: Steps showing basic steps to create our attack generation tool

The tool will be described in more detailed in the next chapter. This chapter will however establish an example case and will discuss what we are trying to achieve from the tool and how. Figure 2.1 will serve as the base example for the explanation here.

3.2 Real World System to Abstract System

The abstract system is the analysable system discussed in section 2.2.2. The abstract system can be viewed as a decomposed view of the real world system where every decomposed part is unique and tagged with its appropriate role. To suffice that, we define our system and while doing so we present the details of all the components of the system in the specification. This specification is the language mentioned in section 2.3. Our whole system of interest is specified in this modelling language which in turn is represented in EBNF (Extended Backus Naur Form). This language is then parsed to create a parse tree where our system components are defined by the nodes of parse tree.

Figure 3.2 shows a portion of the parse tree. In the figure, we can see how a location is broken down to its basic components , i.e., property and actions. When the specification of a system is given then the whole system is converted into parse tree like shown in the diagram 3.2. In section 2.3.1, we presented EBNF notation for the specification language. By the rule of grammar mentioned there, we define our system components and the parse tree will then give the *syntactic structure* of our specification model where each node is representing a system component. In other words, we can see the nodes as collapsible. For instance, the location node is collapsed and so we only know that the mentioned thing is location. Now if we expand the location then we can see there is name of location and location policies where the location policies node is

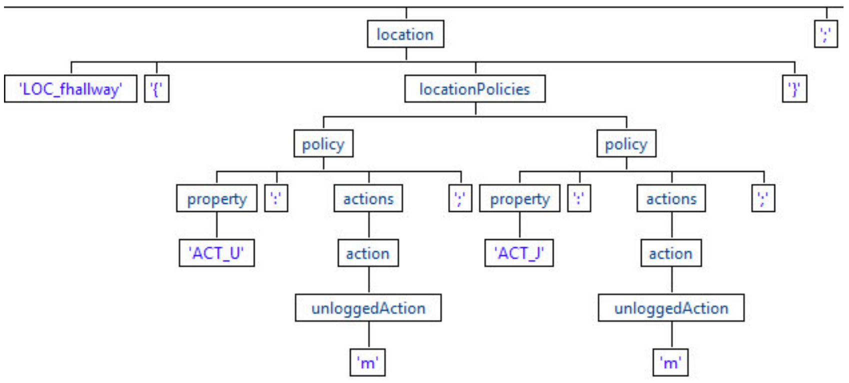


Figure 3.2: A portion of parse tree showing how a location is decomposed into its basic components.

again collapsible. This helps to visualize or interpret the system starting from a broader view to the more narrowed scene.

For another instance, Figure 3.3 and Figure 3.4 show the portions of parse tree where connections and actors are mentioned respectively.

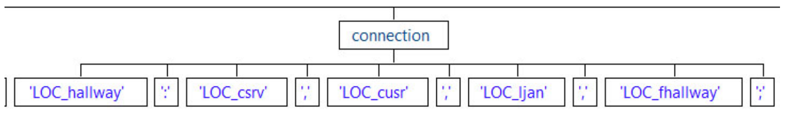


Figure 3.3: A portion of parse tree showing how connection is broken down to its basic components.

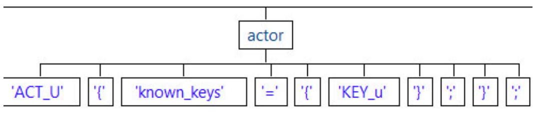


Figure 3.4: A portion of parse tree showing how each actor is divided into its basic components.

3.3 System Model as Graph

Once we get the system components as nodes of the parse tree after the mapping, we want to do something meaningful with these nodes. That is to say if we see a location node then we want to store those locations and similar kind of actions for other components.

As said earlier in section 2.2.2.1, we can view the locations and locks on the locations as nodes of a graph. As in a graph one node is connected to a set of nodes (directed or undirected edge), a location in the real world system such as Figure 2.1 is also connected to a set of other locations. Movement of an actor from one location to another in the system model can be simply viewed as traversing the graph from one node to another node.

Therefore it seems natural to model the system with the help of graph. Once the system is modelled as a graph then the sets of nodes (locations in our case) and sets of connections between nodes are always available for us to perform required analysis which simplifies computations. Figure 3.5 shows the graph representation of the system specified in the language example from section 2.4 . This graph based representation of the abstracted system model is easier to comprehend within a certain range, however, with the complexity of system increasing the graph tends to increase also.

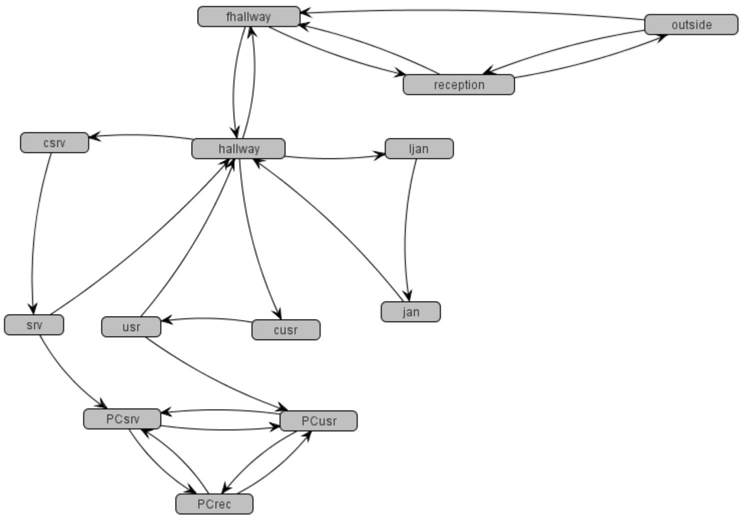


Figure 3.5: Graph representation of the example system model discussed in section 2.4

Hence we propose to convert the system in a graph and henceforth the analysis we perform will be graph based.

3.4 Attack Trees

Attacks trees were first described by (Schneier [1999]). These trees are used to identify and investigate attacks in a systematic and organized way. Basically, it represents an attack in a tree structure. The root node of the tree is the main goal of the attacker and the children of the nodes are the ways or attacks to achieve the goal mentioned by their parent node. In a complex system, there may be more than one root node. The branches can be either an AND-branch or an OR-branch. And AND-branch tells that all the branches must be followed to reach the root whereas in an OR-branch one of the branch is sufficient to reach the root. The idea is to decompose the main goal into possible detailed basic subtasks so that it covers insights of all the grounds that can lead to the fulfilment of the main goal. Figure 3.6 shows a basic example of attack tree taken from Schneier [1999]. Here for example, to open the safe, the attacker can do any one of the four tasks: pick lock; learn combo (safe code); cut the safe open; or install the safe improperly. To eavesdrop successfully, attacker has to perform both of the tasks.

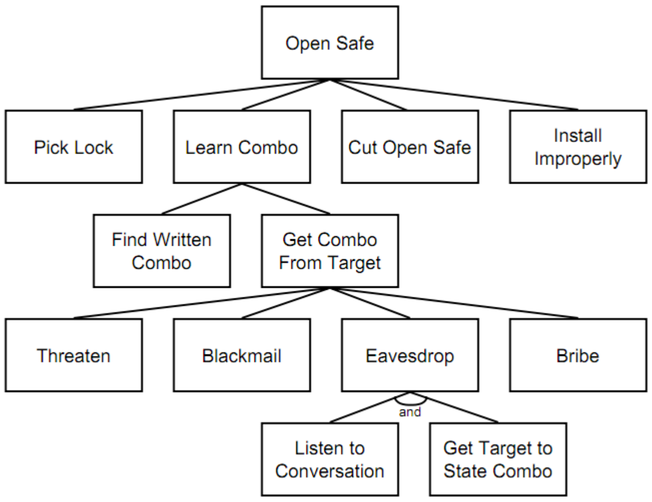


Figure 3.6: Basic example of attack tree. All the nodes except the one marked with *AND* are OR nodes.

Attack trees can be modified to have various annotations attached to each nodes. For example a node can have annotations such as probability of being completed or costs to complete the attack mentioned in the node. Annotations help to describe the node and its attributes which in turn helps us to categorize and rank the attacks. For example, later on one can filter the set of attacks which costs within some range or the set of attacks whose probability of being successful is higher than some percentage.

We can present the attacks generated in the system in the form of an attack tree. Algorithm 5 shows the creation of attack tree.

Algorithm 5 Algrorithm to generate attack trees when actor A traverses from Location X to Y

```
1: for all available sets of routes $R*$ from location X to Y do
2:    for all all nodes $N*$ in $R$ do
3:       $N$ becomes node in attack tree $T$
4:       if $N$ is restricted then
5:          find reason $R$ and append it to node $N$
6:          if finding $R$ means traversing another location set then
7:             Repeat the process from above the new attack tree will be sub
               attack tree of $N$
8:          end if
9:       end if
10:   end for
11: end for
```

Lets elaborate the algorithm with an example. Actor **U** is traversing from *X* to *Y*. **A**, **B** and **C** are the nodes in between. So, the main attack tree, say, **T** has nodes *X, A, B, C* and *Y*. But at location *B*, actor *U* does not have privilege to access. The reason for this is, say, *key K* which is located at some other location **Z** in the system. This means to get the *key K*, *U* will travel from current position to *Z*. Since it was not allowed at **B**, the previous position would be *A*. Traversal from A to Z is successful to assume with $\mathbf{A} \rightarrow \mathbf{E} \rightarrow \mathbf{Z}$. So now the new tree, say, **T'** with nodes *A, E and Z* is the sub tree of node B. Lets assume another reason, why **U** could not pass **B**, was **actor J**. So, it means you must social engineer the actor. The reason is added as another tree to B where there is only one node explaining to social engineer the actor. Notice that node B is an *OR node* since achieving any of the two reasons i.e. finding Key U or social engineering the actor J will let actor U pass the node B. This can be seen from the Figure 3.7

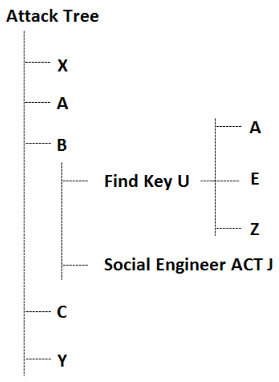


Figure 3.7: Example showing the use of Attack Tree.

3.5 Analyses

Once we have transformed a real world system into an abstract model, this model is now ready to be analysed. Also in the mean time, we converted our abstracted model to a graph based model where each node is a location and each edge is connection. Now the system needs analysis algorithms to perform some analysis on it. What we want to answer mainly is:

- Can an actor or set of actor move from one location to another location , and
- Can an actor or set of actor extract data from a location ?

3.5.1 Extract Credentials

Both of the tasks mentioned above need checking of credentials of the actor at each location point in the path between specified start and end locations. Algorithm 6 shows the extracting mechanism of the credentials. At all the nodes in a route R, the access control list of the node is read. Access control is specified as {property:actions}. While traversing from one location to another the action is *move*. Therefore, we check if the actions available in access control of location L is *move*. In this case, the property, which can be an actor or key, is added as credentials needed at L.

Algorithm 6 Algrorithm to extract credentials at all location nodes in different routes between two location end points

```
1: for all available sets of routes $R*$ from location X to Y do
2:    for all location node $L$ in a single route $R$ do
3:       credentials_$L = \phi$
4:       retrieve access list $ACL$ at $L$
5:       for all {property,actions} in $ACL$ do
6:          for all actions $a$ in actions do
7:             if $a$ is move then
8:                if property is an actor or key then
9:                   credentials_$L$ = credentials_$L \cup$ {property}
10:               end if
11:            end if
12:         end for
13:      end for
14:      return credentials_$L$
15:   end for
16: end for
```

3.5.2 Capability-Restriction Test

When there is no actor present while finding paths between two locations then extract credentials is just as same as stating the policies of location directly, i.e., no capability-restriction match. This is just to show the user what kind of credentials are needed at each location node.

If an actor or set of actors is present then a set of capability is formed which comprises of capabilities of all the actors in the set combined. So, while traversing from one location node to another the algorithm checks if the intersection of reason(restriction reason) at a node and the capability set of actor or actor set is ϕ or not.

For instance to go to node N restriction reason R is KEY_u and KEY_j and capability of actor is KEY_j. Since, {KEY_u, KEY_j} $\cap$ {KEY_j} = {KEY_j} the actor has passed the capability restriction test. However, lets say actor has KEY_m. But, {KEY_u, KEY_j} $\cap$ {KEY_m} = ϕ so the actor could not get to pass the capability-restriction match. Algorithm 7 shows the capability-restriction test mechanism.

Algorithm 7 Algrorithm for capability-restriction match

```
1: for all available sets of routes $R*$ from location X to Y do
2:   for all location node $L$ in a single route $R$ do
3:     Access all the policies $P*$ of the location $L$
4:     credentials_$L = \phi$
5:     for all policies $P*$ do
6:       extract reason/property $R$ of policy $P$
7:       extract capabilities $C$ of actor or actor set
8:       if $R \cap C \neq \phi$ then
9:         credentials_$L$ = credentials_$L \cup \{\text{property}\}$
10:      else
11:        credentials_$L = \phi$
12:      end if
13:    end for
14:    return credentials_$L$
15:    if credentials_$L \neq \phi$ then
16:      restriction-capability test = pass
17:    else
18:      restriction-capability test = fail
19:    end if
20:  end for
21: end for
```

3.6 Generation of Sub-Attacks

A sub-attack, in our scenario, takes place when the capabilities-restriction is not matched. In such a case, a further analyses is performed in order to show what could have been done in order to pass that very unmatched capabilities-restriction test.

For instance, if *U* needs *KEY_j* to go to the janitor room. When travelling from one location to another *U* has to pass the janitor room but he cannot. So now, another level of attack list is being generated which describes *U* how and where to seek the *KEY_j* to open the janitor room. If an attack reason is a key then the sub-attack generation will try to locate the key or if its an actor then it will suggest to social engineer the actor. The reason for finding of sub-attack is to present a detailed level of attack generation in the system. With the stack of restriction-capability test going unmatched, the size and level of these sub-attacks can increase. So, the sub-attack trees with large and complex system can grow into many level of nesting.

In algorithm 3, the goal of line 10-15 is to generate sub-attack levels for the missing credentials. The generation of sub-attacks may go to deeper nested levels. Also, there may occur a loop. For instance, to go to location X one needs to get KEY_k which is located at Y. While finding KEY_k one needs KEY_j in another location. And while trying to find KEY_j one again needs to find KEY_k. The avoidance of this cycle is necessary in order to obtain stable attack trees. To avoid any cycle every keys and actors that have been analysed are noticed and stored in data structure so that when they appear next as the reason for sub-attack they are simply discarded. That is to say if a key or actor is analysed before as the reason for the sub attack then when they appear again as the reason we just mention that they have been previously explored.

3.7 Design Principals

This section will reveal the design principles. The tool is being developed with the goal of future development. We want to design the tool in such a way that the tool is robust, visually appealing, great user interaction and supports future extensions.

Visualization: The tool should be visually appealing and yet simple to use. The representation of graph should be easily comprehensible. We should keep in mind that the tool should represent the graph in simple and elegant manner and let user have choices to perform analysis. Since our tool displays graph, we should have some frame to display this graph. The graph should be easy to zoom and transform so that user can view the graph properly. Also, we have different system components like actors, data and locations. We will give a setting pane where user can set these values according to their choices. Similarly, the representation of attacks should be tree based. We can use some tree data structure to represent these threats in attack tree form.

User Interaction: As much as tool should be visually appealing, it should give the user freedom to operate on it. The tool should be assisting the user to maximize her experience with the tool. For this, we have thought of implementing visual assists to the user. For example, if a user wants to know the policy of any node in graph then she just have to hover over the node and the policy will be displayed as tool tip. Similarly, we can use various colors to represent paths in route, paths an actor can or cannot travel, node an actor can or cannot pass, node an actor picks etc.

Room for future extensions: There is room for further development of every tool and ours is in early stage of development. This makes us to structure the program in such a way that in the future our tool can be easily extended with new extensions and enhancements. For this reason, we have decided to divide the program structure into its basic components. Each components are designated with its own set of tasks. For example, lets say **Visualization** is a module in the tool that will deal specifically only with the design concerns of the tool. This way, the tool can easily be extended in the future without much trouble.

3.8 Extraction of Data

We would also like to see if a user can access some data or not. Data is generally located at some location. So acquiring a data means traversing from initial location to the location where data is located. The traversing analyses algorithms are presented before. But the successful traversal to data location does not mean that actor can access the data. As we know, data has its own policy. So in this case, when the actor has reached to the location where data is located then the capability-restriction test is done once again but this time the restriction is of the data and capability is same of the actor. If this test is successful then we can assume user gained access to data.

3.9 Summary

In this chapter, we presented discussions on how to proceed to develop a tool to generate attacks from system model. While doing so, we discussed briefly about our framework that we developed to deal with the insider problem. We provided with the details of the steps taken to perform analysis on a real world system. We also visited the theory of presenting our system model as graph for its convenience and flexibility. Attack trees were presented and the way to represent our set of attacks in the attack tree form was shown. We also discussed the basic design principals of the tool such as visualization and presented analysis algorithms.

Chapter 4

Implementation

This chapter covers the implementation details of the project. The developed tool is based on the theory **EXASYM** abbreviated for *Extensible Analysable System Model* by Probst and Hansen [2008]. The tool is used for generating attacks at specified locations in the system model graph. The tool can analyse who can access what and where in the system.

The tool is developed in the Java programming language. The choice of the language was based on the free availability of tools written in Java such as **ANTLR** (Another Tool For Language Recognition) and **JUNG**(Java Universal Network Graph) as well as familiarity with the Java language itself. We use **ANTLR** for the creation of grammar shown in section 2.3.1 and **JUNG** for the generation of graph as shown in Figure 3.5.

4.1 Overall Implementation Design

In this section we will describe the overall design of the tool. Figure 4.1 shows the work-flow of our implementation.

The work flow shows the steps of our implementation. A language grammar that specifies the system model (listing 2.4) is prepared and fed to the tool as input.

ANTLR parses the grammar and prepares an appropriate abstract syntax tree of the specification. Also, meantime the tool is integrated with **JUNG** where we prepare the graph from the specification. The graph, for example we receive for listing 2.4, can be seen in Figure 3.5. Until this stage, we were preparing our system model to be ready for the analysis. Since now the system is ready for performing analysis on it, the users of the tool are provided with some analysis actions. The two actions available are finding path between two locations and extraction of the data.

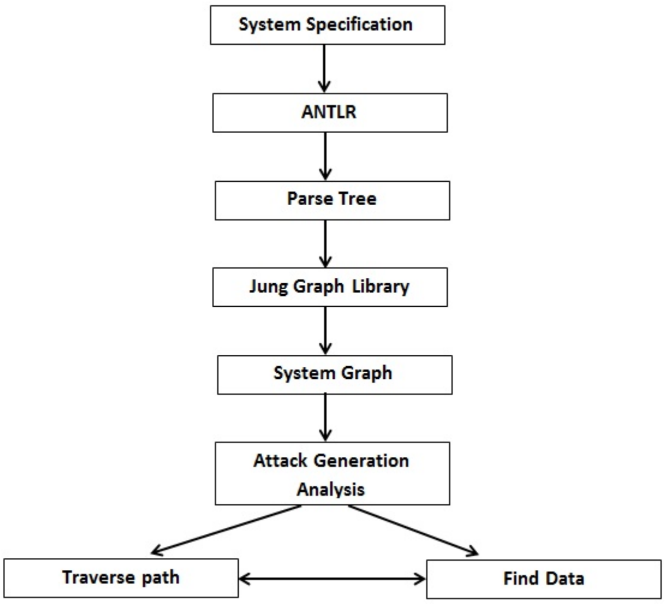


Figure 4.1: Implementation Design of the tool

4.1.1 Finding Path

One of the tasks that the user can perform on the abstracted graph is to find the path between two locations. The task of finding a path can be in the presence of an actor or a set of actors or not in the presence of any. This means that if a user does not specify any actor then the analysis will report all the reasons required at all the nodes in the path. The analysis, in this case, does not perform any restriction-capabilities test. Moreover, the user can choose to find all the paths that exist between two location end points or the existing shortest path between them. To find the shortest path, we have used Dijkstra shortest path algorithm from JUNG library. In JUNG there is no available function for calculating all paths between two nodes at the moment. Algortihm 8 shows the pseudo code that we implemented to find all the available paths between two location points. The pseudo code takes a start and an end location as input. Similarly, we pass an empty set to track visited nodes in graph and a linked list to supervise the current path being examined. The function is recursive and will terminate if there are no more outgoing edges in graph that leads to the specified end node. *Graph* is the final output which contains all the routes between start location *START* and end location *END*.

If the user mentions any presence of an actor or a set of actors then access control check is performed at every nodes in the path between two endpoints. If the access control is denied then the user is provide with information about how to access that particular node as mentioned in algorithm 3. If a set of actors is chosen instead of a single actor then the capabilities are checked from both the actors and if any one of them can bypass the restriction at location node then it is noted as success.

4.1.2 Extracting Data

Another task is to extract data. A user can try to simulate whether the data is accessible from a starting location by an actor (set of actors). In this case destination location would be the location where the data is stored. Then operation *finding path* mentioned before is called to find the routes between two location. And at last the restriction-capability match is performed where restriction is of the data and capability is of the actor.

4.2 External Libraries

In this section, we will give a short overview of the two external libraries that we are integrating with our project.

4.2.1 ANTLR

ANother Tool for Language Recognition (ANTLR) is a parser generator that uses LL(*) parsing. ANTLR takes a grammar, that defines the language (example, the one shown in Listin 2.4), as input and outputs a recognizer for the language.

Algorithm 8 Pseudo code to find all paths between two end locations in the graph

```
  /* At 1st iteration Graph is empty graph with just start and end location set to START and END */
Input: $START$, $END$, $VisitedLocation\_Set = \phi$, $Graph$, $CurrentPath\_LinkedList = \phi$
Output: $GRAPH$
  if $VisitedLocation\_Set$ has $START$ then
    return
  end if
  add $START$ in $VisitedLocation\_Set$
  add $START$ as last element in $CurrentPath\_LinkedList$
  if $START$ is $END$ then
    $predecessor$ = null
    $first$ = true;
    for all locations $l$ in $CurrentPath\_LinkedList$ do
      if $first$ then
        add $l$ in $Graph$
        $first$ = false
      end if
      if $predecessor$ = null then
        add $predecessor$ in $Graph$
        make connection from $predecessor$ to $l$
      end if
      set $predecessor = l$
    end for
    remove last element from $CurrentPath\_LinkedList$
    remove START from $VisitedLocation_set$
    return
  end if
  for all outgoing nodes $N$ from $START$ do
    callback with $START = N$ and other parameters $END$, $VisitedLocation\_Set$, $Graph$, $CurrentPath\_LinkedList$ as it is
  end for
  remove $START$ from $VisitedLocation\_Set$
  remove last element from $CurrentPath\_LinkedList$
```

A language is specified using Context Free Grammar and Extended Backus Naur Form. To do something meaningful with the language, actions can be specified in the grammar. Actions are written in the same language in which the recognizer is being generated in our case Java.

The role of a parser such as **ANTLR** is to parse where parsing refers to the process of converting a well defined input into an internal representation that can be represented by an *Abstract Syntax Tree (AST)* using the grammar rules. An AST represents abstract syntactic view of the input program. Every node of the AST represent the construct occurring in the input program i.e listing 2.4 in our case. We perform parsing in two steps (shown in Figure 4.2) -

- Step 1 refers to Lexical Analysis where the input program is parsed into the internal representation by breaking the program into a sequence of tokens. It is carried out in the lexer unit of the parser. We call it tokenizing the program.
- Step 2 refers to Syntactic Analysis where the internal representation is constructed based on the grammar rules of the parser. A parser takes a stream of tokens generated by the lexer as input and tries to map them to a set of rules where the end result maps the token streams to the AST.

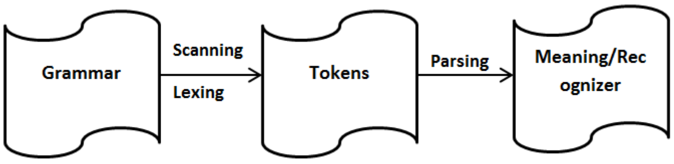


Figure 4.2: Parsing a Grammar into meaning

4.2.2 JUNG

As mentioned in 3.3, we have to model our system specification in a graph system where each nodes in the graph represents location and each edge represents connection between the nodes. In order to achieve the graph representation of the system, we have used an existing graph framework, namely *JUNG (Java Universal Network Graph)*. **JUNG** is a free, open source graph modelling and visualization framework written in Java. **Jung** provides common and extendible language for the analysis and visualization of data that can be represented as graphFisher et al. [2005]. The idea behind using JUNG stems from the following features it possesses:

- Open source,
- Implemented in Java,
- A number of built in visualization layouts,
- Can create your own custom layouts,
- Supports various forms of graphs e.g. directed-undirected graph, multi modal graph, graph with parallel edges, hypergraphs,
- Annotation feature for nodes, edges, entities and relation with meta data,
- Algorithms from graph theory, data mining, and social network analysis such as clustering, decomposition, optimization, network flows and distance calculation, statistical analysis and measures(PageRank, HITS etc.), and
- Filtering mechanisms to allow user to focus on specific portion of graph.

4.3 User Interface

The user interface of the developed tool consists of two frames. One frame is dedicated for the visualization of the graph. Second frame is a tabbed pane consists of three tabs namely *Settings*, *Sets*, and *Attacks*. Figure 4.3 shows the user interface of the tool.

The **Settings** tab contains general settings. Mouse mode is the mode to operate on graph. The start and end location are specified in this tab. If mouse mode is in *transforming mode* then we can zoom, rotate, skew the graph and in *picking mode* we can pick the nodes and connections in the graph. We can define whether we want to see all the paths between two endpoints or the shortest.

Figure 4.4 shows two graph views one for all paths and one for the shortest path.

The **Sets** tab contains two combo boxes that allows users to select the desired actor, data, or combination of both. When no option is chosen then the paths are calculated between two location endpoints without any access control check. This operation lists all the reasons required at all the nodes in the path. If the

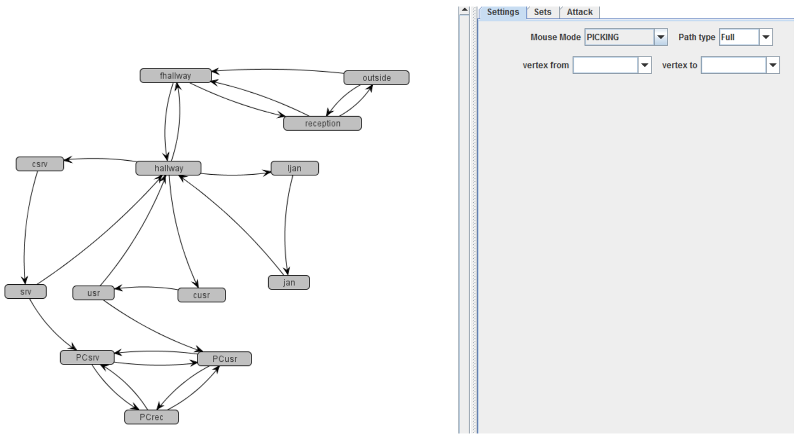


Figure 4.3: User Interface of the tool

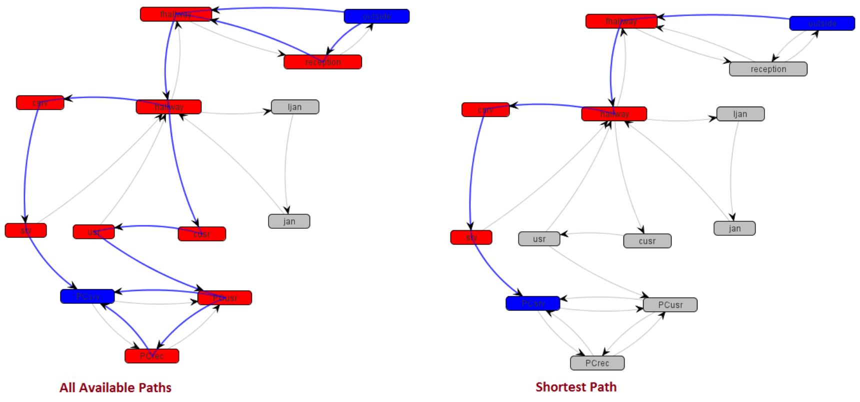


Figure 4.4: All paths and shortest path between LOC_outside and LOC_PCsrv

user selects one or multiple actors then access control check is done in the path where capabilities are those of the selected actor/s. Data box is used if user wants to check the attack on data or whether data can be extracted or not. When a data item is selected from the data combo box, the destination box in **Settings tab** is set to the location where data is located. While using the data set if no actors are selected then again no access control checks are done and all reasons at the nodes in the path are listed. Figure 4.5 shows the **Sets tab**.

The **Attack** tab is used to list all the attacks for the configuration made in *Settings* and *Sets* tabs. Depending on the values inserted in the fields present

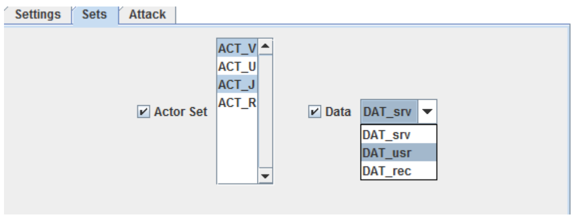


Figure 4.5: *Sets Tab* provides users with option to select actors and data on which they want to perform analysis

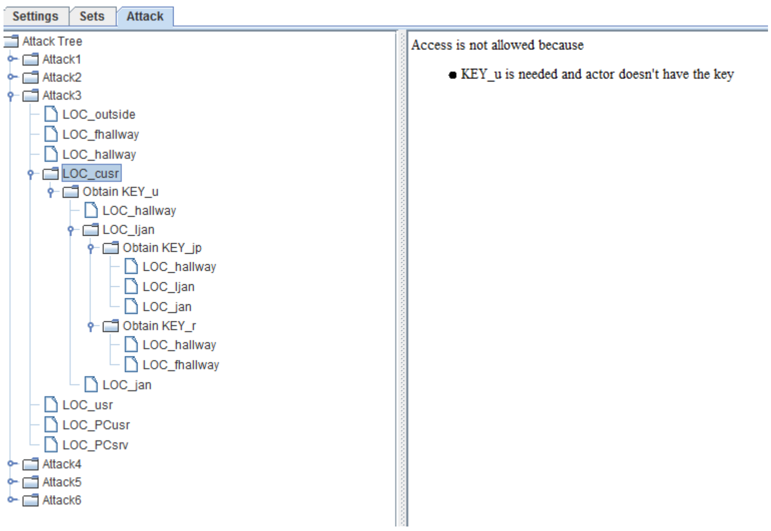
in the other two tabs, the *attack tree* in attack tabs are generated. If some value in the *Settings* or *Sets* tabs are changed then the attack tree is refreshed and updated. Figure 4.6 shows the attack tree generated. The left frame in the figure lists the attack tree and its child nodes (location names). When a node is selected, the reason at that node is displayed in the message box of the right frame. As can be seen from figure, the attack tree can be nested in case the restriction-capabilities test is failed (see Algorithm 3). For instance, the message for selected node LOC_cusr in Figure 4.6 shows that KEY_u was needed to access the location which actor does not possess. Then the nested tree is another attack tree which explains how user can obtain that key from its current location.

Hence while generating attacks in our system model attack trees can be beneficial to mark all the threats with all the description of detailed subtasks which makes computation more precise and focused to one specific area.

4.3.1 Visual Assists in Graph

One of the features in our tool is to provide visual aids in graph representation so that user can get some information while looking at the graph.

In the initial load, all the nodes and connections are of same color here **grey**. Figure 4.7 shows the graph when the user has selected two end location points. The start and end location nodes are changed to the blue color while all the nodes in between them appear in the red. Also, all the routes in between these two locations changed to blue except **cusr** → **usr**. This is because the path color changes from blue to red if that is inaccessible. In this simulation, actor was ACT_j does not have access right at **cusr** and so it cannot go after **cusr**. Paths, which do not fall in the route, did not change their appearance. For example, the paths hallway → ljan is not in the route.



Attack Tree
- Attack1
- Attack2
- Attack3
  - LOC_outside
  - LOC_fhallway
  - LOC_hallway
  - LOC_cusr
    - Obtain KEY_u
      - LOC_hallway
      - LOC_ljan
        - Obtain KEY_jp
          - LOC_hallway
          - LOC_ljan
          - LOC_jan
        - Obtain KEY_r
          - LOC_hallway
          - LOC_fhallway
      - LOC_jan
  - LOC_usr
  - LOC_PCusr
  - LOC_PCsrv
- Attack4
- Attack5
- Attack6

Access is not allowed because

- KEY_u is needed and actor doesn't have the key

Figure 4.6: *Attack Tab* lists all the attacks. Left frame lists all the attacks. Right frame lists the *attack reason* for the selected node in left frame.

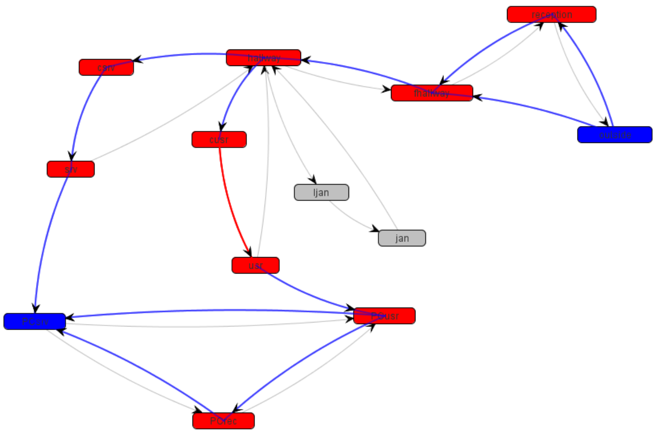


Figure 4.7: Shows the coloring of paths and nodes

Figure 4.8 shows the use of tool tip. When the user hovers mouse over a node then the restriction policies of that particular node appears in the tool tip. As seen in the figure, tool tip of *fhallway* says **ACT_u:!m** and **ACT_j:!m** where *!m* is the logged action. Also, this picture shows when a node is picked the color is change to yellow. Here, fhallway is picked and hence the color changed from blue to red. This is useful when user is navigating the attack tree so that whenever user clicks one node in the attack tree the responding node in the graph seems apparent.

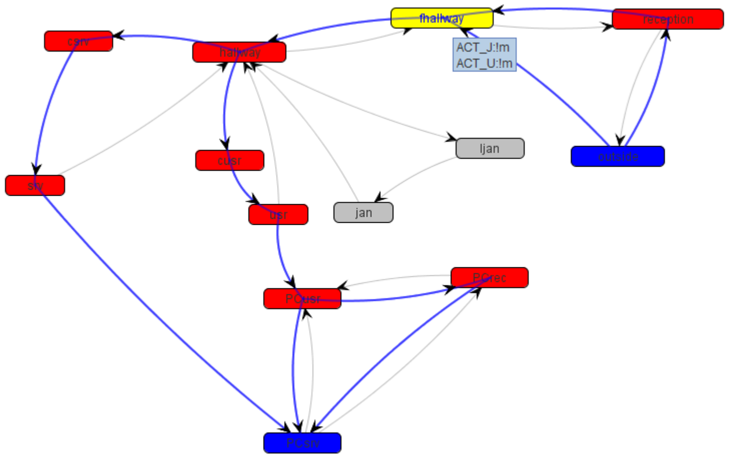


Figure 4.8: Shows the tool tip behaviour when the mouse is hovered over node ljan

4.4 Program Structure

This section will briefly describe how we have design the layout of the program. We will give a high level overview of the program structure. Figure 4.9 gives a basic representation of the structure of our program. As can be seen from the figure, our program can be seen as parts of modules. We have divided different tasks in these modules where each task fits the description of the module in which it belongs. This is to modularize the program so each task or set of related tasks can be viewed as each single entity. This helps in categorization of work and also when someone in the future wants to extend the project by adding extensions to these existing modules or creating new modules. The tool is written in Java and so we view these modules as Java packages in our choice of programming language.

Visualization : This module, as the name indicates, carries all the work necessary for giving a visual layout to the program. It contains graphical codes and interaction with **JUNG** to create the user interface of the program.

Parse: This module holds the information about system components described in section 2.2.2. The specifications of the system maintained by language

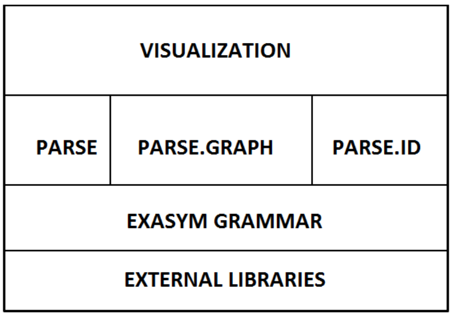


Figure 4.9: Figure shows the basic component design of the program. These components are present at the time and can be extended for later development

grammar is parsed into tokens such as location, actor, connection, data and keys tokens. These tokens are then stored in appropriated data structures. This module contains these data structures and methods to access and operate on the system components. In similar way, the access policies of locations are also handled by this module.

Parse.ID : This module holds the ID information of the system components, for example, location ID, data ID and key ID. This module is used to provide a base for matching IDs with their related components. For example, ID is an abstract class and all the other IDs like location ID, actor ID, key ID are derived from that. So, when we know the ID we can find the matched object to that ID from the **PARSE** module. Also this module implements a boolean function that checks whether an ID is allowed to access the resource. This function is shown in listing 4.1. The implementation of this function takes an ID, figures out what kind of ID is this i.e. key, actor or data id and will then try to find whether with the current ID what access grants are available at current location.

```
boolean allows(Actor currentActor, Location currentLoc,
           Action currentAction);
```

Listing 4.1: A function definition in module *Parse.ID* which returns the result of restriction-capability match

Parse.Graph : This module is made to prepare graph for the system specification. We have discussed earlier in section 3.3 that how we will take a graph based approach for our solution. So in order to achieve that, all the necessary graph works goes in this module. Locations which are vertices or nodes in graph, connections which are edges in the graph are represented in this module in graph based form. Calculation of paths is also carried out in this module.

EXASYM Grammar: This component or module stores the grammar related files. The system specification language is defined in this module. Any changes regarding to the specification language is reflected in this module.

External Libraries: We are dependent on two external libraries for now namely JUNG and ANTLR as discussed earlier. This external libraries module is the collection of these libraries and can be extended with other libraries that can provide any usefulness to the program.

4.5 Summary

In this chapter, we presented the implementation details of our tool. In doing so we presented details on calculating paths and extracting data. We also provided insight on the external libraries namely *JUNG* and *ANTLR* that we are using. The details about various aspects of user interface were discussed. Similarly, we presented the program structure detailing important components or modules of the program.

CHAPTER 5

Evaluation

In the previous chapter, we provided implementation details about our developed tool. This chapter will focus on the evaluation of this tool. To do so, we will set some basic tasks that will be performed by the tool and then we will evaluate the outcomes from the tool. For each task we provide a different set of configurations that are available in the *Settings* and *Sets* tabs. Each separate task shows different utilization and working behaviour of the tool. Another goal is to show the correctness of algorithms and implementation of the tool.

5.1 Task 1: Path Calculations

In this task we will see if the tool outputs the correct path for selected end locations. We have not set any actor or data for this. For the task we will set the source location to LOC_outside and destination location to LOC_PCsrv. At first we want to verify that our tool outputs all the correct available roots for 2 end locations. Figure 5.1 shows the routes that are available between these two paths. The available roots are marked in blue. Greyed edges in the pictures are of no concern, i.e., not in routes. The available paths between these two locations are:

Path 1: LOC_outside → LOC_reception → LOC_fhallway → LOC_hallway →

LOC_csrv → LOC_srv → LOC_PCsrv

Path 2: LOC_outside → LOC_reception → LOC_fhallway → LOC_hallway → LOC_cusr → LOC_usr → LOC_PCusr → LOC_PCsrv

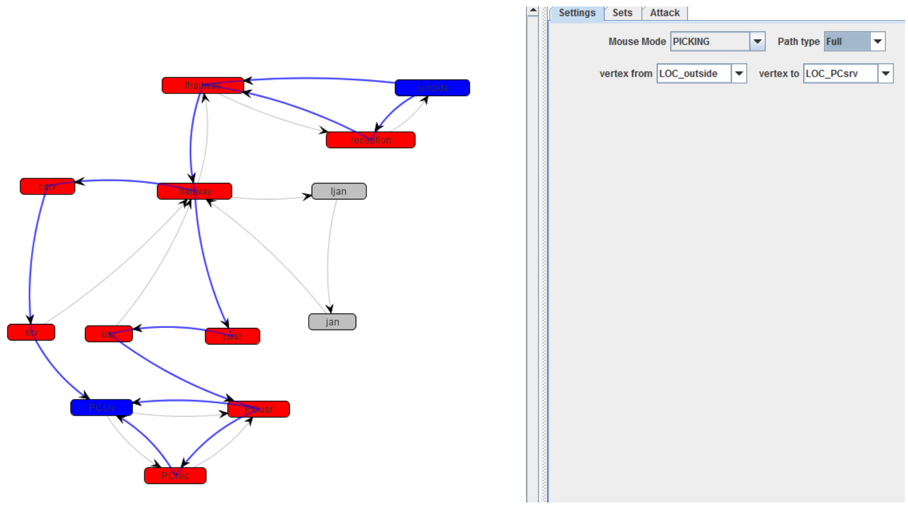
Path 3: LOC_outside → LOC_reception → LOC_fhallway → LOC_hallway → LOC_cusr → LOC_usr → LOC_PCusr → LOC_PCrec → LOC_PCsrv

Path 4: LOC_outside → LOC_fhallway → LOC_hallway → LOC_csrv → LOC_srv → LOC_PCsrv

Path 5: LOC_outside → LOC_fhallway → LOC_hallway → LOC_cusr → LOC_usr → LOC_PCusr → LOC_PCrec → LOC_PCsrv

Path 6: LOC_outside → LOC_fhallway → LOC_hallway → LOC_cusr → LOC_usr → LOC_PCusr → LOC_PCsrv

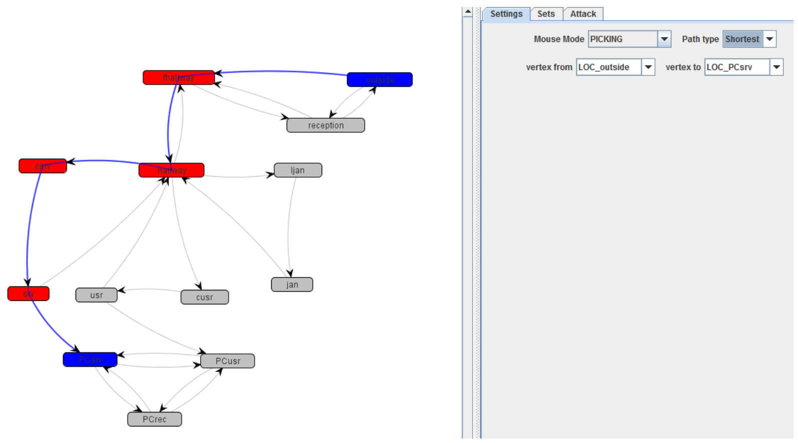
Figure 5.1 calculates all the above 3 mentioned paths and display it correctly. Similary, we now set the tool to calculate the shortest path from LOC_outside



Mouse Mode PICKING
Path type Full
vertex from LOC_outside
vertex to LOC_PCsrv

Figure 5.1: Figure shows that tool outputs all the availbale paths between start location: LOC_outside and end location: LOC_PCsrv. The paths are marked in blue color.

to LOC_PCsrv. From the above calculated paths Path 4 was the shortest one. Figure 5.2 shows the output by the tool while calculating the shortest path. As can be seen from the diagram, the tool outputs Path 4 as the shortest path and hence correctly determines the shortest path. If there are more than one shortest path then one of them is picked in the current implementation of Dijkstra shortest path algorithm in Jung.

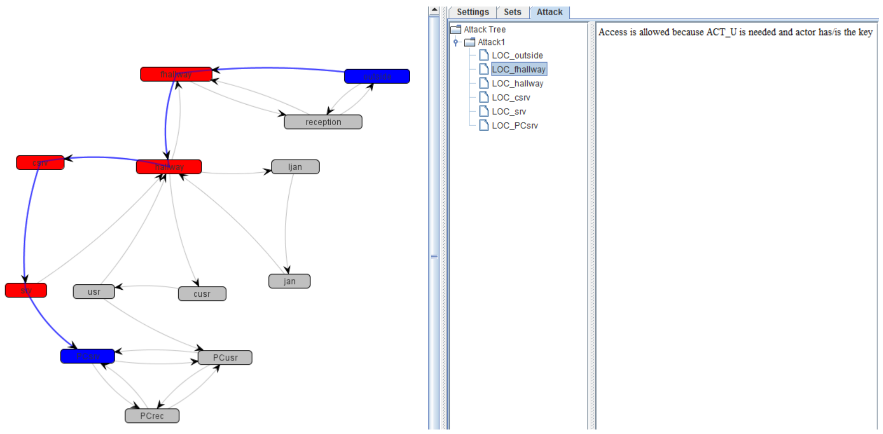


Mouse Mode PICKING Path type Shortest
vertex from LOC_outside vertex to LOC_PCsrv

Figure 5.2: Figure shows that tool outputs only the shortest path between start location: LOC_outside and end location: LOC_PCsrv. The path is marked in blue color.

5.2 Task 2: When actor is present

We will now set the actor in configuration of Task 1. The start locations and end locations are LOC_outside and LOC_PCsrv, the path mode is shortest and the actor is set to ACT_U in the *Sets tab*. Figure 5.3 shows the resulting attack tree. As can be seen from figure 5.3, whenever the user selects a node in attack tree the node in the graph also gets highlighted providing interactivity. So, to travel from LOC_outside and LOC_PCsrv ACT_U has to go through LOC_outside → LOC_fhallway → LOC_hallway → LOC_csrv → LOC_srv → LOC_PCsrv. Listing 5.1 shows the output of the log for this action. From the listing, we can see that ACT_U can reach to the destination from the mentioned source location as its capabilities matches the restriction of the nodes in the path.



Access is allowed because ACT_U is needed and actor has/is the key

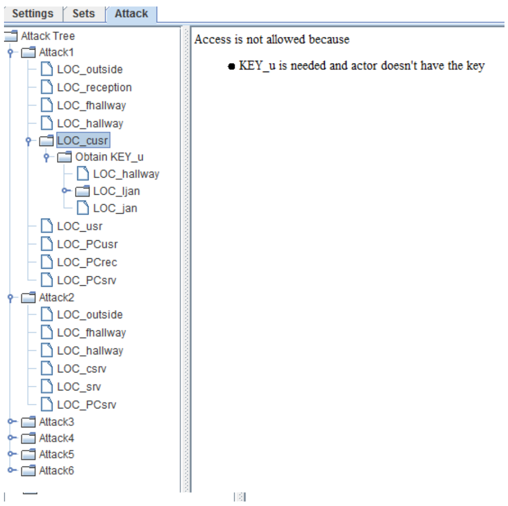
Figure 5.3: Figure shows attack tree being generated for start location: LOC_outside; end location: LOC_PCsrv; actor: ACT_U

```
Attack1
LOC_outside: Access is allowed because any is allowed
LOC_fhallway: Access is allowed because ACT_U is needed
      and actor has/is the key
LOC_hallway: Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_u is needed
      and actor has/is the key
LOC_srv: Access is allowed because any is allowed
LOC_PCsrv: Access is allowed because any is allowed
```

Listing 5.1: Log output for Figure 5.3

Figure 5.4 shows the attack tree for the configuration where we change the actor to be **ACT_J** and path mode to full to show the all sets of attacks generated. We can see from the figure that when full path was chosen a set of attacks has been generated for possible paths instead of a single attack. Also in the snapshot, it is shown that a nested tree occurs if actor could not get pass that node (in this case node LOC_Cusr which is expanded with the nested tree).

Listing 5.2 shows the log sequence for attack number 1 for the above configurations. In this listing, we can see at line 7-8 ACT_J could not get authenticated at the node LOC_cusr because KEY_u is needed there as specified by *LOC_cusr{ KEY_u: m; };* 2.4. KEY_u is located at LOC_jan mentioned in the specification as *KEY_u [LOC_jan];*. Thus, from line 9-24 ways to obtain KEY_u from location LOC_jan is mentioned. To show the nested trees, we added few extra keys in specification example mentioned in section 2.4. KEY_jp and KEY_r were added and defined by *KEY_jp [LOC_jan];* and *KEY_r [LOC_fhallway];*. Location LOC_ljan restrictions reads as *LOC_ljan{ KEY_jp: log_m; KEY_r:log_m;};*.



```
Attack Tree
  Attack1
    LOC_outside
    LOC_reception
    LOC_fhallway
    LOC_hallway
    LOC_cusr
      Obtain KEY_u
        LOC_hallway
        LOC_ljan
        LOC_jan
    LOC_usr
    LOC_PCusr
    LOC_PCrec
    LOC_PCsrv
  Attack2
    LOC_outside
    LOC_fhallway
    LOC_hallway
    LOC_csrv
    LOC_srv
    LOC_PCsrv
  Attack3
  Attack4
  Attack5
  Attack6
```

Access is not allowed because

- KEY_u is needed and actor doesn't have the key

Figure 5.4: Figure shows attack tree being generated for start location: LOC_outside; end location: LOC_PCsrv; actor: ACT_J

We can see at line 15 and line 19 methods to extract those keys are mentioned. Notice at line 21 LOC_ljan is not expanded again because it has been explored earlier at line 11 thus avoiding any cycle.

```
1   Attack1
2   LOC_outside: Access is allowed because any is allowed
3   LOC_reception: Access is allowed because any is allowed
4   LOC_fhallway: Access is allowed because ACT_J is needed
5        and actor has/is the key
6   LOC_hallway: Access is allowed because any is allowed
7   LOC_cusr: Access is not allowed because KEY_u is needed
8       and actor does not have the key
9     Obtain KEY_u
10      LOC_hallway: Access is allowed because any is allowed
11      LOC_ljan: Access is not allowed because KEY_r is needed
12         and actor does not have the key
13            KEY_jp is needed and actor does not have the key
14          Obtain KEY_r
15            LOC_hallway: Access is allowed because
16                any is allowed
17            LOC_fhallway: Access is allowed because ACT_J is
18                  needed and actor has/is the key
19          Obtain KEY_jp
20            LOC_hallway: Access is allowed because
21                any is allowed
22            LOC_ljan: Access is not allowed because KEY_r is
```

```
                needed and actor does not have the key
        KEY_jp is needed and actor does not have the key
              LOC_jan: Access is allowed because any is allowed
     LOC_jan:  Access is allowed because any is allowed
LOC_usr:   Access is not allowed because KEY_u is needed
   and actor does not have the key
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCrec:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed
```

Listing 5.2: Log output for Figure 5.4 showing Attack 1

However, Attack number 2 from the figure shows that ACT_J can gain access if it follows LOC_outside → LOC_fhallway → LOC_hallway → LOC_csrv → LOC_srv → LOC_PCsrv. The listing 5.3 also verifies it and if we look into the grammar then also we can see that ACT_J has enough credentials to pass the nodes lying in this path.

```
Attack2
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
      and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
      and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed
```

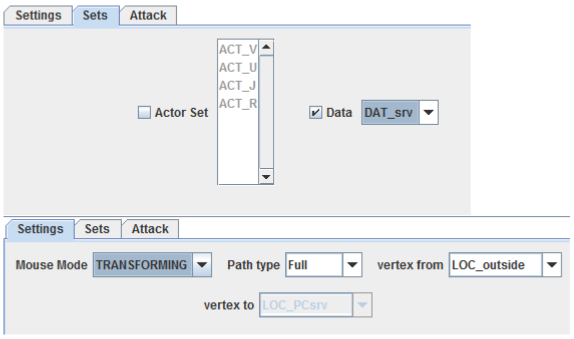
Listing 5.3: Log output for Figure 5.4 showing Attack2

5.3 Task 3: With ActorSet

ActorSet is the collection of actors. In this configuration, the start and end locations are the same. But this time, we choose both ACT_U and ACT_J as the set of actors. Since ACT_U could access these locations, the result of this actorset should result in successful access. This is because though ACT_J did not have permission previously, now the actor set includes ACT_U and the access capabilities of ACT_U matches the restriction of the locations. In other words, it can be viewed as the total knowledge of credentials was combined knowledge of ACT_U and ACT_J. For instance, Listing A.1.3 shows output from the log file with the settings mentioned there. From the log file we can see that no attacks happened. In Attack 1 at line 8 we can see access is allowed for KEY_u because KEY_u belongs to ACT_U which is in our actor set. This verifies that our restriction-capability test is properly using union of credentials of the actors in the actor set.

5.4 Task 4: With data

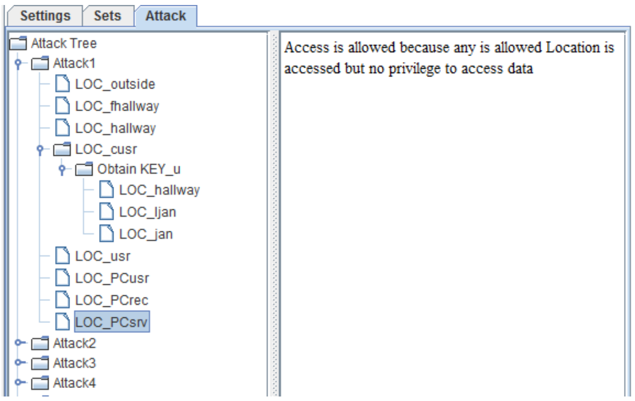
As we have mentioned before, data is located at a location. Accessing data means first the actor has to access the location where data is located. Therefore, when data is selected then the location where data is located is searched. This location is then set as destination location in the settings tab. We choose our data to be *DAT_srv* located at *LOC_PCsrv* as defined by *DAT_srv{KEY_u: r;}[LOC_PCsrv];*. Figure 5.5 shows that when user selects data DAT_srv from the data in *sets tab* then the first thing that is done internally is to find the location of where the data is located. Here the data is located at LOC_PCsrv. The tool correctly lists destination as LOC_PCsrv as shown in *vertex to* section in the figure.



Mouse Mode TRANSFORMING
Path type Full
vertex from LOC_outside
vertex to LOC_PCsrv

Figure 5.5: Figure shows when data is being selected as DAT_srv then the destination location is changed to LOC_PCsrv where data is located as indicated by the changed and greyed *Vertex To*

The Figure 5.6 shows the attack tree for this configuration. This figure is similar to Figure 5.4 except that in this case there is data also. Finding path operation is similar as described before. Once the data is reached, access control check is done on the data. Since this data need KEY_u for reading and current acting actor is ACT_J, the user has no access privilege for the data. This shows the correctness of the application as it correctly displays the path as being traversable and only data as inaccessible which is true in case of the presented task.



Access is allowed because any is allowed Location is accessed but no privilege to access data

Figure 5.6: Figure shows attack tree being generated for start location: LOC_outside; data: DAT_srv (so end location: LOC_PCsrv); actor: ACT_J

5.5 Summary

In this chapter, we saw the functioning and evaluation of our tool. It provides expected results. The complexity of tool will increase with the complexities of the modules extended to it. But for now, the tool seems to output expected result for analysis of reachability from one location to another in (or not in) the presence of actor and data.

Chapter 6

Conclusions And Further Work

6.1 Conclusions

This thesis work provided an explanation of how to generate attacks in a system model. We developed a parser to specify the real world system such as organization buildings and its components. The tool developed here provides a platform where we can perform insider problem analysis on the abstracted analysable system model. The tool gives users options to simulate "Attack Generation" between two location endpoints with or without any set of actors or data. Users are presented with a list of possible attacks that can happen for the specified locations, actors or data.

The tool is developed with a mindset of providing ease, better visualization and freedom in analysis choices. Since the tool is in its early development phase, the components in the tool are well managed so that extensions can be well integrated into the tool for future enhancements. We believe that the framework developed here can be beneficial for the insider problem analysis.

6.2 Further Work

As mentioned in previous section, the tool is in its early development phase. There are number of extensions that can be added to the system to make it more close to the real world system. Some of those extensions that we have come up with but yet remained to be implemented are briefly mentioned below:

- **Complex Tuple Structure** We have used simple tuple structures in the tool, for instance, representing data as simple string. The tool can be extended to deal with complex tuple structures such as nested tuples and so on.
- **Probability** We have mentioned before that we settled for the over approximation of attacks because we did not take probability much into account. For instance, we did not calculate whether an actor is able to social engineer another actor or an actor was able to get the key from the location or not. The tool can use probability algorithms to simulate more realistic situations. For instance, probability algorithms to calculate things such as whether an actor can be social engineered or what information an actor gives another actor.
- **Multiple Key Encryption** The encryption-decryption of key and data presented in the thesis was single encryption-decryption. We can extend the tool to encrypt or decrypt the key or data with multiple keys. For example at present the encryption in the specification looks like {KEY_u:m} however we can model multiple key encryption as {KEY_jKey_u:m} or in some other possible ways. Since the key encryption is single till now in the implementation, our nodes in attack trees are *OR nodes* if the node has any children. If multiple key encryption is implemented then we can also model the **AND nodes** in the attack tree. For example, LOC_cusr:{KEY_jKey_u:m; ACT_U:m} so in the attack tree we can represent that to visit location cusr one need *((KEY_j and KEY_u) or ACT_U).* And visually we can represent these **AND** and **OR** nodes with color combinations in our attack trees. For instance, the branches for KEY_j and KEY_u are of same color so we know that they are **AND** nodes and different color for ACT_U which will suggest that this is *OR*. In this way, with multiple key encryption we can have complex encryption techniques available and we will build a more complete attack tree with both AND and OR nodes.
- **Risk Modelling** The tool can be extended to provide insights on risk management as mentioned in Amenaza [2003]. It would be a nice idea to extend the tool to predict damages and costs done and incurred during

the attack. For instance, in an attack what worth an organization looses and what cost the attacker has to pay to achieve success in that attack. In this task we can annotate all the location nodes with cost to pass the location, skill needed to pass it and success percentage of passing the node. Integrating with *probability* extension mentioned above risk modelling can be valuable for making our tool more closely resemble to the real world.

- **Ranking Attacks** Our attack trees are not ranked now. We are working on ranking algorithms for our tool. Once the ranking attacks extension is prepared we can view the attack trees in order of their ease of completion. The ranking is based on complexity of the path. For instance, to go from *A* to *B* the user has two paths *P1* and *P2*. In *P1*, user does not have to search for another key or social engineer another actor but in *P2* user need to social engineer another actor. So, the attack tree will present *P1* as the first choice because attacker is likely to follow *P1* rather than *P2* as there is no obstacle in *P1*.

- **User Interface** As of now, we feed a specification to the tool which contains all the system specifications. With complex and bigger system these specifications can get large and may be hard for the user to manually code it in language. Also, the language specification being large the user can get confuse and commit errors. For this purpose in future, we can extend our tool in such a way that user can draw the system specification as graph on some drawing canvas provided with the tool. The user should be able to specify actors, keys, data and locations present in the system on the canvas and also later edit/delete/add components directly from the canvas. This way the user does not have to gain the knowledge of the modelling language syntax and a novice user will also be able to use our tool with some basic knowledge.

- **Logs** We have not implemented the LTRA yet in our tool. For now we just keep the logs of the attacks in the text file. We are investigating algorithms to simulate LTRA in best possible way in our defined system. Furthermore, it would be good to integrate these log sequences in some kind of databases so that freedom of operating these log values is leveraged.

Appendix A

Appendix

A.1 Test Output

A.1.1 Test file

```
locations
{
// locations in the building
    domain = DOM_LOC_BuildA;

    LOC_outside    [ area = 1-4,6 ] { *: *; };
    LOC_reception  [ area = 1-4,5 ] { *: *; };
    LOC_fhallway   { ACT_U: log_m; ACT_J: log_m; };
    LOC_hallway    [ area = 1,1-4 ] { *: *; };
    LOC_csrv       { KEY_u: m; KEY_j: log_m; };
    LOC_srv        [ area = 2-4,1 ] { *: *; };
    LOC_cusr       { KEY_u: m; };
    LOC_usr        [ area = 2-4,2 ] { *: *; };
    LOC_ljan       { KEY_j: log_m;};
    LOC_jan        [ area = 2-4,3 ] { *: *; };

// locations in the network
    domain = DOM_NET_BuildA;

    LOC_PCrec [location = LOC_reception] {*:*;};
    LOC_PCusr [location = LOC_usr] {*:*;};
    LOC_PCsrv [location = LOC_srv] {*:*;};
}

connections
{
  LOC_outside : LOC_reception, LOC_fhallway ;
  LOC_reception : LOC_outside, LOC_fhallway ;
  LOC_fhallway : LOC_hallway, LOC_reception;
  LOC_hallway : LOC_csrv, LOC_cusr, LOC_ljan, LOC_fhallway;
  LOC_csrv : LOC_srv;
  LOC_cusr : LOC_usr;
  LOC_ljan : LOC_jan;
  LOC_srv : LOC_hallway, LOC_PCsrv;
  LOC_usr : LOC_hallway, LOC_PCusr;
  LOC_jan : LOC_hallway;

  // connection for dom_network
  LOC_PCrec : LOC_PCusr, LOC_PCsrv;
  LOC_PCusr : LOC_PCrec, LOC_PCsrv;
  LOC_PCsrv : LOC_PCrec, LOC_PCusr;

}

actors
{
 ACT_U{
  known_keys = {KEY_u};
 };
 ACT_J{
  owned_keys = {KEY_j};
 };
 ACT_V{
    // a visitor
 };
 ACT_R{
     // receptionist
  known_keys = {KEY_r};
 };
}
data
{
 DAT_rec{KEY_u: r; KEY_r: r;}[LOC_PCrec];
    DAT_srv{KEY_u: r;}[LOC_PCsrv];
 DAT_usr{KEY_u: r;}[LOC_PCusr];
```

```
}

key{
 KEY_u  [LOC_jan];
 KEY_j  ;
 KEY_r   [LOC_fhallway];
 KEY_m  ;
 KEY_jp  [LOC_jan];
 KEY_t  [LOC_reception];
}
```

Listing A.1: Test Grammar File of which test results are put below

A.1.2 Task 1

Source LOC_outside **Destination** LOC_PCsrv **Path** Shortest **Actor Set** ϕ **Data** null

```
Attack1
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
    and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed
```

A.1.3 Task 2

Source LOC_outside **Destination** LOC_PCsrv **Path** Full **Actor Set** ACT_U, ACT_J (Almost similar for ACT_U as well as ϕ) **Data** null

```
Attack1
LOC_outside:  Access is allowed because any is allowed
LOC_reception:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
      and actor has/is the key Access is allowed
      because ACT_U is needed  and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:  Access is allowed because
   KEY_u is needed  and actor has/is the key
```

```
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCrec:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack2
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key Access is allowed
    because ACT_U is needed  and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
   and actor has/is the key Access is allowed
   because KEY_u is needed  and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack3
LOC_outside:  Access is allowed because any is allowed
LOC_reception:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key Access is allowed
    because ACT_U is needed  and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:  Access is allowed because KEY_u is needed
   and actor has/is the key
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack4
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key Access is allowed because
    ACT_U is needed  and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:  Access is allowed because
    KEY_u is needed  and actor has/is the key
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack5
LOC_outside:  Access is allowed because any is allowed
LOC_reception:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key Access is allowed
    because ACT_U is needed  and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
    and actor has/is the key Access is allowed
    because KEY_u is needed  and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack6
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is
    needed  and actor has/is the key Access is allowed
    because ACT_U is needed  and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:  Access is allowed because KEY_u is needed
   and actor has/is the key
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCrec:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed
```

A.1.4 Task 3

Source LOC_outside **Destination** LOC_PCsrv **Path** Full **Actor Set** ACT_J **Data** null

```

Attack1
LOC_outside:  Access is allowed because any is allowed
LOC_reception:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:   Access is not allowed because
    KEY_u is needed and actor does not have the key
    Obtain KEY_u
      LOC_hallway:  Access is allowed because any is allowed
      LOC_ljan:  Access is allowed because KEY_j is
         needed  and actor has/is the key
      LOC_jan:  Access is allowed because any is allowed
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCrec:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

```

```
Attack2
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
    and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack3
LOC_outside:  Access is allowed because any is allowed
LOC_reception:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
    and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack4
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:   Access is not allowed because
    KEY_u is needed and actor does not have the key
    Obtain KEY_u
     LOC_hallway: Access is allowed because any is allowed
     LOC_ljan:  Access is allowed because KEY_j is
        needed  and actor has/is the key
     LOC_jan:  Access is allowed because any is allowed
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCrec:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack5
LOC_outside:  Access is allowed because any is allowed
LOC_reception:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:   Access is not allowed because
    KEY_u is needed and actor does not have the key
    Obtain KEY_u
```

```
     LOC_hallway: Access is allowed because any is allowed
     LOC_ljan:  Access is allowed because KEY_j is
        needed  and actor has/is the key
     LOC_jan:  Access is allowed because any is allowed
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed

Attack6
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_cusr:   Access is not allowed because
    KEY_u is needed and actor does not have the key
    Obtain KEY_u
     LOC_hallway: Access is allowed because any is allowed
     LOC_ljan:  Access is allowed because KEY_j is
        needed  and actor has/is the key
     LOC_jan:  Access is allowed because any is allowed
LOC_usr:  Access is allowed because any is allowed
LOC_PCusr:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed
```

A.1.5 Task 4

Source LOC_outside **Destination** LOC_PCsrv **Path** Full **Actor Set** ACT_J **Data** DAT_srv

```
Attack1
LOC_outside:  Access is allowed because any is allowed
LOC_fhallway:  Access is allowed because ACT_J is needed
    and actor has/is the key
LOC_hallway:  Access is allowed because any is allowed
LOC_csrv:  Access is allowed because KEY_j is needed
    and actor has/is the key
LOC_srv:  Access is allowed because any is allowed
LOC_PCsrv:  Access is allowed because any is allowed
    Location is accessed but no
    privilege to access data
```

Bibliography

Amenaza. Creating secure systems through attack tree modelling. 2003. URL `http://www.amenaza.com/downloads/docs/5StepAttackTree_WP.pdf`.

Robert H. Anderson and Richard Brackney. Understanding the insider threat. In *Proceedings of a March 2004 Workshop*, Santa Monica, CA, U.S.A., 2004. RAND Corporation, CF-196-ARDA.

Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. An infrastructure language for open nets. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 373–377, New York, NY, USA, 2002. ACM. ISBN 1-58113-445-2. doi: http://doi.acm.org/10.1145/508791.508862. URL `http://doi.acm.org/10.1145/508791.508862`.

Matt Bishop. Position: "insider" is relative. In *Proceedings of the 2005 workshop on New security paradigms*, NSPW '05, pages 77–78, New York, NY, USA, 2005. ACM. ISBN 1-59593-317-4. doi: http://doi.acm.org/10.1145/1146269.1146288. URL `http://doi.acm.org/10.1145/1146269.1146288`.

14th Annual CSI Computer Crime and Security Survey Executive Summary, 2009. Computer Security Institute (CSI), CSI. URL `http://www.pathmaker.biz/whitepapers/CSISurvey2009.pdf`.

Trajce Dimkov, Wolter Pieters, and Pieter Hartel. Portunes: Representing attack scenarios spanning through the physical, digital and social domain. In Alessandro Armando and Gavin Lowe, editors, *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, volume 6186 of *Lecture Notes in Computer Science*, pages 112–129. Springer Berlin / Heidelberg, 2011. URL `http://dx.doi.org/10.1007/978-3-642-16074-5_9`. 10.1007/978-3-642-16074-5_9.

Boris Dragovic and Jon Crowcroft. Containment: from context awareness to contextual effects awareness. In *2nd Inernational Workshop on Software Aspects of Context*, Santorini, Greece, July 2005.

Danyel Fisher, Joshua O'madadhain, Padhraic Smyth, Scott White, and Yan-Biao Boey. Analysis and Visualization of Network Data using JUNG. *Journal of Statistical Software*, 2005. URL `http://http://www.jstatsoft.org/http://www.jstatsoft.org/`.

Daniele Gorla and Rosario Pugliese. Resource access and mobility control with dynamic privileges acquisition. In Jos Baeten, Jan Lenstra, Joachim Parrow, and Gerhard Woeginger, editors, *Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 188–188. Springer Berlin / Heidelberg, 2003. URL `http://dx.doi.org/10.1007/3-540-45061-0_11`. 10.1007/3-540-45061-0_11.

Rene Rydhof Hansen, Christian W. Probst, and Flemming Nielson. Sandboxing in myklaim. In *The First International Conference on Availability, Reliability and Security*, pages 174–181, Vienna, Austria, April 2006. IEEE Computer Society.

Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 24(5):315–324, May 1998.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN 978-3-540-65410-0.

Christian Probst, Rene Rydhof Hansen, and Flemming Nielson. Where can an insider attack? In Theo Dimitrakos, Fabio Martinelli, Peter Ryan, and Steve Schneider, editors, *Formal Aspects in Security and Trust*, volume 4691 of *Lecture Notes in Computer Science*, pages 127–142. Springer Berlin / Heidelberg, 2007. URL `http://dx.doi.org/10.1007/978-3-540-75227-1_9`. 10.1007/978-3-540-75227-1_9.

Christian W. Probst and Rene Rydhof Hansen. An extensible analysable system model. *Information Security Technical Report*, 13(4):235 – 246, 2008. doi: DOI:10.1016/j.istr.2008.10.012. URL `http://www.sciencedirect.com/science/article/pii/S1363412708000502`.

Christian W. Probst and Rene Rydhof Hansen. Analysing access control specifications. *Systematic Approaches to Digital Forensic Engineering, IEEE International Workshop on*, 0:22–33, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/SADFE.2009.13.

Bruce Schneier. Modeling security threats. *Dr. Dobb's Journal*, 1999. URL `http://www.schneier.com/paper-attacktrees-ddj-ft.html`.

Frank Swiderski and Window Snyder. *Threat Modeling.* Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619913.