MulVAL: A Logic-based Network Security Analyzer *

Xinming Ou Sudhakar Govindavajhala Andrew W. Appel
Princeton University
{*xou, sudhakar, appel*}*@cs.princeton.edu*

Abstract

To determine the security impact software vulnerabilities have on a particular network, one must consider interactions among multiple network elements. For a vulnerability analysis tool to be useful in practice, two features are crucial. First, the model used in the analysis must be able to automatically integrate formal vulnerability specifications from the bug-reporting community. Second, the analysis must be able to scale to networks with thousands of machines.

We show how to achieve these two goals by presenting MulVAL, an end-to-end framework and reasoning system that conducts multihost, multistage vulnerability analysis on a network. MulVAL adopts Datalog as the modeling language for the elements in the analysis (bug specification, configuration description, reasoning rules, operating-system permission and privilege model, etc.). We easily leverage existing vulnerability-database and scanning tools by expressing their output in Datalog and feeding it to our MulVAL reasoning engine. Once the information is collected, the analysis can be performed in seconds for networks with thousands of machines.

We implemented our framework on the Red Hat Linux platform. Our framework can reason about 84% of the Red Hat bugs reported in OVAL, a formal vulnerability definition language. We tested our tool on a real network with hundreds of users. The tool detected a policy violation caused by software vulnerabilities and the system administrators took remediation measures.

1 Introduction

Dealing with software vulnerabilities on network hosts poses a great challenge to network administration. With the number of vulnerabilities discovered each year growing rapidly, it is impossible for system administrators to keep the software running on their network machines free of security bugs. One of a sysadmin's daily chores is to read bug reports from various sources (such as CERT, BugTraq etc.) and understand which reported bugs are actually security vulnerabilities in the context of his own network. In the wake of new vulnerabilities, assessment of their security impact on the network is important in choosing the right countermeasures: patch and reboot, reconfigure a firewall, dismount a file-server partition, and so on.

A vulnerability analysis tool can be useful to such a sysadmin, but only if it can automatically integrate formal vulnerability specifications from the bug-reporting community, and only if the analysis can scale to networks with thousands of machines. These two issues have not been addressed by the previous work in this area.

We present MulVAL (Multihost, multistage Vulnerability Analysis), a framework for modeling the interaction of software bugs with system and network configurations. MulVAL uses Datalog as its modeling language. The information in the vulnerability database provided by the bug-reporting community, the configuration information of each machine and the network, and other relevant information are all encoded as Datalog facts. The reasoning engine consists of a collection of Datalog rules that captures the operating system behavior and the interaction of various components in the network. Thus integrating information from the bug-reporting community and off-the-shelf scanning tools in the reasoning model is straightforward. The reasoning engine in MulVAL scales well with the size of the network. Once all the information is collected, the analysis can be performed in seconds for networks with thousands of machines.

The inputs to MulVAL's analysis are, **Advisories**: What vulnerabilities have been reported and do they exist on my machines? **Host configuration**: What software and services are running on my hosts, and how are they configured? **Network configuration**: How are my network routers and firewalls configured? **Principals**: Who are

*This research was supported in part by DARPA award F30602-99-1-0519 and by ARDA award NBCHC030106. This information does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.
To appear in 14th Usenix Security Symposium, August 2005.

the users of my network? **Interaction**: What is the model of how all these components interact? **Policy**: What accesses do I want to permit?

In the next section, we give examples of the Datalog clauses for each of these elements and the tools that can be leveraged to gather the information.

2 Representation

MulVAL comprises a scanner—run asynchronously on each host and which adapts existing tools such as OVAL to a great extent—and an analyzer, run on one host whenever new information arrives from the scanners.

Advisories. Recently, the *Open Vulnerability Assessment Language* [26] (OVAL) has been developed that formalizes how to recognize the presence of vulnerabilities on computer systems. An OVAL scanner takes such formalized vulnerability definitions and tests a machine for vulnerable software. We convert the result of the test into Datalog clauses like the following:

```
vulExists(webServer, 'CAN-2002-0392', httpd).
```

Namely, the scanner identified a vulnerability with CVE [1] ID `CAN-2002-0392` on machine `webServer`. The vulnerability involved the server program `httpd`. However, the effect of the vulnerability — how it can be exploited and what is the consequence — is not formalized in OVAL. ICAT [18], a vulnerability database developed by the National Institute of Standards and Technology, provides the information about a vulnerability's effect. We convert the relevant information in ICAT into Datalog clauses such as

```
vulProperty('CAN-2002-0392', remoteExploit,
              privilegeEscalation).
```

The vulnerability enables a remote attacker to execute arbitrary code with all the program's privileges.

Host configuration. An OVAL scanner can be directed to extract configuration parameters on a host. For example, it can output the information of a service program (port number, privilege, etc). We convert the output to Datalog clauses like

```
networkService(webServer, httpd,
                   TCP, 80, apache).
```

That is, program `httpd` runs on machine `webServer` as user `apache`, and listens on port `80` using `TCP` protocol.

Network configuration. MulVAL models network (router and firewalls) configurations as abstract host access-control lists (HACL). This information can be provided by a firewall management tool such as the Smart Firewall [4]. Here is an example HACL entry that allows `TCP` traffic to flow from `internet` to port `80` on `webServer`:

```
hacl(internet, webServer, TCP, 80).
```

Principals. Principal binding maps a principal symbol to its user accounts on network hosts. The administrator should define the principal binding like:

```
hasAccount(user, projectPC, userAccount).
hasAccount(sysAdmin, webServer, root).
```

Interaction. In a multistage attack, the semantics of the vulnerability and the operating system determine an adversary's options in each stage. We encode these as Horn clauses (i.e., Prolog), where the first line is the conclusion and the remaining lines are the enabling conditions. For example,

```
execCode(Attacker, Host, Priv) :-
  vulExists(Host, VulID, Program),
  vulProperty(VulID, remoteExploit,
              privEscalation),
  networkService(Host, Program,
                 Protocol, Port, Priv),
  netAccess(Attacker, Host, Protocol, Port),
  malicious(Attacker).
```

That is, if `Program` running on `Host` contains a remotely exploitable vulnerability whose impact is privilege escalation, the buggy program is running under privilege `Priv` and listening on `Protocol` and `Port`, and the attacker can access the service through the network, then the attacker can execute arbitrary code on the machine under `Priv`. This rule can be applied to any vulnerability that matches the pattern.

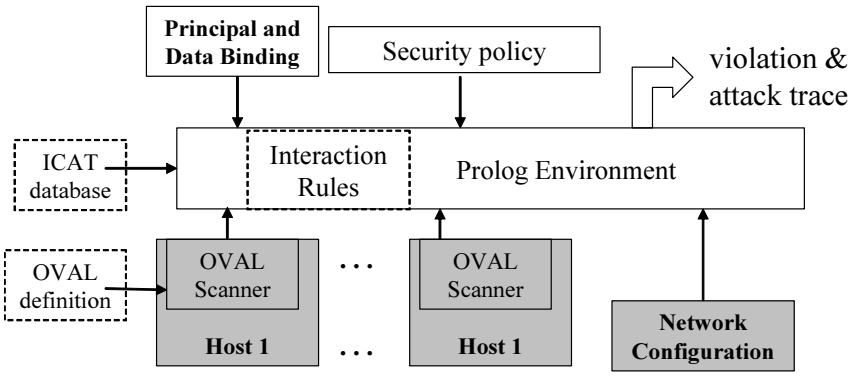

Figure 1: The MulVAL framework

Policy. In MulVAL, a policy describes which principal can have what access to data. Anything not explicitly allowed is prohibited. Following is a sample policy.

```
allow(Everyone, read, webPages).
allow(systemAdmin, write, webPages).
```

Because `Everyone` is capitalized, it is a Prolog variable, so it can match any user.

Analysis framework. Since Datalog is a subset of Prolog, the encoded information can be directly loaded into a Prolog environment and executed. We use the XSB [22] environment because it supports tabled execution of Prolog programs. Tabling is a form of dynamic programming that avoids recomputation of previously calculated facts. Also, tabling provides complete declarative-style logic programing because the order of rules does not affect the result of the execution. The framework is shown in Figure 1. An OVAL scanner runs on each machine and outputs vulnerability report and relevant configuration parameters. The tuples from the scanners, the network configuration (represented as HACL), the deduction rules, and the administrator-defined security policy are loaded into an XSB environment. A Prolog query (see section 5.2) can then be made to search for policy violations. Our program can also generate a detailed attack tree.

The rest of the paper describes in detail the various components of MulVAL. Section 3 briefly introduces the formal vulnerability definitions from bug-reporting communities and how they are integrated into MulVAL. Section 4 discusses the reasoning and input Datalog clauses used in MulVAL and the analysis algorithm. Section 5 shows two examples that illustrate the analysis process. Section 6 discusses how to conduct hypothetical vulnerability analysis in MulVAL. Performance data is shown in section 7. Some design and implementation issues are discussed in section 8. We compare our approach with some related work in section 9 and conclude in section 10.

3 Vulnerability specification

A specification of a security bug consists of two parts: how to recognize the existence of the bug on a system, and what is the effect of the bug on a system. The recognition specification is only used in the scanning of a machine, whereas the effect specification is used in the reasoning process. Recently, the bug-reporting community has started to provide these kinds of information in formal, machine-readable formats. In the next two subsections, we briefly describe OVAL, a formal specification language for recognizing vulnerabilities, and ICAT, a database that provides a vulnerability's effect.

3.1 The OVAL language and scanner

The Open Vulnerability Assessment Language (OVAL) [26] is an XML-based language for specifying machine configuration tests. When a new software vulnerability is discovered, an OVAL definition can specify how to check a machine for its existence. Then the OVAL definition can be fed to an OVAL-compatible scanner, which will conduct the specified tests and report the result. Currently, OVAL vulnerability definitions are available for the Windows, Red Hat Linux and Solaris platforms. OVAL-compliant scanners are available for Windows and Red Hat Linux platforms. OVAL vulnerability

definitions have been created since 2002 and new definitions are being submitted and reviewed on a daily basis. As of January 31, 2005, the number of OVAL definitions for each platform is:

Platform	Submitted	Accepted
Microsoft Windows	543	489
Red Hat Linux	203	202
Sun Solaris	73	57
Total	819	748

For example, we ran the OVAL scanner on one machine using the latest OVAL definition file and found the following vulnerabilities:

```
VULNERABILITIES FOUND:
OVAL Id     CVE Id
-------------------------
OVAL2819    CAN-2004-0427
OVAL2915    CAN-2004-0554
OVAL2961    CAN-2004-0495
OVAL3657    CVE-2002-1363
-------------------------
```

We convert the output of an OVAL scanner into Datalog clauses like the following:

```
vulExists(webServer, 'CVE-2002-0392', httpd).
```

Besides producing a list of discovered vulnerabilities, the OVAL scanner can also output a detailed machine configuration information in the System Characteristics Schema. Some of this information is useful for reasoning about multistage attacks. For example, the protocol and port number a service program is listening on, in combination with the firewall rules and network topology expressed as HACL, helps determine whether an attacker can send a malicious packet to a vulnerable program. Currently the following predicates about machine configurations are used in the reasoning engine.

```
networkService(Host, Program,
                    Protocol, Port, Priv).
clientProgram(Host, Program, Priv).
setuidProgram(Host, Program, Owner).
filePath(H, Owner, Path).
nfsExport(Server, Path, Access, Client).
nfsMountTable(Client, ClientPath,
              Server, ServerPath).
```

`networkService` describes the port number and protocol under which a service program is listening and the user privilege the program has on the machine. If the same server is listening under multiple ports and protocols, this is described by multiple `networkService` statements. `clientProgram` describes the privilege of a client program once it gets executed. `setuidProgram` specifies an a setuid executable on the system and its owner. `filePath` specifies the owner of a particular path in the file system. `nfsExport` describes which portion of the file system on an NFS server is exported to a client. `nfsMountTable` describes an NFS mounting table entry on the client machine. The scanner used in MulVAL is implemented by augmenting a standard off-the-shelf OVAL scanner, such that it not only reports the existence of vulnerabilities, but also outputs machine configuration information in the form of these predicates.

3.2 Vulnerability effect

One can find detailed information about the vulnerabilities from OVAL's web site[2]. For example, the OVAL description for the bug OVAL2961 is:

> Multiple unknown vulnerabilities in Linux kernel 2.4 and 2.6 allow local users to gain privileges or access kernel memory, ...

This informal short description highlights the effect of the vulnerability — how the vulnerability can be exploited and the consequence it can cause. If a machine-readable database were to provide information on the effect of a bug such as *bug 2961 is only locally exploitable*, one could formally prove properties like *if all local users are trusted, then the network is safe from remote attacker*. Unfortunately, OVAL does not present the information about the effect of a vulnerability in a machine readable format. Fortunately, the ICAT database [18] classifies the effect of a vulnerability in two dimensions: exploitable range and consequences.

- exploitable range: *local*, *remote*
- consequence: *confidentiality loss*, *integrity loss*, *denial of service*, and *privilege escalation*

A *local* exploit requires that the attacker already have some local access on the host. A *remote* exploit does not have this requirement. Two most common exploit consequences are *privilege escalation* and *denial of service*. Currently all OVAL definitions have corresponding ICAT entries (the two can be cross-referenced by CVE Id). It would be nice if OVAL and ICAT be merged into a single database that provides both information.

We converted the above classification in the ICAT database into Datalog clauses such as

```
vulProperty('CVE-2004-00495',
          localExploit, privEscalation).
```

4 The MulVAL Reasoning System

The reasoning rules in MulVAL are declared as Datalog clauses. A *literal*, $p(t_1, \ldots, t_k)$ is a predicate applied to its arguments, each of which is either a constant or a variable. In the formalism of Datalog, a variable is an identifier that starts with an upper-case letter. A constant is one that starts with a lower-case letter. Let $L_0, \ldots, L_n$ be literals, a sentence in MulVAL is represented as a Horn clause:

$$L_0 \texttt{ :- } L_1, \ldots, L_n$$

Semantically, it means if $L_1, \ldots, L_n$ are true then L_0 is also true. The left-hand side is called the *head* and the right-hand side is called the *body*. A clause with an empty body is called a *fact*. A clause with a nonempty body is called a *rule*.

4.1 Reasoning rules

MulVAL reasoning rules specify semantics of different kinds of exploits, compromise propagation, and multi-hop network access. The MulVAL rules are carefully designed so that information about specific vulnerabilities are factored out into the data generated from OVAL and ICAT. The interaction rules characterize general attack methodologies (such as "Trojan Horse client program"), not specific vulnerabilities. Thus the rules do not need to be changed frequently, even if new vulnerabilities are reported frequently.

4.1.1 Exploit rules

We introduce several predicates that are used in the exploit rules. `execCode(P,H,UserPriv)` indicates that principal `P` can execute arbitrary code with privilege `UserPriv` on machine `H`. `netAccess(P, H, Protocol, Port)` indicates principal `P` can send packets to `Port` on machine `H` through `Protocol`.

The effect classification of a vulnerability indicates how it can be exploited and what is the consequence. We have already seen a rule for remote exploit of a service program in section 2. Following is the exploit rule for remote exploit of a client program.

```
execCode(Attacker, Host, Priv) :-
  vulExists(Host, VulID, Program),
  vulProperty(VulID, remoteExploit,
              privEscalation),
  clientProgram(Host, Program, Priv),
  malicious(Attacker).
```

The body of the rule specifies that 1) the `Program` is vulnerable to a remote exploit; 2) the `Program` is client software with privilege `Priv`[3]; 3) the `Attacker` is some principal that originates from a part of the network where malicious users may exist. The consequence of the exploit is that the attacker can execute arbitrary code with privilege `Priv`.

The rule for the exploit of a local privilege escalation vulnerability is as follows:

```
execCode(Attacker, Host, Owner) :-
   vulExists(Host, VulID, Prog),
   vulProperty(VulID, localExploit,
               privEscalation),
   setuidProgram(Host, Prog, Owner),
   execCode(Attacker, Host, SomePriv),
   malicious(Attacker).
```

For this exploit, the precondition `execCode` requires that an attacker first have some access to the machine `Host`. The consequence of the exploit is that the attacker can gain privilege of the owner of a setuid program.

In our model, the Linux kernel is both a network service running as `root`, and a setuid program owned by `root`. That is, the consequence of exploiting a privilege-escalation bug in kernel (either local or remote) will result in a root compromise.

Currently we do not have exploit rules for vulnerabilities whose exploit consequence is confidentiality loss or integrity loss. The ICAT database does not provide precise information as to what confidential information may be leaked to an attacker and what information on the system may be modified by an attacker. ICAT statistics shows that 84% of vulnerabilities are labeled with privilege-escalation or only labeled with denial-of-service, the two kinds of exploits modeled in MulVAL. It seems in reality privilege-escalation bugs are the most common target for exploit in a multistage attack.

4.1.2 Compromise propagation

One of the important features of MulVAL is the ability to reason about multistage attacks. After an exploit is successfully applied, the reasoning engine must discover how the attacker can further compromise a system.

For example, the following rule says if an attacker `P` can access machine `H` with `Owner`'s privilege, then he can have arbitrary access to files owned by `Owner`.

```
accessFile(P, H, Access, Path) :-
  execCode(P, H, Owner),
  filePath(H, Owner, Path).
```

On the other hand, if an attacker can modify files under `Owner`'s directory, he can gain privilege of `Owner`. That is because a Trojan horse can be injected by modified execution binaries, which `Owner` might then execute:

```
execCode(Attacker, H, Owner) :-
        accessFile(Attacker, H, write, Path),
        filePath(H, Owner, Path),
        malicious(Attacker).
```

Network file systems Some multistage attacks also exploit normal software behaviors. For example, through talking to system administrators we found that the NFS file-sharing system is widely used in many organizations and has contributed to many intrusions. One scenario is that an attacker gets `root` access on a machine that can talk to an NFS server. Depending on the file server's configuration, the attacker may be able to access any file on the server.

```
accessFile(P, Server, Access, Path) :-
   malicious(P),
   execCode(P, Client, root),
   nfsExportInfo(Server, Path, Access, Client),
   hacl(Client, Server, rpc, 100003)),
```

`hacl(Client, Server, rpc, 100003)` is an entry in *host access control list* (section 4.2), which specifies machine `Client` can talk to `Server` through NFS, an RPC (remote procedure call) protocol with number `100003`.

4.1.3 Multihop network access

```
netAccess(P, H2, Protocol, Port) :-
    execCode(P, H1, Priv),
    hacl(H1, H2, Protocol, Port).
```

If a principal `P` has access to machine `H1` under some privilege and the network allows `H1` to access `H2` through `Protocol` and `Port`, then the principal can access host `H2` through the protocol and port. This allows for reasoning about multihost attacks, where an attacker first gains access on one machine inside a network and launches an attack from there. Predicate `hacl` stands for an entry in the host access control list (HACL).

4.2 Host Access Control List

A host access control list specifies all accesses between hosts that are allowed by the network. It consists of a collection of entries of the following form:

```
hacl(Source, Destination, Protocol, DestPort).
```

Packet flow is controlled by firewalls, routers, switches, and other aspects of network topology. HACL is an abstraction of the effects of the configuration of these elements. In dynamic environments involving the use of Dynamic Host Configuration Protocol (especially in wireless networks), firewall rules can be very complex and can be affected by the status of the network, the ability of users to authenticate to a central authentication server, etc. In such environments, it is infeasible to ask the system administrator to manually provide all HACL rules. We envision that an automatic tool such as the Smart Firewall [4] can provide the HACL list automatically for our analysis.

4.3 Policy specification

The security policy specifies which principal can access what data. Each principal and data is given a symbolic name, which is mapped to a concrete entity by the binding information discussed in section 4.4. Each policy statement is of the form

```
allow(Principal, Access, Data).
```

The arguments can be either constants or variables (variables start with a capital letter and can match any constant). Following is an example policy:

```
allow(Everyone, read, webPages).
allow(user, Access, projectPlan).
allow(sysAdmin, Access, Data).
```

The policy says anybody can read `webPages`, `user` can have arbitrary access to `projectPlan`. And `sysAdmin` can have arbitrary access to arbitrary data. Anything not explicitly allowed is prohibited.

The policy language presented in this section is quite simple and easy to make right. However, the MulVAL reasoning system can handle more complex policies as well (see section 4.6).

4.4 Binding information

Principal binding maps a principal symbol to its user accounts on network hosts. For example:

```
hasAccount(user, projectPC, userAccount).
hasAccount(sysAdmin, webServer, root).
```

Data binding maps a data symbol to a path on a machine. For example:

```
dataBind(projectPlan,workstation,'/home').
dataBind(webPages, webServer, '/www').
```

The binding information is provided manually.

4.5 Algorithm

The analysis algorithm is divided into two phases: *attack simulation* and *policy checking*. In the attack simulation phase, all possible data accesses that can result from multistage, multihost attacks are derived. This is achieved by the following Datalog program.

```
access(P, Access, Data) :-
  dataBind(Data, H, Path),
  accessFile(P, H, Access, Path).
```

That is, if `Data` is stored on machine `H` under path `Path`, and principal `P` can access files under the path, then `P` can access `Data`. The attack simulation happens in the derivation of `accessFile`, which involves the Datalog interaction rules and data tuple inputs from various components of MulVAL. For a Datalog program, there are at most polynomial number of facts that can be derived. Since XSB's tabling mechanism guarantees each fact is computed only once, the attack simulation phase is polynomial.

In the policy checking phase, the data access tuples output from the attack simulation phase are compared with the given security policy. If an access is not allowed by the policy, a violation is detected. The following Prolog program performs policy checking.

```
policyViolation(P, Access, Data) :-
  access(P, Access, Data),
  not allow(P, Access, Data).
```

This is not a pure Datalog program because it uses negation. But the use of negation in this program has a well-founded semantics [10]. The complexity of a Datalog program with well-founded negation is polynomial in the size of input [6]. In practice the policy checking algorithm runs very efficiently in XSB (see section 7).

4.6 More complex policies

The two-phase separation in the MulVAL algorithm allows us to use richer policy languages than Datalog without affecting the complexity of the attack simulation phase. The MulVAL reasoning system supports general Prolog as the policy language. Should one need even richer policy specification, the attack simulation can still be performed efficiently and the resulting data access tuples can be sent to a policy resolver, which can handle the richer policy specification efficiently.

No policy? Because the attack simulation is *not* guided by or dependent on the security policy, it is possible to use MulVAL without a security policy; the system administrator may find useful the raw report of who can access what. However, the policy is useful in filtering undesirable accesses from harmless accesses.

5 Examples

5.1 A small real-world example

We ran our tool on a small network used by seven hundred users. We analyzed a subset of the network that contains only machines managed by the system administrators.[4] Our tool found a violation of policy because of a vulnerability. The system administrators subsequently patched the bug.

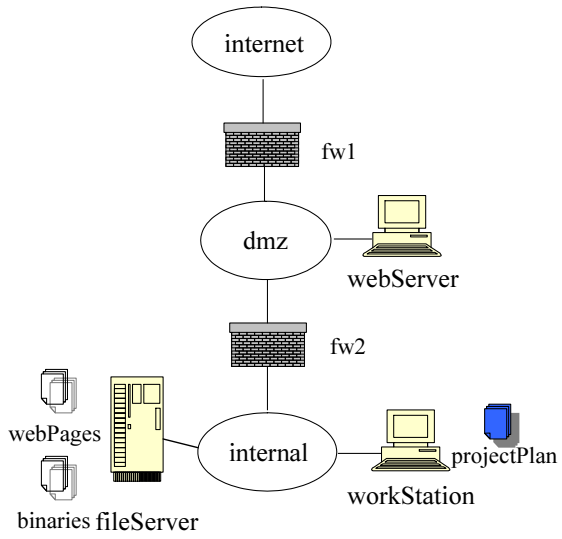

Figure 2: Example

Network topology. The topology of the network is very similar to the one in Figure 2. There are three zones (`internet`, `dmz` and `internal`) separated by two firewalls (`fw1` and `fw2`). The administrators manage the `webserver`, the `workStation` and the `fileserver`. The users have access to the public server `workStation` which they use for their computing needs. The host access control list for this network is:

```
hacl(internet, webServer, tcp, 80).
hacl(webServer, fileServer, rpc, 100003).
hacl(webServer, fileServer, rpc, 100005).
hacl(fileServer, AnyHost,
                 AnyProtocol, AnyPort).
hacl(workStation, AnyHost,
                 AnyProtocol, AnyPort).
hacl(H, H, AnyProtocol, AnyPort).
```

Machine configuration The following Datalog tuples describe the configuration information of the three machines.

```
networkService(webServer , httpd,
                    tcp , 80 , apache).
nfsMount(webServer, '/www',
             fileServer, '/export/www').

networkService(fileServer, nfsd,
                    rpc, 100003, root).
networkService(fileServer, mountd,
                    rpc, 100005, root).
nfsExport(fileServer, '/export/share',
                    read, workStation).
nfsExport(fileServer, '/export/www',
                    read, webServer).
nfsMount(workStation, '/usr/local/share',
         fileServer, '/export/share').
```

The `fileServer` serves files for the `webServer` and the `workStation` through the NFS protocol. There are actually many machines represented by `workStation`. They are managed by the administrators and run the same software configuration. To avoid the hassle of installing each application on each of the machines separately, the administrators maintain a collection of application binaries under `/export/share` on `fileServer` so that any change like recompilation of an application program needs to be done only once. These binaries are exported through NFS to the `workStation`. The directory `/export/www` is exported to `webServer`.

Data binding.

```
dataBind(projectplan, workStation, '/home').
dataBind(webPages, webServer, '/www').
```

Principals. The principal `sysAdmin` manages the machines with user name `root`. Since all the users are treated equally, we model one of them as principal `user`. `user` uses the `workStation` with user name `userAccount`. For this organization, the primary worry is a remote attacker launching an attack from outside the network. The attackers are modeled by a single principal `attacker` who uses the machine `internet` and has complete control of it. The Datalog tuples for principal bindings are:

```
hasAccount(user, workStation, userAccount).

hasAccount(sysAdmin, workStation, root).
hasAccount(sysAdmin, webServer, root).
hasAccount(sysAdmin, fileServer, root).

hasAccount(attacker, internet, root).
malicious(attacker).
```

Security policy The administrators need to ensure that the confidentiality and the integrity of users' files will not be compromised by an attacker. Thus the policy is

```
allow(Anyone, read, webPages).
allow(user, AnyAccess, projectPlan).
allow(sysAdmin, AnyAccess, Data).
```

Results We ran the MulVAL scanner on each of the machines. The interesting part of the output was that workStation had the following vulnerabilities:

```
vulExists(workStation, 'CAN-2004-0427', kernel).
vulExists(workStation, 'CAN-2004-0554', kernel).
vulExists(workStation, 'CAN-2004-0495', kernel).
vulExists(workStation, 'CVE-2002-1363', libpng).
```

The MulVAL reasoning engine then analyzed this output in combination with the other inputs described above. The tool did indeed find a policy violation because of the bug CVE-2002-1363 — a remotely exploitable bug in the libpng library. A reasoning rule for remote exploit derives that the workStation machine can be compromised. Thus the projectPlan data stored on it can be accessed by the attacker, violating the policy. Our system administrators subsequently patched the vulnerable libpng library.

One might be curious that there was only one vulnerability that contributed to the policy violation though the host workStation actually had four vulnerabilities. The other three bugs on the workStation are locally exploitable vulnerabilities in the kernel. Since only trusted users access these hosts, after patching the libpng bug our tool indicates the policy is no longer violated. These machines have uptimes in the order of months and upgrading the kernel would require a reboot. Patching these vulnerabilities would result in a loss of availability, which is best avoided. The administrators can meet the security goals without patching the kernel and rebooting the workStation. We expect our tool to be useful in mission-critical systems like commercial mail servers serving millions of users and servers running long computations.

5.2 An example multistage attack

We now illustrate how our framework works in the case of multistage attacks. Let us consider a simulated attack on the network discussed in the previous example. Suppose the following two vulnerabilities are reported by the scanner:

```
vulExists(webServer, 'CVE-2002-0392',
          httpd).

vulExists(fileServer, 'CAN-2003-0252',
          mountd).
```

Both vulnerabilities are remotely exploitable and can result in privilege escalation. The corresponding Datalog clauses from ICAT database are:

```
vulProperty('CVE-2002-0392',
            remoteExploit, privEscalation).

vulProperty('CAN-2003-0252',
            remoteExploit, privEscalation).
```

The machine and network configuration, principal and data binding, and the security policy are the same as in the previous example.

Results The MulVAL reasoning engine analyzed the input Datalog tuples. The Prolog session transcript is as follows:

```
| ?- policyViolation(Adversary,
                      Access, Resource).

Adversary = attacker
Access = read
Resource = projectPlan;

Adversary = attacker
Access = write
Resource = webPages;

Adversary = attacker
Access = write
Resource = projectPlan;
```

We show the trace of the first violation in Appendix A. Here we explain how the attack can lead to the policy violation.

An attacker can first compromise webServer by remotely exploiting vulnerability CVE-2002-0392 to get control of webServer. Since webServer is allowed to access fileServer, he can then compromise fileServer by exploiting vulnerability CAN-2003-0252 and become root on the server. Next he can modify arbitrary files on fileServer. Since the executable binaries on workStation are mounted on fileServer, their integrity will be compromised by the attacker. Eventually an innocent user will execute the compromised client program; this will give the attacker access to workStation. Thus the files stored on it would also be compromised.

One way to fix this violation is moving webPages to webServer and blocking inbound access from dmz zone to internal zone. After incorporating these counter measures, we ran MulVAL reasoning engine on the new inputs and verified that the security policy is satisfied.

6 Hypothetical analysis

One important usage of vulnerability reasoning tools is to conduct "what if" analysis. For example, the administrator would like to ask *"Will my network still be secure if two CERT advisories arrive tomorrow?"*. After all, an important purpose of using firewalls is to guard against *potential* threats. Even there is no known vulnerability in the network today, one might be discovered tomorrow. Analysis that can reveal weaknesses in the network under hypothetical circumstances is useful in improving security. Performing this kind of hypothetical analysis is easy in our framework. We introduce a predicate `bugHyp` to represent hypothetical software vulnerabilities. For example, following is a hypothetical bug in the web service program `httpd` on host `webServer`.

```
bugHyp(webServer, httpd,
               remoteExploit, privEscalation).
```

The fake bugs are then introduced into the reasoning process.

```
vulExists(Host, VulID, Prog) :-
   bugHyp(Host, Prog, Range, Consequence).

vulProperty(VulID, Range, Consequence) :-
   bugHyp(Host, Prog, Range, Consequence).
```

The following Prolog program will determine whether a policy violation will happen with two arbitrary hypothetical bugs.

```
checktwo(P, Acc, Data, Prog1, Prog2) :-
  program(Prog1),
  program(Prog2),
  Prog1 @< Prog2,
  cleanState,
  assert(bugHyp(H1, Prog1, Range1, Conseq1)),
  assert(bugHyp(H2, Prog2, Range2, Conseq2)),
  policyViolation(P, Acc, Data).
```

The two `assert` statements introduce dynamic clauses about hypothetical bugs in two programs (Prolog backtracking will cycle through all possible combination of two programs.). The policy check is conducted with the existence of the dynamic clauses. If no policy violation is found, the execution will back track and another two hypothetical bugs (in different two programs) will be tried. `@<` is the term comparison operator in Prolog. It ensures a combination of two programs is tried only once. If there exist two programs whose hypothetical bugs will break the security policy of the network, the violation will be reported by `checktwo`. Otherwise the network can withstand two hypothetical bugs.

7 Performance and Scalability

We measured the performance of our scanner on a Red Hat Linux 9 host (kernel version 2.4.20-8). The CPU is a 730 MHz Pentium III processor with 128MB RAM. The analysis engine runs on a Windows PC with 2.8GHz Pentium 4 processor with 512MB RAM. We constructed examples with configurations similar to the network in section 5, but with different numbers of web servers, file servers and workstations.

To analyze a network in the MulVAL reasoning engine, one needs to run the MulVAL scanner on each host and transfer the results to the host running the analysis engine. The scanners can execute in parallel on multiple machines. The analysis engine then operates on the data collected from all hosts. Since the functioning of the scanner is the same on various hosts, we measured the scanner running time on one host. We measured the running time for the analysis engine for real and synthetic benchmarks. The running times (in seconds) are as:

MulVAL scanner		236 s
MulVAL reasoning engine	§5.1	0.08
	1 host	0.08
	200 hosts	0.22
	400 hosts	0.75
	1000 hosts	3.85
	2000 hosts	15.8

MulVAL scanner is the time to run the scanner on one (typically configured) Linux host; in principle, the scanner can run on all hosts in parallel. The benchmark §5.1 is the real-world 3-host network described in section 5.1. Each benchmark labeled "n hosts" consists of n similar Linux hosts, (approximately one third web servers, one-third file servers, and one-third workstations), with host access rules (i.e., firewalls) similar to §5.1. Our reasoning engine can handle networks with thousands of hosts in less than a minute.

A typical network might have a dozen kinds of hosts: many web servers, many file servers, many compute servers, many user machines. Depending on network topology and installed software (e.g., are all the web servers in the same place with respect to firewalls, and are they all

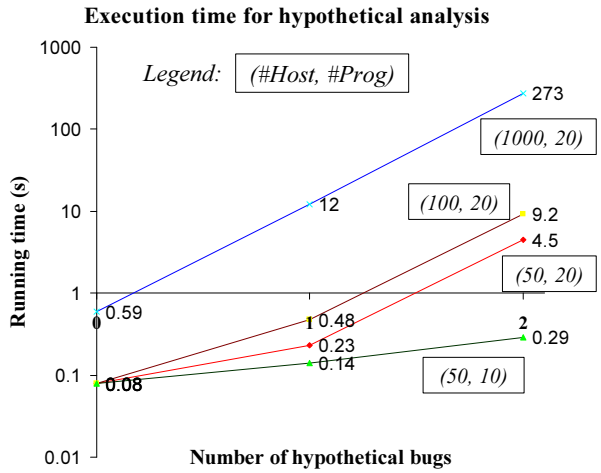

Figure 3: Hypothetical analysis. For a network of 1000 hosts running 20 kinds of installed software, analyzing security assuming the existence of any 1 unreported vulnerability takes 12 seconds.

running the same software?) it may be possible that each group of hosts can be treated as one host for vulnerability analysis, so that $n = 12$ rather than $n = 12,000$. It would be useful to formally characterize the conditions under which such grouping is sound.

To test the speed of our hypothetical analysis, we constructed synthesized networks with different numbers of hosts and different numbers of programs. Each program runs on multiple machines. Since the hypothetical analysis goes through all combination of programs to inject bugs, the running time is dependent on both the number of programs and the number of hypothetical bugs. Figure 3 shows the performance with regard to different number of hosts, number of programs and number of injected bugs. The running time increases with the number of hypothetical bugs, because the analysis engine will need to go through $\binom{n}{k}$ combinations of programs, where n is the number of different kinds of programs and k is the number of injected bugs. $k = 0$ is the case where no hypothetical bug is injected. The performance degraded significantly with the increase of k. But it still only takes 273 seconds for $k = 2$ on a network with 1000 hosts and 20 different kinds of programs. Since hypothetical analysis can be performed offline before the existence of a bug is known, it is not important to have fast real-time response time. The degraded performance is acceptable. Figure 3 shows our system can perform this analysis in a reasonable time frame for a big network.

The input size to the MulVAL reasoning engine is:

Data	Source[1]	hosts=200	=2000
Data Bind	sys admin	26	3004 lines
Policy	sys admin	3	3
Principal Bind	sys admin	10	10
HACL	Smart Firewall	342	3342
Scanner Output	OVAL/ICAT	1222	12022

Coverage Our system can reason about privilege escalation vulnerabilities and denial of service vulnerabilities. We cannot currently reason about confidentiality loss or integrity loss vulnerabilities. Overall, we could reason about 84% of the Red Hat Linux bugs reported in OVAL. The detailed statistics are (as of January 31, 2005):

OVAL definitions for Red Hat	202
Those with PrivEsc or only DoS	169
Coverage	84%

Size of our code base To implement our framework on Red Hat platform, we adapted the OVAL scanner and wrote the interaction rules. The size of our code base is:

Module	Original	New
OVAL scanner	13484	668 lines
Interaction rules		393

The modularity and simplicity of our design allowed us to effectively leverage the existing tools and databases by writing about a thousand lines of code. We note that the small size and declarative style of our interaction rules makes them easy to understand and debug. The interaction rules model Unix-style security semantics. We foresee that to reason about Windows platforms in addition, the effort involved is comparable. The rules are independent of the vulnerability definitions.

7.1 Scanning a distributed network

We measured the performance of running the MulVAL scanner in parallel on multiple hosts. We used PlanetLab, a worldwide testbed of over 500 Linux hosts connected via the Internet [20]. We selected 47 hosts in such a way as to get geographical diversity (U.S., Canada, Switzerland, Germany, Spain, Israel, India, Hong Kong, Korea,

[1]The indicated "Source" shows what person or tool would provide the information in a real installation; for this benchmark measurement, we constructed the data synthetically.

Japan). We were able to log into 39 of these hosts; of these, we successfully installed the scanner on 33 hosts.[5] We ran a script that, in parallel on 33 hosts, opened an SSH session and ran the MulVAL scanner. We assume that many hosts were carrying a normal workload, as we made no attempt to reserve them for this use. The first host responded with data in 1.18 minutes; the first 25 hosts responded within 10 minutes; the first 29 hosts responded within 15 minutes; at this point we terminated the experiment.

For a local area network, we expect fast and uniform response time. But for distributed networks, we recommend that scanning be done asynchronously. Each machine, either when its configuration is known to have changed or periodically, should scan and report configuration information. Then, whenever newly scanned data arrives or whenever new vulnerability data is obtained from OVAL or ICAT, the reasoning engine can be run within seconds.

8 Discussion

8.1 Implementing a scanner

Currently the MulVAL scanner is implemented by augmenting the standard off-the-shelf OVAL scanner. The OVAL scanner is overloaded with both the task of collecting machine configuration information and the task of comparing the configuration with formal advisories to determine if vulnerabilities exist on a system. The drawback of this approach is that when a new advisory comes, the scanning will have to be repeated on each host. It would be more desirable if the collection of configuration information can be separated from the recognition of vulnerabilities, such that when a new bug report comes, the analysis can be performed on the pre-collected configuration data.

There are also many other issues related to scanning, such as how to deal with errors in configuration files. A full discussion of configuration scanning is out of the scope of this paper.

8.2 Modeling normal software behavior

Let us consider the `sudo` program in GNU/Linux operating system. It is a mechanism to enable a permitted user to execute a command as the superuser or another user, as specified in the `sudoers` configuration file. Upon execution, the `sudo` program runs with superuser privileges and the command supplied as argument is executed as superuser or another user depending on the configuration.

Suppose that there is a misconfiguration in the `sudoers` file that lets any user execute any command as user `joe`. In order to do so the scanner must understand the configuration file `sudoers` and the interaction rules modeling the behavior of program `sudo` must be added. In general, we expect that we need to model the normal software behavior of a small number of programs. Although it's easy enough to model new programs using Datalog clauses, a substantial advantage of our approach has been that the set of modeling clauses grows much more slowly than the number of advisories.

9 Related Work

There is a long line of work on network vulnerability analysis [27, 25, 23, 24, 1, 17]. These works did not address how to automatically integrate vulnerability specifications from the bug-reporting community into the reasoning model, crucial for applying the analysis in practice. A major difference between MulVAL and these previous works is that MulVAL adopts Datalog as the modeling language, which makes integrating existing bug databases straightforward. Datalog also makes it easy to factor out various information needed in the reasoning process, which enabling us to leverage off-the-shelf tools and yield a deployable end-to-end system.

Ritchey and Amman proposed using model checking for network vulnerability analysis [23]. Sheyner, et. al extensively studied attack-graph generation based on model-checking techniques [24]. MulVAL adopts a logic-programming approach and uses Datalog in the modeling and analysis of network systems. The difference between Datalog and model-checking is that derivation in Datalog is a process of accumulating true facts. Since the number of facts is polynomial in the size of the network, the process will terminate efficiently. Model checking, on the other hand, checks temporal properties of every possible state-change sequence. The number of all possible states is exponential in the size of the network, thus in the worst case model checking could be exponential. However, in network vulnerability analysis it is normally not necessary to track every possible state change sequence. For network attacks, one can assume the *monotonicity property* — gaining privileges does not hurt an attacker's ability to launch more attacks. Thus when a fact is derived

stating that an attacker can gain a certain privilege, the fact can remain true for the rest of the analysis process. Also, if at a certain stage an attacker has multiple choices for his next step, the order in which he carries out the next attack steps is irrelevant for vulnerability analysis under the monotonicity assumption. While it is possible that a model checker can be tuned to utilize the monotonicity property and prune attack paths that do not need to be examined, model checking is intended to check rich temporal properties of a state-transition system. Network security analysis requires only a small fraction of model-checking's reasoning power. And it has not been demonstrated that the approach scales well for large networks.

Amman et. al proposed a graph-based search algorithms to conduct network vulnerability analysis [1]. This approach also assumes the monotonicity property of attacks and has polynomial time complexity. The central idea is to use an *exploit dependency graph* to represent the pre- and postconditions for exploits. Then a graph search algorithm can "string" individual exploits and find attack paths involves multiple vulnerabilities. This algorithm is adopted in Topological Vulnerability Analysis (TVA) [13], a framework that combines an exploit knowledge base with a remote network vulnerability scanner to analyze exploit sequences leading to attack goals. However, it seems building the exploit model involves manual construction, limiting the tool's use in practice. In MulVAL, the exploit model is automatically extracted from the off-the-shelf vulnerability database and no human intervention is needed. Compared with a graph data structure, Datalog provides a declarative specification for the reasoning logic, making it easier to review and augment the reasoning engine when necessary.

Datalog has also been used in other security systems. The Binder [7] security language is an extension of Datalog used to express security statements in a distributed system. In D1LP, the monotonic version of Delegation Logic [15], Datalog is extended with delegation constructs to represent policies, credentials, and requests in distributed authorization. We feel Datalog is an adequate language for many security purposes due to its declarative semantics and efficient reasoning.

Modeling vulnerabilities and their interactions can be dated back to the Kuang and COPS security analyzers for Unix [2, 8]. Recent works in this area include the one by Ramakrishnan and Sekar [21], and the one by Fithen et al [9]. These works consider vulnerabilities on a single host and use a much finer grained model of the operating system than ours. The goal is to analyze intricate interactions of components on a single host that would render the system vulnerable to certain attacks. The result of this analysis could serve as attack methodologies to be added as interaction rules in MulVAL. Specifically, it is possible that one can write an interaction rule that expresses the attack pre and postconditions without mentioning the details of how the low-level system components interact. These rules can then be used to reason about the vulnerability at the network level. Thus the work on single-host vulnerability analysis is complementary to ours.

MulVAL leverages existing work to gather information needed for its analysis. OVAL [26] provides an excellent baseline method for gathering per-host configuration information. Also, research in the past ten years has yielded numerous tools that can manage network configurations automatically [11, 12, 3, 4]. Although these works do not directly involve vulnerability analysis, they provide a good abstraction for the network model, which is used in MulVAL and simplifies its reasoning process.

Intrusion detection systems have been widely deployed in networks and extensively studied in the literature [5, 16, 14]. Unlike IDS, MulVAL aims at detecting potential attack paths *before* an attack happens. The goal of the work is not to replace IDS, but rather to complement it. Having an a priori analysis on the configuration of a network is important from the defense-through-depth point of view. Undoubtedly, the more problems discovered before an attack happens, the better the security of the network.

10 Conclusion

We have demonstrated how to model a network system in Datalog so that network vulnerability analysis can be performed automatically and efficiently. Datalog enables us to effectively incorporate bug databases into our analysis and leverage existing vulnerability and configuration scanning tools. With all the information represented in Datalog, a simple Prolog program can perform "what-if" analysis for hypothetical software bugs efficiently. We have implemented an end-to-end system and tested it on real and synthesized networks. MulVAL runs efficiently for networks with thousands of hosts, and it has discovered interesting security problems in a real network.

Notes

[1]Common Vulnerabilities and Exposures (CVE) is a list of standardized names for vulnerabilities and other information security exposures. http://cve.mitre.org

[2]http://oval.mitre.org/oval/

[3]Different `Priv` constructors distinguish between setuid and non-setuid permissions. For lack of space in this paper, we have not described the details of our privilege model, which combines concrete users accounts and special symbols that represent groups of accounts.

[4]In this benchmark we did not model hundreds of user machines. We recommend that these should be modeled as we did "internet," as one machine. In this case, unlike "internet, " the host would have non-malicious users, but would be assumed to have many vulnerabilities. In our future work we plan to experiment with such models; at present we recommend our framework for networks of managed, not unmanaged, hosts.

[5] Normally one needs root privileges to install the scanner; PlanetLab gives its users fake "root" privileges in a chroot environment; for production use of MulVAL, root privileges are advisable.

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] R. Baldwin. Rule based analysis of computer security. Technical Report TR-401, MIT LCS Lab, 1988.

[3] 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.

[4] James Burns, Aileen Cheng, Proveen Gurung, David Martin, Jr., S. Raj Rajagopalan, Prasad Rao, and Alathurai V. Surendran. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, Anaheim, California, June 2001.

[5] Frdric Cuppens and Alexandre Mige. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 202. IEEE Computer Society, 2002.

[6] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[7] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105. IEEE Computer Society, 2002.

[8] Daniel Farmer and Eugene H. Spafford. The cops security checker system. Technical Report CSD-TR-993, Purdue University, September 1991.

[9] William L. Fithen, Shawn V. Hernan, Paul F. O'Rourke, and David A. Shinberg. Formal modeling of vulnerabilities. *Bell Labs technical journal*, 8(4):173–186, 2004.

[10] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 221–230, New York, NY, USA, 1988. ACM Press.

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

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

[13] Sushil Jajodia, Steven Noel, and Brian O'Berry. Topological analysis of network attack vulnerabity. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challanges*, chapter 5. Kluwer Academic Publisher, 2003.

[14] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *The 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

[15] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003. To appear.

[16] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 245–254. ACM Press, 2002.

[17] 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.

[18] National Institute of Standards and Technology. ICAT metabase. http://icat.nist.gov/icat.cfm, October 2004. web page fetched on October 28, 2004.

[19] Giridhar Pemmasani, Hai-Feng Guo, Yifei Dong, C.R. Ramakrishnan, and I.V. Ramakrishnan. Online justification for tabled logic programs. In *The 7th International Symposium on Functional and Logic Programming*, April 2004.

[20] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[21] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1-2):189–209, 2002.

[22] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[23] 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.

[24] 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.

[25] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38. ACM Press, 2000.

[26] Matthew Wojcik, Tiffany Bergeron, Todd Wittbold, and Robert Roberge. Introduction to OVAL: A new language to determine the presence of software vulnerabilities. http://oval.mitre.org/documents/docs-03/intro/intro.html, November 2003. Web page fetched on October 28, 2004.

[27] Dan Zerkle and Karl Levitt. NetKuang–A multi-host configuration vulnerability checker. In *Proc. of the 6th USENIX Security Symposium*, pages 195–201, San Jose, California, 1996.

A A Sample Attack Trace

In this section, we present a trace for the example policy violation discussed in section 5.2. We wrote a meta-interpreter to generate the attack tree and visualize it in plain text or html format. In the future we hope to use XSB's online justifier [19] to dump an attack graph and visualize it.

The trace for one of the policy violation is shown below. Each internal node is attributed with the rule used to derive the node.

```
|-- policyViolation(attacker,read,projectPlan)
 |-- dataBind(projectPlan,workStation,/home)
 |-- accessFile(attacker,workStation,read,'/home')
 Rule: execCode implies file access
    |-- execCode(attacker,workStation,root)
    Rule: Trojan horse installation
       |-- malicious(attacker)
       |-- accessFile(attacker,workStation,write,'/sharedBinary')
       Rule: NFS semantics
          |-- nfsMounted(workStation,'/sharedBinary',fileServer,'/export',read)
          |-- accessFile(attacker,fileServer,write,'/export')
          Rule: execCode implies file access
             |-- execCode(attacker,fileServer,root)
             Rule: remote exploit of a server program
                |-- malicious(attacker)
                |-- vulExists(fileServer,CAN-2003-0252,mountd,remoteExploit,privEscalation)
                |-- networkServiceInfo(fileServer,mountd,rpc,100005,root)
                |-- netAccess(attacker,fileServer,rpc,100005)
                Rule: multi-hop access
                   |-- execCode(attacker,webServer,apache)
                   Rule: remote exploit of a server program
                      |-- malicious(attacker)
                      |-- vulExists(webServer,CAN-2002-0392,httpd,remoteExploit,privEscalation)
                      |-- networkServiceInfo(webServer,httpd,tcp,80,apache)
                      |-- netAccess(attacker,webServer,tcp,80)
                      Rule: direct network access
                         |-- located(attacker,internet)
                         |-- hacl(internet,webServer,tcp,80)
                   |-- hacl(webServer,fileServer,rpc,100005)
             |-- localFileProtection(fileServer,root,write,/export)
    |-- localFileProtection(workStation,root,read,/home)
 |-- not allow(attacker,read,projectPlan)
```

Figure 4: A sample attack tree