Network Security Management with Security Policies

Vinoth Sivasubramanian
Indian Institue of Technology - Mumbai
vinoth.sivasubramanian@gmail.com

Abstract

A key issue in network security management is how to define a formal security policy. A good policy specification should be easy to get right and relatively stable, even in a dynamically changing network. Much work has been done in automating network security management. But the policy languages used are usually operational and do not explicitly express the underlying security goal.

We propose an approach where policy is defined as statements of desired security properties, whose compliance can be checked automatically by analyzing the configuration of the network. We use a simple policy model, the data access-control list (DACL) to demonstrate this idea. We present a framework and corresponding algorithms for checking that low-level configurations altogether uphold the high-level DACL policy, taking into consideration potential software vulnerabilities.

1 Introduction

The rapid growth in the size and complexity of organizational networks will soon make the current way of manual management infeasible. Recent years have seen many tools developed to automate this process [3, 6, 12]. There are also tools that scan networks and discover possible attack scenarios involving complex combination of multiple vulnerabilities [1, 10]. However, what lacks is a formal specification of security management goals in terms of a high-level security policy, whose compliance can be automatically verified.

We envision a world in which an administrator would manage his network as follows.

1. Define a high-level security policy as rules like "only allow project managers to access project plans".
2. Use tools such as HP OpenView[1] and Nessus[2] to collect various kinds of network configuration information. In our example, this may include which file server the project plan is stored on, what version of file sharing service is running on the server, and so on.
3. Use a tool (such as the one described in this paper) to check that low-level configurations collectively uphold the high-level policy.

In this paper, we describe our architecture for a tool that can check that a high level security policy is collectively guaranteed by the various elements of the network.

1. We formally specify a high-level policy language for confidentiality and integrity.
2. We show how to formally describe the configurations of various network elements.
3. We design an algorithm that takes as input a high level policy and configuration descriptions, and checks that the configurations conform to the policy. In case the tool detects a policy violation it outputs a trace of actions leading to it.

[1] http://www.openview.hp.com/
[2] http://www.nessus.org/

1.1 High-level security policy

Most existing tools for automatic network management adopt a policy-based approach. System administrators decide upon a global policy specifying how the network should be configured. The tools can verify that a given policy is correctly implemented by low-level mechanisms. In some cases, they can also translate policies into sets of configuration directives and push them to the corresponding network devices. In order to make sure certain security requirements are met, the administrator only needs to examine the policy, which is easier and less error-prone than examining every piece of the configuration.

While the separation of policy from mechanisms is an important step towards eliminating human errors, an equally important question is how to make policy itself less error-prone. A good policy language design should require little technical knowledge to write a "correct" policy. However, this ideal is often hard to achieve, largely because most security problems are caused by complex interactions among different network components. The correct behavior of a device is not only dependent on its own configuration, but also on other devices in the network. This kind of interactions are usually explicitly expressed in policies, either in the form of policy constraints [12, 15] or as a set of conditions associated with a policy rule [11]. This puts more burden on the policy maker to take caution in defining those constraints or policy conditions. Any mistake in this process may lead to insecure system. For example, a policy regarding logging in to hosts may have a rule "*allow logging in to A only if the protocol is SSH*". The purpose is to prevent machine A's data from being transmitted as plain text. However, one can telnet to a machine B (if that is allowed by the policy) and then ssh into A, potentially leaking information communicated with A through the telnet hop. To fix this problem one can either set a constraint on policy rules to avoid any such insecure multi-hop logins, or put an extra condition on A's login policy like "*allow logging in to A from B only if the protocol is SSH and communication to B is secure*". While it is relatively easy to express the inter-domain configuration constraints in this simple example, in reality they can be too complex to specify and reason about. A better policy should start from a higher level, i.e. the goal of security administration. For the login example, a high-level policy could be "*information communicated with machine A should not appear as plaintext in the network*". This way we abstract away the complex interactions among different components and state the ultimate effect desired. In comparison to low-level policies, a high-level policy says "what we want" instead of "what to do". Since it closely matches the intention of policy decision-maker, a high-level policy is easier to get right.

Another problem with low-level policies is that they are often designed to control a particular kind of resource. For example, a packet-filtering policy only controls configuration elements such as firewalls, routers, and network topology. Changes in other configuration elements may require modification to the packet-filtering policy in order to achieve the ultimate security goal. Frequent policy update is undesirable because a policy's correctness affects security. A high-level policy, on the other hand, is much more stable because the goals of security administration usually do not change very frequently.

Figure 1 illustrates the proposed framework. Multiple low-level policies describe different parts of configurations; for each low-level policy domain, there will be a tool to check that implementations conform with the low-level policy (or in other words, the low-level policy is a correct description of the actual network). A checker described in this paper will detect violations of the high-level policy, given a set of low-level configuration descriptions. The high-level policy does not replace the low-level ones. Instead, it is used to reason about them — making sure that together they satisfy certain security criteria.

1.2 Dealing with software vulnerabilities

Many security problems are caused by software vulnerabilities. Thus when configuring a network, a system administrator must take into consideration the security robustness of installed software components. While one should generally avoid running bug-ridden software, the ubiquitous existence

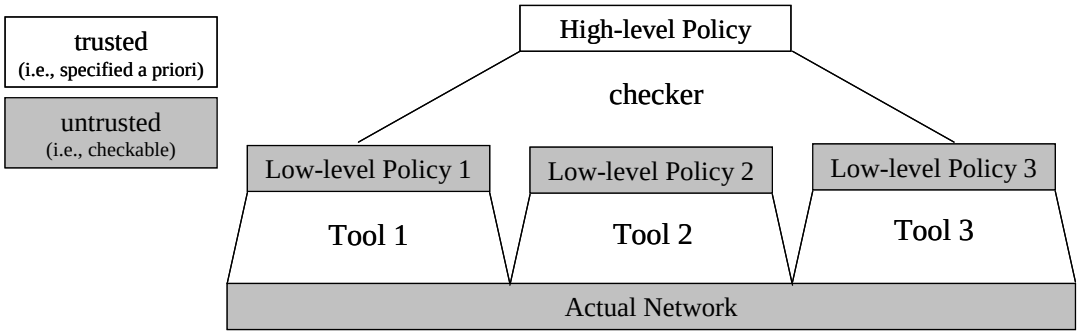

Figure 1: Levels of security policy

of vulnerabilities is a status quo that has to be dealt with for the foreseeable future. And it is a harsh reality that some vendors of popular software do not release patches for even a critical vulnerability quickly. In the wake of a new bug report, an automatic tool that quickly identifies what security risks it brings to the network would be useful. This allows the system administrator to take preemptive actions before a patch is available — modifying firewall rules, moving sensitive information from potentially compromisable zones or disabling the buggy software if there is no other alternative. To this end, the tool presented in this paper takes potential software vulnerabilities into consideration when checking compliance with high-level policies.

Software bugs can be quite subtle and it requires expertise to fully understand their security impact. Since new varieties of exploits emerge every day, security administrators need to pay attention to reports like CERT advisory or BugTraq to keep updated on the most recent threats. This poses a significant challenge for incorporating software vulnerabilities into automatic security management: the tool must be able to take as input new software bugs and reason about them. This requires a formal language to specify software vulnerabilities. These formal descriptions could be written by security experts from an outside source like CERT and local administrators would only need to plug them into the tool.

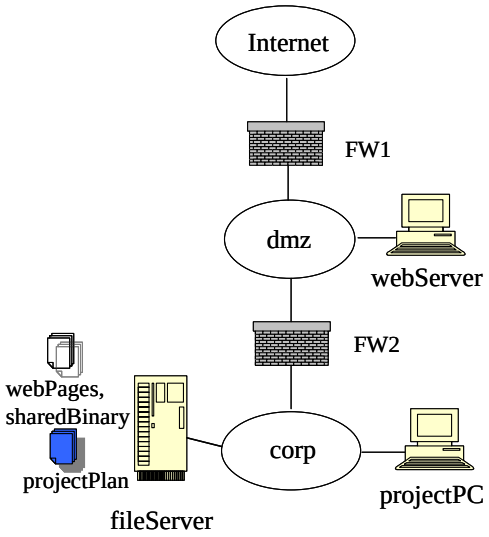

Figure 2: Example

2 A Motivating Example

We use the example network in Figure 2 to illustrate our approach. There are three zones (`Internet`, `dmz` and `corp`) separated by two firewalls (`FW1` and `FW2`). The administrator manages the `webServer` and the `fileServer` while the `projectPC` is operated by corporate employees. The company owns proprietary information so the security management needs to ensure that their confidentiality will not be compromised by an outside attacker. To achieve this security goal, multiple configuration elements must be set up appropriately. First, the topology and firewall configuration allow outside packets to reach `dmz` zone, but not `corp` zone where the confidential `projectPlan` is stored. Second, the

file sharing service running on `fileServer` requires authentication for access to `projectPlan`. Only project managers have the access rights. Third, the administrator maintains a collection of application binaries so that individual employees do not need to install programs on the `projectPC`.

This scheme may look quite secure — even if an attacker can compromise `webServer`, it is still hard for him to get his hands on the confidential data. Since `FW2` only allows `webServer` to communicate with `fileServer`, the attacker will have to somehow compromise `fileServer` to proceed. If the file sharing service is secure, it seems that we can rest assured.

However, a dedicated and clever attacker can still cause a security breach in this configuration. Remember `webServer` is managed by the administrator. So if `webServer` is compromised, it is likely that the credential of the administrator will also be leaked to the attacker, through a password sniffer for example. The administrator's credential does not enable the attacker to access `projectPlan` on `fileServer` (it can only be accessed by project managers). However, it does allow the attacker to update the application binaries. So he can install his version of Acrobat Reader. Some day a project manager will open a project plan in PDF format,and besides showing the file, the Trojan horse Acrobat Reader communicates the content to the attacker.

A safer way to configure the network is to move web pages from `corp` zone to `dmz` zone and ban any inbound access to `corp` zone.

In our management framework, the administrator can define a high-level policy for data confidentiality such as "`projectPlan` can only be read by project managers". He can then run our tool to check if the policy is upheld. In this case the tool will report a violation and attack steps as described above.

3 DACL — A High-level Policy

Our high-level policy language is in the form of data access list (*DACL*). The grammar of the language is defined in Figure 3.

A data access list is a list of data access rules (*DataAccRule*). Each rule specifies a legal operation a subject (*Principal*) is permitted to perform on an object (*Data*). *Principal* is represented by a list of symbols. Each symbol stands for a group of people. The list of symbols stands for the union of the groups represented by each symbol. A similar representation is used for *Data*. *Op* can be either **read** or **write**. A DACL policy for the example is shown below.

$$
\begin{array}{rcl}
\textit{Principal} & ::= & \texttt{p} \mid \texttt{p}; \textit{Principal} \\
\textit{Data} & ::= & \texttt{d} \mid \texttt{d}; \textit{Data} \\
\textit{Op} & ::= & \textbf{read} \mid \textbf{write} \\
\textit{DataAccRule} & ::= & \textbf{allow}\ \textit{Principal Op Data} \\
\textit{DACL} & ::= & \textit{DataAccRule} \\
 & \mid & \textit{DataAccRule}; \textit{DACL}
\end{array}
$$

Figure 3: The DACL Policy Language

Insiders = `projectManager`; `sysAdmin`.
Everyone = `outsiders`; *Insiders*

sysAdminData = `webPages`; `sharedBinary`

allow *Everyone* **read** `webPages`;
allow *Insiders* **read** `sharedBinary`;
allow `projectManager` **read** `projectPlan`;
allow `sysAdmin` **write** *sysAdminData*;

There is a subtle difference between the semantics of DACL and that of a low-level access-control policy. A DACL rule cannot simply be translated into some enforcement mechanism that controls access to the data. As shown by the example, an attacker can steal confidential information from the `fileServer` without violating the file server's access control mechanism. Rather, he exploits vulnerabilities in `webServer`, which is allowed to access `fileServer` by the firewall. The security of a piece of data is affected not only by the authentication mechanisms on the host where it is stored, but also depends on the "hardness" of all machines that can access the host. The security property specified by DACL is enforced *collectively* by the configurations of all those hosts. Thus, to verify that the high-level DACL policy is upheld, all configuration parameters in the network must be taken into account.

DACL is a positive policy — there are no "deny" rules and anything not explicitly allowed is forbid-

den. In such a policy, access rules are independent of each other and there are no possible conflicts among them. This further eases the job of policy-making. For example, to make sure a policy rules out certain data access, it suffices to examine every rule independently, without having to worry about interactions among positive and negative rules.

4 Configuration Description

To verify that a DACL policy is upheld, our checker needs descriptions of the actual network. Part of the descriptions can be low-level policies that control certain aspects of network configurations. In particular, a host access control list (HACL) is a good abstraction for the overall effects of packet-filtering devices and there are existing firewall management tools that can be used to relate a HACL policy to the actual status of those devices [2, 6]. So we use HACL as the description language for those devices.

This section shows description languages for the other configuration information — information about principals and hosts.

4.1 Principal binding

$$
\begin{array}{rcl}
\mathit{Usr} & ::= & \texttt{privileged} \mid \texttt{unprivileged} \\
\mathit{PLoc} & ::= & \mathit{Usr}@\mathit{Hostgrp} \\
\mathit{Trust} & ::= & \texttt{trusted} \mid \texttt{incompetent} \mid \texttt{malicious} \\
\mathit{PBnd} & ::= & \mathit{Principal} : [\ \textbf{location} = \mathit{PLoc},\ \textbf{trust} = \mathit{Trust}\ ]
\end{array}
$$

A principal binding (*PBnd*) provides security-relevant information about people. The **location** field includes the hosts from which the principal can operate and the local user account he has on the hosts. As a first step we only differentiate two kinds of users: `privileged` and `unprivileged`. The **trust** field indicates the principal's trustworthiness. A principal is `trusted` if his intention is always good and it is unlikely he will perform potentially dangerous operations. An example of such operations is opening attachments in unsolicited emails. A principal is `incompetent` if his intention is always good but he may inadvertently perform those potentially dangerous operations. A `malicious` principal has bad intentions and may try to illegally access information by launching attacks.

A sample principal binding for the example is shown below. In reality one can write a program to query a LDAP database to get the information, although we have not done so.

```
sysAdmin :
      [ location = privileged@
                   (fileServer; webServer),
        trust = trusted ]
projectManager :
      [ location = unprivileged@projectPC,
        trust = incompetent ]
outsiders :
      [ location = @Internet,
        trust = malicious ]
```

The importance of principal binding information in policy verification lies in two aspects. First, the locations of malicious principals provide an initial attack condition for the attack simulation algorithm; second, the locations of `incompetent` principals provide a set of targets for client-side attacks. These will become clear in section 5.

4.2 Host configuration

The language for describing configuration of a host is shown below.

$$
\mathit{HostConf} ::= \mathit{Host} : [\ \textbf{app} = \mathit{AppList},\ \textbf{data} = \mathit{DataList}\ ]
$$

A host configuration (*HostConf*) includes the network address of the host (*Host*)[3], descriptions of applications running on the host, and descriptions of data accessible from the host. The latter two are explained below.

Application Description

$$
\begin{array}{rcl}
\mathit{Option} & ::= & \mathit{Tag} = \mathit{Value} \\
\mathit{OptionList} & ::= & \texttt{nil} \mid \mathit{Option}; \mathit{OptionList} \\
\mathit{App} & ::= & (\mathit{SwID}, \mathit{Version}, \mathit{Usr}, \mathit{OptionList}) \\
\mathit{AppList} & ::= & \texttt{nil} \mid \mathit{App}; \mathit{AppList}
\end{array}
$$

An application (*App*) is a piece of software running under the privilege of *Usr* and with certain configuration options (*OptionList*). Each software

[3] If DHCP is used, *Host* is the range of network addresses the host may be assigned.

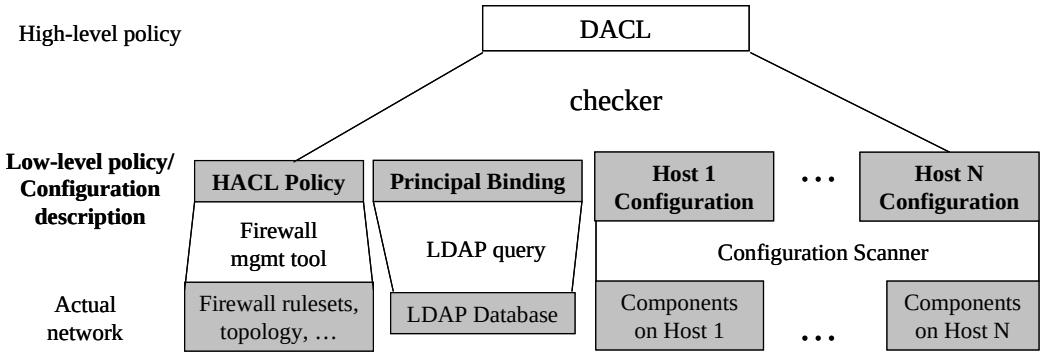

Figure 4: Configuration description languages

is given a unique ID (*SwID*). The configuration options are application-specific but they all come as a list of tags and the corresponding values for the tags.

An example configuration of an Apache web service is shown here.

$$(\texttt{apache},\ \texttt{1.3},\ \texttt{privileged},\ (\texttt{port} = \texttt{80}; \texttt{nil}))$$

The software is `apache` version `1.3`. It is running under the permission of a privileged user, with option `port` set to `80`.

Data binding

$$\begin{array}{rcl} \textit{DataAcc} & ::= & \textit{Usr}\ \textbf{readable} \mid \textit{Usr}\ \textbf{writable} \\ \textit{FileMnt} & ::= & \textit{Protocol}\ \textit{HMntInfo} \\ \textit{DataLoc} & ::= & \textbf{local} \mid \textbf{remote}\ \textit{FileMnt} \\ \textit{DataBnd} & ::= & \textit{Data} : [\ \textbf{access} = \textit{DataAcc} \\ & & \quad \textbf{location} = \textit{DataLoc}\ ] \\ \textit{DataList} & ::= & \texttt{nil} \mid \textit{DataBnd}; \textit{DataList} \end{array}$$

Data binding relates a conceptual notion of data to its physical representation on a host. The **access** field indicates the access control mechanism on the local host to protect the data. For example, "`privileged` **writable**" means privileged users can modify the data. The **location** field tells if the data is locally stored on the host or resides in a remote file server. For the latter case, the file transfer protocol, the address of the server, and relevant mount information must be provided. If the same data has more than one **access** or **location** attributes, there will be multiple bindings for the data.

A sample data binding for `projectPlan` on `projectPC` is shown below.

```
projectPlan : [
  access = unprivileged readable
  location = remote AFS fileServer
             credential(projectManager) ]
```

The binding information shows an unprivileged user can issue a read request to `projectPlan`, which is stored remotely at `fileServer` through the AFS file sharing protocol. The mount information indicates the credential of `projectManager` is necessary for the request to succeed.

A sample configuration description for the three hosts in the example is shown in Figure 5. In reality a configuration scanner such as Nessus or a host management tool such as HP OpenView can be used to gather this information.

5 Compliance Checking

In this section we address the problem of the higher-level check — assuming a given configuration description is a truthful representation of the actual network, how can we reason about it in order to determine whether the high-level policy is upheld. We achieve this goal in two stages.

In the first stage, we model the whole network as a state-transition system. This gives us the ability to reason about multi-stage attacks. The transition rules are contributed by all components in the system. For each component class, there is a general specification as to what kind of transition rules the component generates under various configuration options. From this specification, combined with the component description information,

```
webServer : [
        app = (apache, 1.3, privileged, (port = 80; nil)); nil
        data = (webPages; sharedBinary) : [
                        access = unprivileged readable,
                        location = remote AFS fileServer ];
                (webPages; sharedBinary) : [
                        access = privileged writable,
                        location = remote AFS fileServer
                                        credential(sysAdmin) ];
                nil. ]

fileServer : [
        app = (OpenAFS, 1.2.11, privileged,
                (export_list =
                        ((webPages; sharedBinary), readAccess, Anyone);
                        ((webPages; sharedBinary), writeAccess, sysAdmin);
                        (projectPlan, readAccess, projectManager);
                        nil);
                nil)
        data = nil. ]

projectPC : [
        app = nil,
        data = sharedBinary : [
                        access = unprivileged readable,
                        location = remote AFS fileServer ];
                projectPlan : [
                        access = unprivileged readable,
                        location = remote AFS fileServer
                                        credential(projectManager) ];
                nil. ]
```

Figure 5: Example host configurations.
(All this data could be automatically collected)

we can uniquely determine the set of transition rules a particular instance of the component contributes. The transition rules of the whole network system is the collection of all transition rules contributed by every component on every host.

In the second stage, we simulate attacks on the state-transition system. An initial attack condition is introduced in the system and an efficient algorithm finds out the final state under the transition rules. Conditions in the final state are examined to see if any of them is in violation of the policy.

Section 5.1 discusses the component specification language, and section 5.2 discusses the attack simulation algorithm.

5.1 Component specification

A component specification describes the transition rules a software or data module generates under various configuration situations. This specification can be provided by an outside source with expertise in security analysis. Local administrators only need to supply the relevant configuration information. Thus, component specifications should be general so that they can be reused across sites.

Following is an example specification for a buffer overflow bug in certain versions of Apache:

```
specDef{
      SwID = apache
      Version = 1.2.2 - 1.3.24
      perm = Usr
      net in anyHost http exploit
              ==> fullControl Usr
}
```

Intuitively, the above specification says "if a software component `apache` with version from 1.2.2 to 1.3.24 is running under the permission of `Usr`, and a malicious packet from an attacker arrives through `http` protocol, then he can hijack the software and do whatever `Usr` is allowed in the local system.

Actually, this is just a formalization of the following excerpt from the CERT Advisory regarding this bug[4]:

> For Apache versions 1.2.2 through 1.3.24 inclusive, this vulnerability may allow the execution of arbitrary code by remote attackers ...

[4] http://www.cert.org/advisories/CA-2002-17.html

According to the configuration description of `webServer`, the `apache` component in our example contributes the following transition rule to the local host:

```
net in anyHost http exploit
        ==> fullControl privilegedUsr
```

A transition rule has the form $Pre \Longrightarrow Post$. It means if condition Pre is true at the host, condition $Post$ will also be true at the host. A condition is either a predicate applied to its arguments, or a conjunction of two conditions:

$$C \quad ::= \quad PX_1 \ldots X_n \mid C \& C$$

There is a collection of predefined predicates in our system. For example, `net` is a predicate that takes four arguments: a direction (either `in` or `out`), a host group, a protocol and an *annotation* of the network packet. The `exploit` annotation indicates this is a malicious packet that contains an exploit of the bug in the software. Specification writers can also define new predicates.

A *satisfaction relation* is defined among conditions. For example, condition

`net in attackerHost http exploit`

satisfies condition

`net in anyHost http exploit`.

Specification writers can specify the satisfaction relation for a new predicate they introduce, or the default relation, which only relates two identical conditions, will be used.

We find this way of specifying components quite flexible in expressing various security-relevant behaviors of software and data modules. For example, one can specify the `AFS` file sharing service as follows.

```
specDef{
      SwID = AFS
      Version = anyVersion
      perm = privilegedUsr
      export_list = ((Data, Acc, Cred)
                     ; nil)
      (net in anyHost AFS_RPC
             (normal (access Acc Data))
       & owns Cred
   ==> access Acc Data))
}
```

This is a simplified specification — there is only one entry in `export_list`. Our specification language has libraries that include list operations, which

can be used to write a full AFS specification. `normal` is a packet annotation (like `malicious`). It takes as argument a requested condition on the destination host. `access` is a predefined predicate which means the attacker has certain access (read or write) to the data. The specification indicates the attacker's request will succeed only if he owns the appropriate credential.

Given specifications of the software and data modules, one can generate a set of transition rules for a host from the host configuration information. Once the transition rules for every host are generated, the next step is to simulate attacks on the network modeled as the state-transition system.

5.2 Attack simulation

We give definitions for the state of a host and the state of a network.

Definition 1 *A host state is a triple (H, R, C), where H is the network address of the host, R is a set of transition rules and C is a set of conditions that reflects the attacker's status on the host.*

Definition 2 *A network state is a tuple $(S_1, \ldots, S_n, G, G_0)$, where S_i is the state of a host in the network, G is a set of global conditions, and G_0 is the global conditions that have not been propagated to the hosts.*

The reason to include transition rules as part of the host state is that new rules may be generated during simulation. For example, a host initially has rule $A_1 \& A_2 \Longrightarrow B$ and at one stage of simulation condition A_1 becomes true. Then a new rule $A_2 \Longrightarrow B$ will be added to the rule set for the next stage.

The set of global conditions in the network state is used to propagate attacks from one host to another. For example, if an attacker compromises host `H` it can generate a condition of "scanning the network for new victims":

```
net out anyHost anyProtocol exploit
```

the global form of this local condition is

```
netGlobal H anyHost anyProtocol exploit
```

Several functions are used in the attack simulation algorithm:

Input: $S = (H, R, C)$, and G

1. $C_0 =$ **condLocal**(H, G);
   $C \leftarrow C_0 \cup C$; $C_g = \emptyset$
2. $(R', C_0') =$ **apply**(R, C_0);
   $R \leftarrow R'$; $C_0 \leftarrow C_0' \setminus C$
3. If C_0 is empty, terminate and output
   $((H, R, C),\ C_g)$.
4. $C \leftarrow C_0 \cup C$,
   $C_g' =$ **condGlobal**(H, C_0), $C_g \leftarrow C_g' \cup C_g$
   goto step 2

Figure 6: Algorithm **hostStateTrans**

condLocal(H, G) finds in a set of global conditions the ones that are relevant to host H and converts them to the local form.

condGlobal(H, C) is the reverse process: find in a set of local conditions the ones that affect other hosts and convert them to the global form.

apply(R, C) finds all applications of rules in R to conditions in C. It outputs the resulting conditions and a rule set that includes both R and any new rules generated by the application.

filter$(HACL, G)$ filters the network conditions in G according to a given firewall policy *HACL*. It leaves non-network conditions in G unchanged.

In each step of simulation, the new global conditions in G_0 are propagated to each host, possibly causing the host state to change. Algorithm **hostStateTrans**(S, G) (shown in figure 6) computes the final state of a host, given an initial state S and a set of global conditions G propagated to it. The algorithm repeatedly applies the rule set R to the newly generated conditions C_0 until no more new conditions can be generated. The set of new global conditions (C_g) is also returned. Figure 7 is the algorithm for the attack simulation. The algorithm continues as long as there are still new global conditions generated by the hosts.

Let K be the number of predicates defined in the system, A_m be the maximum arity of all predicates, and D_m be the maximum cardinality of all argument domains. The number of different conditions that may enter G is bounded by $KD_m^{A_m}$ (conjunction conditions are decomposed before entering the condition set). Thus the algorithm will en-

Input: host transition rule set R_i for host H_i,
initial global condition G_0,
a host access policy *HACL*

1. Construct initial network state
   $S_i = (H_i, R_i, \emptyset)$; $G \leftarrow G_0$; $G'_0 = \emptyset$
2. For each i,
   $(S'_i, C_g) =$ **hostStateTrans**(S_i, G_0);
   $S_i \leftarrow S'_i$; $G'_0 \leftarrow C_g \cup G'_0$
3. $G'_0 \leftarrow G'_0 \setminus G$
4. $G_0 =$ **filter**$(HACL, G'_0)$
5. If $G_0 = \emptyset$, terminate.
   else $G \leftarrow G_0 \cup G$; goto step 2

Figure 7: Attack simulation algorithm

ter step 2 at most $KD_m^{A_m}$ times before terminating. By the same argument, algorithm **hostStateTrans** also will loop at most $KD_m^{A_m}$ times before terminating. So the complexity of the attack simulation algorithm is bounded by $p \cdot n \cdot (KD_m^{A_m})^2$, where n is the number of hosts and p is the time complexity for computing function **apply**(R, C). Both K and A_m are constants independent of the configuration size (number of hosts and data entities). D_m is at most linear in the size of configuration. For p, our current implementation uses sequential search so it is quadratic in the size of R and C. We believe an efficient implementation using hash tables can achieve near constant time. In either case, the time complexity of the attack simulation algorithm is polynomial in the size of network configuration.

Simulating client-side attacks An attack can be targeted to a server, like the `apache` program in the example. It can also be targeted to a client, like the `projectManager` who opens the Trojan horse Acrobat Reader. For a client-side attack, the attacker creates a situation such that if the client performs a certain operation he will be compromised. In the example, the attacker modifies the executables on the `fileServer` such that whoever invokes them will become a victim.

To reason about client-side attacks, we use the *clientCompromise condition*, $\triangleright C$, to describe the "trap" situation the attacker creates. Intuitively it means if a client performs an operation that leads to condition C, he will be compromised. The client-side attack in the example can be expressed as the following trap condition on `fileServer`:

$$\triangleright(\texttt{access read sharedBinary})$$

Anyone who retrieves the contaminated binaries may be compromised.

To propagate client-side attacks, function **apply** matches a trap condition against the *post* condition of the rules. The application of rule $Pre \Longrightarrow Post$ to condition $\triangleright Post$ will generate a new condition $\triangleright Pre$, because if condition *Post* can make a client compromised, condition *Pre* will do as well.

5.3 Putting everything together

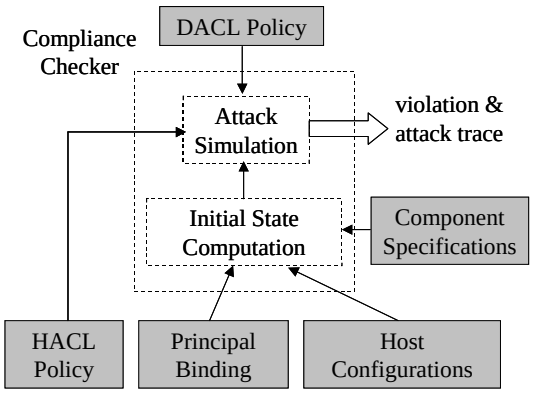

Figure 8: Compliance Checker

Figure 8 shows the complete structure of the checker. It reads in the network configuration descriptions — including a low-level HACL policy, principal bindings and host configurations, and checks them against a high-level DACL policy, with the help of expert knowledge encoded as component specifications. The checker is split into two stages. The first stage computes an initial network state according to the configuration description. This basically involves generating transition rules for every host and providing an initial set of global conditions. In generating host rule sets, the general component specifications are instantiated according to the particular settings on the host. Besides the rules from software and data modules, there are also rules generated from the principal binding information. For each host a principal operates from, there

will be a rule that states if the host is compromised, the principal's credential will be obtained by the attacker. If the principal is also `incompetent` there will be a rule stating that if the attacker can create a trap condition on the host the principal will be compromised.

Theoretically, the initial network state could includes hosts with non-empty conditions. To make our simulation algorithm concise we have required that in the initial state all hosts have empty conditions. We introduce a global condition `inject H C` to "inject" an initial condition `C` to host `H`. The initial set of global conditions is a collection of

```
inject H (net out
              anyHost anyProtocol exploit)
```

for every host `H` where a malicious principal operates.

After the initial network state is computed, stage two of the checker runs the attack simulation algorithm to find the final state of the system. Every `access` condition in the final state is examined to detect any violation of the DACL policy.

Definition 3 *An attack trace is a DAG with a single source and a single sink. Each node in the DAG is a pair (H_i, C_i) with H_i being a host and C_i being a condition. The host in the source node is operated by a malicious principal.*

We use $\delta(C)$ to denote the set of conditions gotten by decomposing all conjunctions (&) in C. A trace is valid with respect to a set of host rules and a HACL policy if for every nonsource node (H, C), either one of the following holds

1. The node has only one predecessor (H', C'). $H \neq H'$ and

   $C \in$ **condLocal**$(H,$ **filter**$($`HACL`, **condGlobal**$(H', (C'; \texttt{nil}))))$

   or

2. The node has one or more predecessors (H, C_i). Host H has a rule $D \Longrightarrow E$ and $\forall c \in \delta(D)$ there is a C_i that satisfies c. And $C \in \delta(E)$.

Informally, case 1 is when a condition is propagated from another host and case 2 is when a condition is gotten by satisfying all the preconditions of a rule on the host. An attack trace for the example can be found in figure 9. The direction of the edges is from bottom to top.

Theorem 1 (Completeness) *For any attack trace that is valid with respect to the HACL policy and initial host rules, if the sink of the trace is (H, C), then condition C is true at host H in the final state.*

6 Discussion

In compliance checking, we use a polynomial attack simulation algorithm to search all possible attack scenarios. Another approach is to use a standard model checker to do the search. It is well known that the complexity of model-checking is inherently exponential in the size of the state-transition system. Our algorithm can achieve polynomial time due to an implicit assumption of *monotonicity* [1, 14]. Under the monotonicity assumption, an attacker does not need to relinquish the access he has got in order to gain more accesses, thus no backtracking is needed during search. The monotonicity assumption is generally true if the policy only involves confidentiality and integrity. If one also wants to reason about availability the assumption will no longer hold — a denial of service attack not only compromises the availability of a host, but also compromises the ability of the attacker to launch further attacks from that host. In that case backtracking would be necessary and we suspect an off-the-shelf model checker may outperform a custom-tailored search engine.

In reality trust management (TM) plays an important role in network security [5]. The TM policies themselves can be quite complex and the analysis of their security properties is hard [13]. We simplified the modeling of trust relationships by ignoring the possible relationships between different principals' credentials. After the attacker obtains one principal's credential, our model does not infer its effects on trust relationships other than that the attacker can gain privileges of the victim principal.

7 Related Work

Policy-based network management has been studied extensively in the past ten years [8, 9, 2, 15, 7,

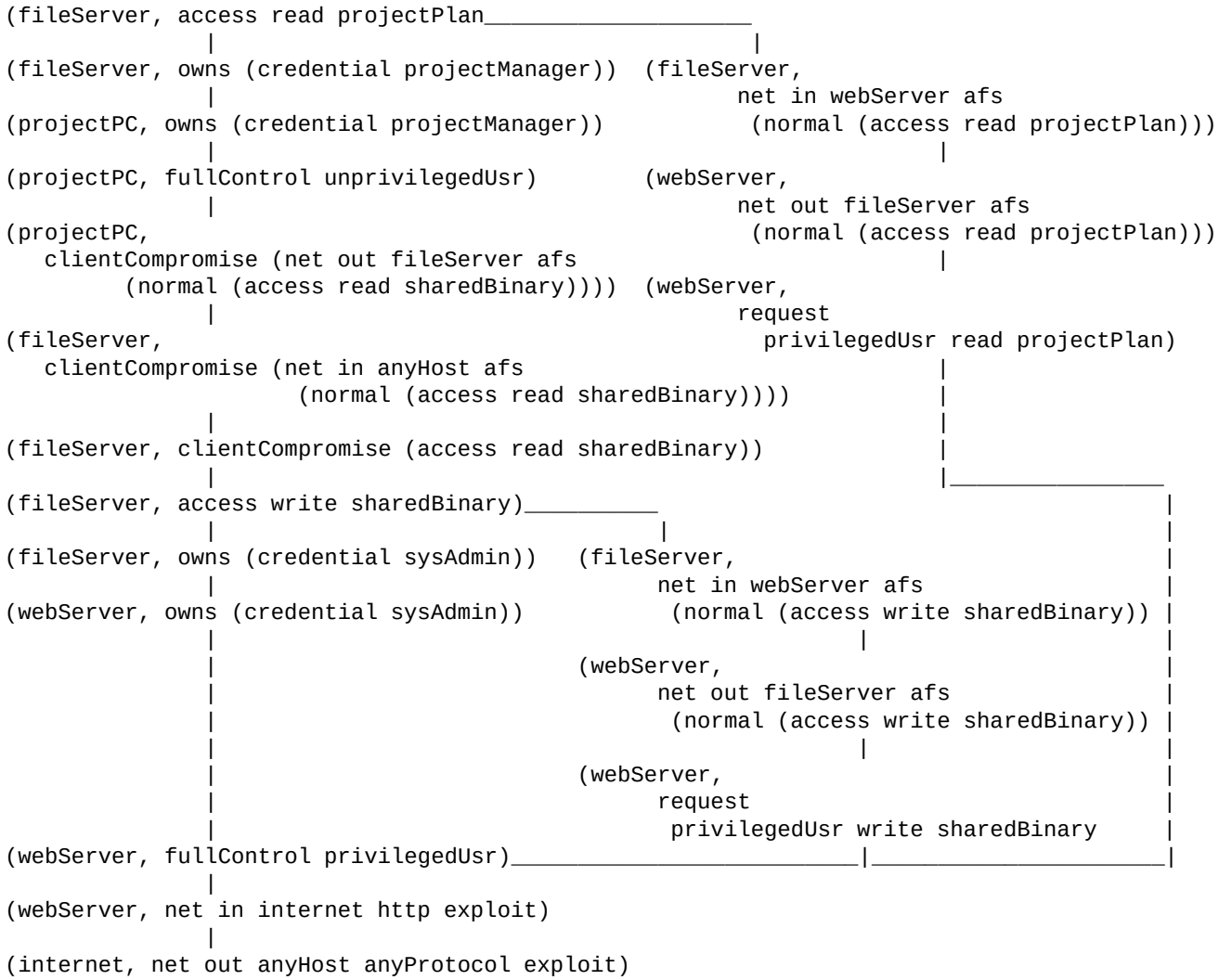

Figure 9: An example attack trace

6]. Recent work by Burns et al. [6] has yielded a tool that given a security policy specified as HACL, automatically checks that the configuration of a network with multiple firewalls upholds the policy. The tool can also generate a complete configuration that satisfies a given policy from a partial configuration. A HACL policy only deals with packet-filtering devices and its correctness cannot be easily verified without reasoning about the interaction with other parts of the network.

In the STRONGMAN project [11], multiple high-level policy languages are used to specify security requirements for different application domains. Those high-level policies are compiled into a common intermediate policy language KEYNOTE [4], where different applications' policies can be composed. The decoupling of high and low level policies and policy composition provide better modularity and extensibility across application domains. Our concern is different. By specifying a policy at a level closer to policy maker's intention, we hope to increase security assurance of the system. We also account for possible software vulnerabilities in the compliance checking.

Recent works have applied model checking techniques in analyzing network security vulnerabilities caused by combinations of exploits [16, 17, 18]. Our algorithm assumes the monotonicity of attacks and thus avoids exponential complexity. However, one could make a more expressive policy language using the power of temporal logic, especially the ability to use universal quantifiers. It is possible that the OBDD techniques in model checking would deal with quantified formulas more efficiently than expanding the formula on the quantified domain.

Topological Vulnerability Analysis (TVA) [10] combines an exploit knowledge base with a local network vulnerability scanner to analyze exploit sequences leading to attack goals. Our framework is similar to theirs in that general component specifications are combined with configuration situations at local site in checking the compliance of a high-level policy. Since TVA does not explicitly specify the security policy to enforce, it cannot provide information as to whether a potential attack sequence is really harmful. Also, our framework is aimed at the analysis of complete network configurations, including firewalls and simple trust relationships.

8 Conclusions

We have proposed a policy-based approach to automatic network security management. The approach has the following characteristics.

1. A high-level policy expresses security concerns of protecting data. The policy language is simple to understand and easy to get right.

2. Potential software vulnerabilities are taken into consideration in compliance checking. Our tool can incorporate knowledge of software bugs from an independent outside source.

3. The two-level framework allows for leveraging existing management tools based on low-level policies and provides good modularity.

We have implemented the compliance checker in a prolog-style logic program.

References

[1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[2] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[3] S. Bhatt, A.V. Konstantinou, S. R. Rajagopalan, and Yechiam Yemini. Managing security in dynamic networks. In *13th USENIX Systems Administration Conference (LISA'99)*, Seattle, WA, USA, November 1999.

[4] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *The KeyNote Trust-Management System, Version 2*, Sept 1999. Request For Comments (RFC) 2704.

[5] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, Springer-Verlag Lecture Notes in Computer Science State-of-the-Art series, pages 185 – 210, Berlin, 1999.

[6] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, and et al. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, Anaheim, California, June 2001.

[7] N. Damianou, N. Dulay, and M Sloman E. Lupu. The ponder policy specification language. In *Workshop on Policies for Distributed Systems and Networks (Policy2001)*, HP Labs Bristol, Jan 2001.

[8] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 120–129, Oakland, CA, 1997.

[9] Susan Hinrichs. Policy-based management: Bridging the gap. In *15th Annual Computer Security Applications Conference*, Phoenix, Arizona, Dec 1999.

[10] Sushil Jajodia, Steven Noel, and Brian O'Berry. *Topological Analysis of Network Attack Vulnerabiity*, chapter 5. Kluwer Academic Publisher, 2003.

[11] Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald, and Jonathan M. Smith. The STRONGMAN architecture. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 178 – 188, Washington, DC, April 2003.

[12] Alexander V. Konstantinou, Yechiam Yemini, and Danilo Florissi. Towards self-configuring networks. In *DARPA Active Networks Conference and Exposition (DANCE)*, San Franscisco, CA, May 2002.

[13] Ninghui Li, William H. Winsborough, and John C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *2003 IEEE Symposium on Security and Privacy*, Berkeley, California, May 2003.

[14] Steven Noel, Sushil Jajodia, Brian O'Berry, and Michael Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *19th Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[15] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies and complex constraints. In *Network and Distributed System Security Symposium (NDSS)*, Feb 2001.

[16] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *2000 IEEE Symposium on Security and Privacy*, pages 156–165, 2000.

[17] Ronald W. Ritchey, Brian O'Berry, and Steven Noel. Representing TCP/IP connectivity for topological analysis of network security. In *19th Annual Computer Security Applications Conference (ACSAC)*, December 2002.

[18] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.