A logic-programming approach to network security analysis

Xinming Ou

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2005

© Copyright by Xinming Ou, 2005.

Abstract

An important problem in network security management is to uncover potential multistage, multihost attack paths due to software vulnerabilities and misconfigurations. This thesis proposes a logic-programming approach to conduct this analysis automatically. We use Datalog to specify network elements and their security interactions. The multihost, multistage vulnerability analysis can be conducted by an off-the-shelf logic-programming engine that can evaluate Datalog efficiently.

Compared with previous approaches, Datalog is purely declarative, providing a clear specification of reasoning logic. This makes it easy to leverage multiple third-party tools and data in the analysis. We built an end-to-end system, MulVAL, that is based on the methodology discussed in this thesis. In MulVAL, a succinct set of Datalog rules captures generic attack scenarios, including exploiting various kinds of software vulnerabilities, operating-system sematics that enables or prohibits attack steps, and other common attack techniques. The reasoning engine takes inputs from various off-the-shelf tools and formal security advisories, performs analysis on the network level to determine if vulnerabilities found on individual hosts can result in a condition violating a given high-level security policy.

Datalog is a language that has efficient evaluation, and in practice it runs fast in off-the-shelf logic programming engines. The flexibility of general logic programming also allows for more advanced analysis, in particular hypothetical analysis, which searches for attack paths due to unknown vulnerabilities. Hypothetical analysis is useful for checking the security robustness of the configuration of a network and its ability to guard against future threats. Once a potential attack path is discovered, MulVAL generates a visualized attack tree that helps the system administrator understand how the attack could happen and take countermeasures accordingly.

Acknowledgments

I would like to thank my advisor Andrew Appel for his guidance, wisdom, and support throughout my five years at Princeton. Andrew introduced me to the fields of programming languages and formal methods, and most importantly, helped me identify the important problem of formalizing the analysis of network security. In retrospect, I feel that I have been very lucky to have someone who has such a far-reaching insight in scientific research, encourages me to tackle the real hard problems, and gives me the most crucial encouragement at the most difficult times.

I would like to thank Raj Rajagopalan for the many inspiring discussions we have had ever since the beginning of this research. His visions in security research, at once sound with clear theoretical reasoning and practical with a deep understanding of real problems in the field, set a model for me as to what is meaningful computer science research.

I would like to thank the two readers on my committee, Edward Felten and Jonathan Smith, not only for spending tremendous amount of time helping me improve the presentation of this dissertation, but also for providing invaluable inputs and suggestions ever since I started working on this project.

At last, I would like to thank my fellow graduate students at Computer Science Department, who are largely responsible for making my experience at Princeton a memorable one.

This research was supported in part by DARPA award F30602-99-1-0519 and by ARDA award NBCHC030106.

To my parents

Contents

Abstract iii

1 Introduction **1**

1.1 Software vulnerabilities and network security management 1

1.2 Previous works on vulnerability analysis 5

1.3 Specification language 12

1.4 The modeling problem 14

1.4.1 Formal model of vulnerability 16

1.4.2 Configuration scanners 18

1.5 Policy-based analysis 19

1.6 Contributions 21

2 Formal model of reasoning **24**

2.1 Datalog review 24

2.2 Analysis framework 26

2.3 Interaction rules 26

2.3.1 Types of constants 27

2.3.2 Vulnerability rules 29

2.3.3 Exploit rules 30

2.3.4 File access 34

2.3.5 Trojan-horse programs 36

2.3.6 NFS semantics 37

2.3.7 User credentials 40

2.4 Network topology 43

2.4.1 Host Access Control List 43

2.4.2 Multihop host access 44

2.5 Policy specification 44

2.5.1 Binding information 45

2.6 Discussion 47

2.6.1 Using negations in the model 47

2.6.2 Nonmonotonic attacks 48

3 Analysis database 50

3.1 Vulnerability specification 50

3.1.1 Recognition specification 51

3.1.2 Semantics specification 53

3.2 Host configuration 59

3.3 Network configuration 64

3.4 Binding information 64

3.5 Putting everything together 64

4 Basic analysis 66

4.1 Datalog evaluation and XSB 66

4.1.1 Properties of Datalog evaluation in XSB 69

4.2 Attack simulation 70

4.3 Policy check 71

4.3.1 More policies 72

4.4 Attack-tree generation 74

4.5 Attack-graph generation 76

5 Hypothetical analysis 78

5.1 Definition 79

5.2 Conducting hypothetical analysis in Prolog 80

6 Practical Experience 84

6.1 Experimental result on small networks 84

6.1.1 A small real-world example 84

6.1.2 An example multihost attack 89

6.1.3 Hypothetical analysis 94

6.2 Performance and Scalability 94

7 Conclusions 100

A Interaction Rules for Unix-family Platform 102

B Meta-programming in XSB 109

B.1 A meta-interpreter for definite Prolog programs 109

B.2 A meta-interpreter for generating proofs 111

B.3 Dealing with negation and side effects 112

Chapter 1

Introduction

1.1 Software vulnerabilities and network security management

Dealing with software vulnerabilities on network hosts poses a challenge to network administration. The past 15 years have seen an ever-growing number of security vulnerabilities discovered in software (and information systems in general). According to the statistics published by CERT/CC, a central organization for reporting security incidents, the number of reported vulnerabilities have grown considerably in the last five years (Figure 1.1). It is expected that the rate at which new software vulnerabilities emerge will continue to increase in the foreseeable future. With thousands of new vulnerabilities discovered each year, maintaining a 100% patch level is untenable and sometimes undesirable for most organizations. While in many cases patches come right after vulnerability reports, people do not always apply patches right away for various reasons [3]. Hastily written patches are unstable and may even introduce more bugs. Patching an operating system kernel often requires a reboot, affecting

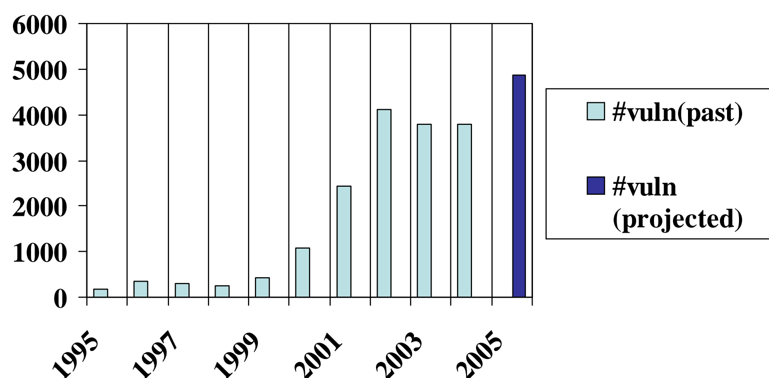

Figure 1.1: Number of vulnerabilities reported by CERT (http://www.cert.org/stats/cert_stats.html)

availability in a way that may be cost-prohibitive for some organizations. Thus it is not uncommon for a network administrator to keep running buggy software for a period of time after the bug has been reported. As part of a disciplined enterprise risk-management program, security managers must make decisions on which information systems are most critical and prioritize security countermeasures for such systems. They must make sure any potential exploit of the unpatched bugs will not happen, or even if it did happen it would not cause damage. One of the daily chores of administrators is to read vulnerability reports from various sources and understand which reported vulnerabilities can actually compromise the security of their managed network. Some bugs may not be exploitable under the settings of the local network. Even when they can be exploited, the access gained by the attacker may be no more than what he is already permitted.

For example, in the network of Figure 1.2, there may exist vulnerabilities on machine `webServer`. But if a bug on `webServer` is only locally exploitable[1] and all users with accounts on `webServer` are trusted, there is no immediate danger of

[1]A bug is locally exploitable if the attacker has to first gain some local access on a machine, *e.g.* a login shell of a user.

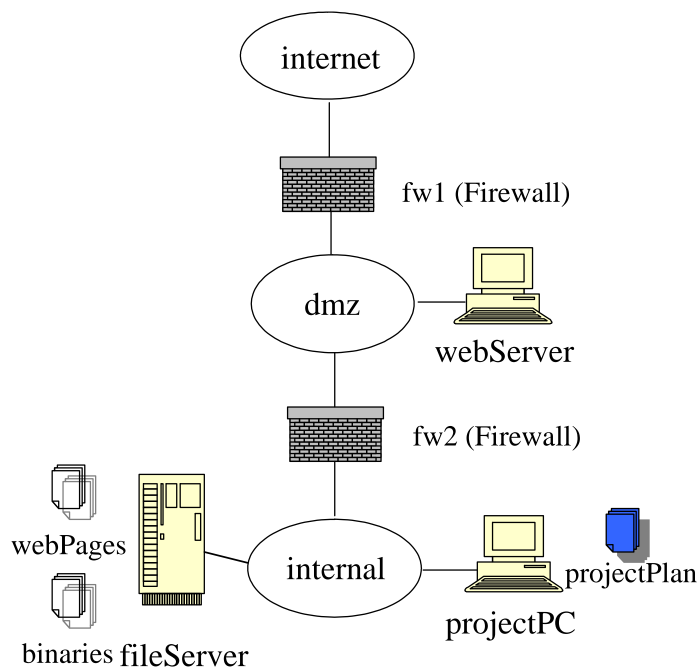

Figure 1.2: An example network

exploit. If the bug is remotely exploitable[2] but the firewall `fw1` blocks the traffic to the vulnerable port, the machine is still safe. If the firewall allows access to the vulnerable port (perhaps for normal access to `webServer`), but the consequence of a potential exploit is only that an attacker can read `webPages`, it is also safe because the data is supposed to be publicly available anyway.

In the wake of new vulnerabilities, assessment of their security impact on the network infrastructure is important in choosing the right countermeasures: patch and reboot, reconfigure a firewall, unmount a file-server partition, and so on. Unfortunately, the way a network can be broken into is not always obvious. For the example network in Figure 1.2, if one day a new vulerability is reported about the web service program on `webServer`, it would not seem to be an imminent threat to the confidential data `projectPlan` stored on `workStation`. However, depending on the configuration

[2] A bug is remotely exploitable if an attacker can launch an attack across a network.

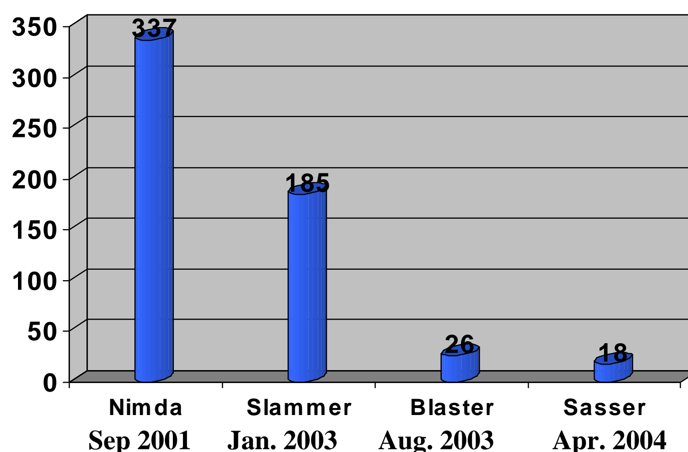

Figure 1.3: Vulnerability-to-exploit window (in days)
(From Sharp Ideas: http://www.sharp-ideas.net/)

of the two firewalls (`fw1` and `fw2`), the configuration of the file server, and the configuration of the workstation, this may not be the case. For example, many corporations use NFS file sharing to mount file system partitions on file servers. NFS is an insecure protocol and adopts a host-based trust relationship. If a client machine is compromised, all the files that are exported to the client can potentially be accessed by the intruder. Thus, if an attacker from the Internet can first compromise `webServer` by exploiting the vulnerability, he can potentially modify files stored on `fileServer`. If the shared executable binaries are stored in a partition exported to the web server, the integrity of the executables will be compromised — the attacker can install a Trojan-horse program. If the same partition is also mounted by a workstation, a user on that machine may execute the Trojan-horse program, thus giving the attacker access to `workStation`. As a result the confidential data `projectPlan` can potentially be leaked to the outside attacker.

In order to discover these potential attack paths in a network, one must not only examine configuration parameters on every network element — machines, firewalls,

routers, *etc.* — but also consider all possible interactions among them. Conducting this multihost, multistage vulnerability analysis by human beings is error-prone and labor-intensive. Automating this assessment process is important given the fact that the window between the time a vulnerability is reported to the time it is exploited on a large scale has diminished substantially [3] (also see Figure 1.3). Defenders of networks and systems can now plan on having only days to deploy countermeasures in protection of the vulnerable systems and services that are connected to public networks. To exacerbate the situation, networks being used in organizations are getting bigger and more complex. Unfortunately, current technology has until now failed to provide adequate methodologies to achieve automatic management of network security. As a result, network configuration management in today's world still depends largely on human experience. According to a survey conducted by the Computing Technology Industry Association, among all security breaches reported by the 900 organizations surveyed in 2004, 84% of them were caused by human errors. The exponential increase in security incidents reported to CERT (Figure 1.4) shows that there is a compelling need for effective methodology to automate network security management.

1.2 Previous works on vulnerability analysis

Automatic vulnerability analysis can be dated back to Kuang [4] and COPS [17]. Kuang formalizes security semantics of UNIX as a set of rules, and conducts search for ways a system can be broken into based on those rules. COPS is a UNIX security checker that incorporates the Kuang rule set. NetKuang [54] extended the rule set in Kuang to capture configuration information that has security impact across a network, such as the `.rhosts` file, and thus is capable of reasoning about misconfigu-

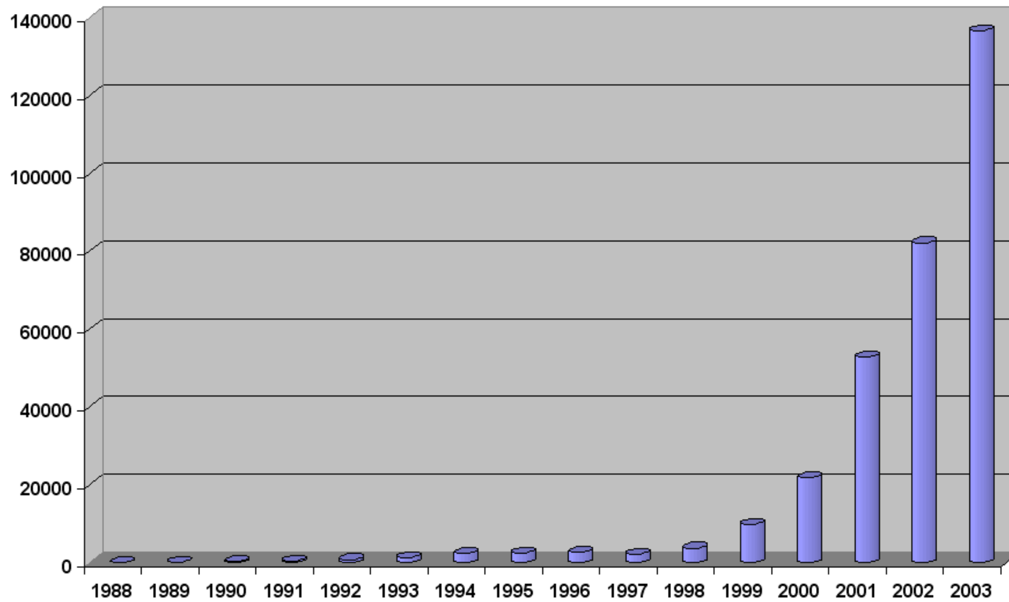

Figure 1.4: Security incidents reported to CERT
(http://www.cert.org/stats/cert_stats.html)

rations within a network of UNIX machines. At the time when Kuang and NetKuang were developed, software vulnerabilities have not become a major problem for network security, and the scale of network attacks was much less than it is today. The rules of Kuang and NetKuang are limited to the few attack scenarios and hardcoded into the implementation. There is no incorporation of third-party security knowledge such as vulnerability advisories. This piece-meal approach can no longer meet the security need for the threats facing computer networks today. For a security analysis tool to be viable with the changing threats, the reasoning logic must be formally specified and separated from implementation. The formal specification should be able to incorporate information from third-party agencies that provide software vulnerability definitions. The reasoning must be sound in theory and efficient in practice.

Levitt and Templeton proposed a *requires* and *provides* model for computer attacks [48], which essentially specifies the pre- and postcondition of each attack step.

This allows for multiple attack steps being combined such that previous steps provide necessary conditions for later ones to succeed, leading to discovery of attack paths not obvious by looking at each component in isolation. Levitt's model has a clear semantics for attacks and is much more flexible than signature-based models. This idea has been matrerialzed in various works of vulnerability analysis. In terms of specific modeling and analysis mechanisms, two approaches have been proposed: model checking and exploit-dependency graph search.

Using model checking in network vulnerability analysis was first proposed by Ritchey and Ammann [43]. In the model-checking approach, a network is modeled as a state-transition system. The configuration information is encoded as state variables. An attack step is modeled as a transition relation between two states. A transition relation is specified in the form of $(\mathcal{S}_1, \mathcal{S}_2)$, where $\mathcal{S}_1$ is the values of boolean variables characterizing the preconditions of the attack, and $\mathcal{S}_2$ represents the postcondition of the attack. An attack path manifests itself as a sequence of valid state transitions from the initial state leading to a state where the security property of the network is broken. A model checker can check the model against a temporal formula, which can express properties such as "all states reachable from $\mathcal{S}_0$ will satisfy the given security property", where $\mathcal{S}_0$ is the known initial state of the network. If the formula satisfies the model, no attack paths can lead to a bad situation. If the formula does not satisfy the model, the model checker can output a sequence of state transitions that ends up at a state in which the security property does not hold. This counterexample trace shows an attack path that leads to the violation of the security property.

The advantage of the model-checking approach is that one can leverage the reasoning power of off-the-shelf model checkers rather than writing a customized analysis engine. However, one has to be careful to avoid the combinatorial explosion that often occurs in model checking. In software engineering, people have proposed various ap-

proaches to make model checking fast in verifying safety properties of large software systems [21, 1, 53]. However, there has been no work showing techniques that can speed up model checking in software verification can also speed up network security analysis. The only experimental data we can find that shows the performance and scalability of using model checking to analyze network vulnerability is in Sheyner, *et al.*'s work [46]. The paper describes an experimental setting that consists of three machines, a router, and a firewall. The number of atomic attacks in the model is four. The run time of the tool on this example is about 5 seconds. When the example is enlarged with two additional hosts, four additional atomic attacks, several new vulnerabilities, and flexible firewall configurations, it took the tool 2 hours to find all attack paths, of which 5 min is spent in model checking and the rest of the time is spent in attack graph generation. This result did not give a convincing evidence that model checking scales well for network security analysis. At this point it is still questionable whether such approach will work for large networks with thousands of hosts.

Model checking is intended to examine rich temporal properties of a state-transition system. While such expressive power is crucial in verifying properties of software and concurrent systems, it is not clear whether the full reasoning power is useful for network security analysis. One problem of using a standard model checker as the analysis engine is that most state transition sequences in the model do not actually need to be examined for the purposes of network security analysis. For network attacks one can assume the *monotonicity property*, under which assumption the checking can be dramatically sped up.

Monotonicity The monotonicity property states that gaining more privileges can only help the attacker in further compromising the system. For example, if there

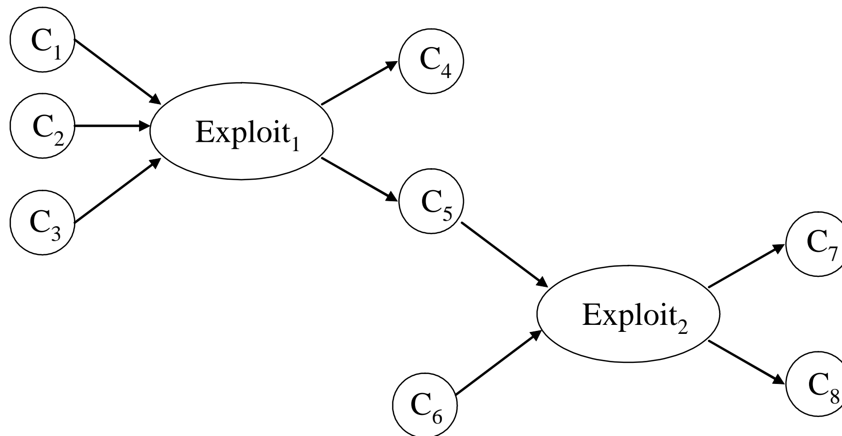

Figure 1.5: Exploit dependency graph

are two web servers that can be compromised by an attacker, attacking one of them typically does not affect his ability to attack the other [3]. Thus, once the analysis derives that the attacker can gain certain privilege, this fact can remain true for the remainder of the anaylsis process. There is no need for backtracking. However, in a standard model checker, all possible paths — ones with the fact being true and ones without — have to be examined. When dealing with large networks, there will be a large number of choices for state transition at each step and this backtracking will waste a significant amount of computing power. In the worst case, this could lead to an exponential blowup. Partial order reduction [35, 19] can eleviate this problem in model-checking software systems. However, it has not been shown how to apply the technique in model-checking network security.

Based on the monotonicity property, Ammann, *et al.* proposed an approach where dependencies among exploits are modeled in a graph structure and attack analysis becomes a graph search problem [2]. Figure 1.5 shows a portion of an exploit dependency graph. A node in the graph is either a *condition* or an *exploit*. A condition is

[3]This assumption does not necessarily hold for nonmonotonic attacks. For example, compromising one web server may trigger the intrusion detection system so that further attack paths are blocked. For more discussions on nonmonotonic attacks, see section 2.6.2.

a boolean variable representing certain state of the system, such as whether a particular version of software is installed on a machine. An exploit can happen if all its preconditions are true. If a condition C_i is a precondition of an exploit e, there will be an edge from the node representing C_i to the node of e. After an exploit is carried out, the state of the network system will change. In a monotone system, the state change only causes more conditions to be true. Those conditions are the postconditions of the exploit and there will be an edge from the exploit to each of its postconditions. Because the number of conditions and exploits is in proportion to the size of the network, the size of the graph is also in proportion to the size of the network. The search algorithm can be viewed as a graph marking process, where a marked condition node is true and an unmarked one is false. An exploit node can be marked if all its predecessors (preconditions) are marked. Then all its successors (postconditions) will also be marked if they have not been. Once a node is marked, it will stay marked forever. The algorithm terminates if no more nodes can be marked. Since every node and edge will be visited only once, the execution time is polynomial in the size of the graph.

This graph-based algorithm based on monotonicity assumption avoids the potential exponential explosion in model checking. However, the algorithm is hardcoded as program code and there is no clear specification of properties being checked and interactions within a network. The work described in this dissertation assumes the same monotonicity property, but adopts a logic-based approach, which formally specifies every relevant element in the reasoning and their interactions. As a result it can put various information and tools together, yielding an end-to-end automatic system.

Attack graphs One purpose of network security analysis is to generate an attack-graph. Roughly speaking, an attack graph is a DAG that represents the dependency

of actions that lead to the violation of the security property of a network. Like the analysis mechanisms, there are also two approaches to representing attack graphs. In one of them, each vertex in the graph represents the state of the whole network system and the edges represent attack steps that cause the network to change from one state to another. We call this a *network-state attack graph* and it corresponds to the model-checking based analysis. The other approach corresponds to the graph-search algorithm based on the monotonicity property, where an attack graph is essentially a portion of the exploit-dependency graph that contributes to the attack.

Sheyner *et al.* extensively studied automatic generation and analysis of network-state attack graphs based on symbolic model checking [46]. Phillips and Swiler also studied network vulnerability analysis based on network-state attack graphs [38], although they did not use model-checking techniques but rather developed a customized attack-graph generation tool [47]. Network-state attack graphs suffer from exponential explosion. In Sheyner's work, the authors report that the running time of their tool grows from 5 seconds to 2 hours when the size of the network grows from 3 hosts to 6 hosts (with other parameters also growing proportionally) [4]. The potential state space grows from 2^{91} to 2^{229}, and the reachable state space grows from 101 to 6190. In Swiler, *et al.*'s work [47], the authors also discussed the issue of graph explosion and proposed several alleviating methods, but no experimental results were given. On the other hand, attack graphs based on exploit-dependency are polynomial because individual conditions, not the whole network states, are represented as nodes. While there is only a polynomial number of conditions, the number of all possible states are exponential.

The problem with network-state attack graphs is that they do not utilize the

[4]The authors did report that the model checking part of the larger example took only 5 minutes and the 2-hour running time was largely due to the graph generation process.

monotonicity property. Since launching one attack does not decrease the attacker's ability to launch another, the order in which independent attack steps are carried out is not important. But this order is explicit in network-state attack graphs, which results in exponential number of redundent attack paths that differ only in the order of attack steps. The method proposed by Swiler, *et al.* [47] to eliminate those redundant attack paths is actually an implicit use of an exploit-dependency graph by enforcing a total order on network conditions.

1.3 Specification language

An important step in network security analysis is to specify, in a machine readable format, the network elements and how they interact. Then an automatic analysis engine can compute possible attack paths based on the specification. A clear specification is crucial in building a viable analysis tool. Security is a problem that involves every aspect of a system. Both intended and unintended behaviors of system components may be utilized in an attack. Any system that hardcodes the security knowledge in the implementation is doomed to fail in the face of ever-growing threats. Given the rate at which new vulnerabilities are reported, an automatic tool must be able to take as input formal specification of security bugs. A clear specification of the analysis logic makes it easier to integrate such expert knowledge from independent sources, such as CERT, CVE, and other bug-reporting agencies. Attack methodologies evolve as new technologies are invented which bring more complex interactions among elements in a network system. Any security analysis tool is incapable of capturing all those interactions. Specifying those interactions in a formal, declarative language makes it easy to understand what can and cannot be handled by the tool, and to enhance the tool when necessary. The analysis process also needs to know numerous

configuration parameters of every machine in the network, as well as those of the routers, firewalls, and switches. Various scanning tools have been developed recently that can provide this configuration information [52, 6, 7]. A clear specification of the analysis logic makes it possible to factor out various configuration information and leverage the corresponding tools to collect them, instead of reinventing the wheel.

The clarity of specification has not been given enough emphasis previously. In the model-checking approach, the network state is modeled as a collection of boolean variables, each representing some condition on the network. The security interactions are specified as state transition relations. While it is possible to make this encoding modular and extensible, its artificiality makes it hard to understand for human beings. In the exploit-dependency graph, the network conditions are encoded as labels in the graph. The security interactions are encoded as graph edges. This encoding also lacks the level of clarity provided by a formal specification language. Tidwell, *et al.* proposed a language for modeling Internet attacks [49]. However, the language is too complicated and it is not clear how easy it is to use third-party security knowledge or scanner output in the language.

The work described in this dissertation addresses the problem by adopting a logic-based approach. The interactions among network elements are specified formally in the logic-programming language Datalog [11]. Datalog is a syntactic subset of Prolog, so the specification is also a program that can be loaded into a standard Prolog environment and executed. Datalog has a clear declarative semantics and it is a monotone logic, making it especially suitable for network attack analysis. Datalog is popular in deductive databases, and several decades of work in developing reasoning engines for databases has yielded tools that can evaluate Datalog efficiently [41, 51]. Leveraging those evaluation engines allows for analyzing large enterprise networks with thousands of machines. A deeper reason for adopting a logic-based approach is

that it captures human reasoning, which is exactly what a system administrator has to do today in managing the security of networks. The reasoning system described in this dissertation can be viewed as an expert system that alleviates the burden of reasoning about large and complex systems from human beings, whose brain power cannot keep up with the scale of the task.

1.4 The modeling problem

While choosing the right specification language is important, a harder problem is deciding what to specify. For any analysis model, there will always be attack scenarios that are not captured. However, the vast majority of security incidents do not involve clever inventions of new attack methodologies, but rather consist of attack steps using stale techniques known for years or even decades. The reason they are hard to prevent is not because the system administrators are not aware of those techniques, but rather because the size of the system makes it impossible for a human being to capture every possible way the components may interact. The major challenge in designing a vulnerability analysis system is identifying the correct granularity under which the components of a network are modeled, such that the interactions among components that vary from one network to another can be examined automatically, whereas the details of individual attack steps that are common to all networks are abstracted out.

Modeling a computer system to detect security vulnerabilities caused by interactions among system compoents dates back to Baldwin's Kuang system [4], which is incorporated into the COPS Unix security checker [17]. Recent work includes Ramakrishnan and Sekar [40], and Fithen, *et al.* [18]. These works deal with vulnerabilities on a single host and the system is modeled at a fine grain such that unknown techniques of compromising a single system can be discovered. However,

for network-level analysis, using such fine-grained model is not desirable, because the focus is more on interactions among different hosts, not within a single host. Modeling too much details on a single host will likely lead to duplication of reasoning across multiple machines. The purpose of network vulnerability analysis is not to identify unknown ways to compromise a single system, but rather to uncover *multi-host, multistage* attack paths where each individual attack step utilizes some attack methodology well known to the literature. For this reason, the model for network security analysis should be coarser-grained than that for a single host. The result of a single-host vulnerability analysis can be abstracted as one interaction rule for the network-level analysis.

In deciding upon the granularity of the model, this thesis adopts a "model as needed" approach. Specifically, aspects of a system are modeled only if they are relevant to determining the preconditions and consequences of some known attack methodologies. For example, a common attack methodology is buffer overrun, in which an attacker sends a specially crafted input to a vulnerable program that causes the program's memory boundary to be exceeded. If the program does not perform rigorous check on input, a malicious input can contaminate the execution stack and override the return address to make the program jump to injected malicious code. If a service program has a buffer overrun bug, a remote attacker can potentially execute arbitrary code as the user under which the service is running. To model a buffer overrun attack against a service program, one needs to model the protocol and port under which the program is listening, because it is relevant in determining whether an attacker is able to send a malicious packet to the program; one also needs to model the user privilege of the service process, because it is relevant to the consequence of the attack. We do not need to model, for example, the stack layout of the program. Although it is relevant to whether the attack can be successful, this is not the task

of the network security analysis. A software security analyst, on the other hand, can study the stack layout of a buggy program and determine if a bug will enable an attacker to take full control of the program's process, or just to crash it. Once a conclusion is reached, the result should be formally specified and directly used in the network-level analysis.

1.4.1 Formal model of vulnerability

A vulnerability is an unintended behavior of a component that can be exploited by an attacker. Most network intrusions involve some vulnerability on software installed on networked hosts. There are several well known sources for reporting security-relevant software bugs — CERT, CVE, BugTraq, and so on. However, the bug reports are usually written as informal natural language descriptions and cannot be directly used in automatic analysis. Figure 1.6 shows an example bug description from CERT.

Two kinds of information in the report are useful in vulnerability analysis. One is how to check if the vulnerability exists on a system, such as the version number of the buggy software and the configuration options under which it manifests. We call this the *recognition specification*. The other is the precondition under which the bug can be exploited and the consequence of the exploit. We call this the *semantics specification*. To automate the vulnerability assessment process, both information need to be formalized.

Currently, the *Open Vulnerability Assessment Language* (OVAL) [52] is being developed which formalizes machine configuration tests. Recognition specification of reported software vulnerabilities in the form of OVAL definitions are now being released by the bug-reporting community. Other formal recognition specifications of vulnerabilities include the *Nessus Attack Scripting Language* (NASL) used by the

```
CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability
Original release date: June 17, 2002
Last revised: March 27, 2003
Source: CERT/CC

Systems Affected

    * Web servers based on Apache code versions 1.2.2 and above
    * Web servers based on Apache code versions 1.3 through 1.3.24
    * Web servers based on Apache code versions 2.0 through 2.0.36

Overview

There is a remotely exploitable vulnerability in the way that Apache web
servers (or other web servers based on their source code) handle data encoded
in chunks. This vulnerability is present by default in configurations of Apache
web server versions 1.2.2 and above, 1.3 through 1.3.24, and versions 2.0
through 2.0.36. The impact of this vulnerability is dependent upon the software
version and the hardware platform the server is running on.

I. Description

Apache is a popular web server that includes support for chunk-encoded data
according to the HTTP 1.1 standard as described in RFC2616. There is a
vulnerability in the handling of certain chunk-encoded HTTP requests that may
allow remote attackers to execute arbitrary code.

The Apache Software Foundation has published an advisory describing the details
of this vulnerability. This advisory is available on their web site at

    http://httpd.apache.org/info/security_bulletin_20020617.txt

Vulnerability Note VU#944335 includes a list of vendors that have been contacted
about this vulnerability.

II. Impact

For Apache versions 1.2.2 through 1.3.24 inclusive, this vulnerability may
allow the execution of arbitrary code by remote attackers. Exploits are publicly
available that claim to allow the execution of arbitrary code.

For Apache versions 2.0 through 2.0.36 inclusive, the condition causing the
vulnerability is correctly detected and causes the child process to exit.
Depending on a variety of factors, including the threading model supported by
the vulnerable system, this may lead to a denial-of-service attack against the
Apache web server.
```

Figure 1.6: A CERT advisory

Nessus security scanner [6]. However, there has been much less vigorous effort in formalizing the semantic specification of software security bugs. What exists is classifications according to exploitable range and consequences, found in some vulnerability databases such as NVD (National Vulnerability Database), and OSVDB (Open Source Vulnerability Database). These classifications do not give precise specification of a vulnerability's semantics. But since many exploits happen in similar ways, they can still provide useful input to a reasoning system.

1.4.2 Configuration scanners

Once the formal model of reasoning is decided upon, configuration scanners are needed to collect system information that is used by the model. For example, if the formal model needs to know the port number and protocol under which a service program is listening, the scanners on every host should collect this information and report it in the data format of the reasoning model. Although conceptually simple, the time and energy involved in implementing and testing such scanners is significant. The formal model in the analysis should provide a simple data format so that the labor involved in implementing a scanning tool can be minimized. The model should also be modular so that when new information is needed from the scanner, its scanning ability can be added incrementally without disrupting the existing implementation.

There are off-the-shelf scanners that can take as input formal vulnerability recognition specifications and check if the vulnerability exists on a computer system. Two such scanners are the OVAL "interpreter", which can handle formal vulnerability definition in OVAL, and the Nessus scanner, which can handle vulnerability definitions in NASL. Such scanners provide limited capability of outputing configuration information other than those relevant in testing the existence of certain vulnerabiliities.

For a comprehensive network security analysis, these scanners should be augmented to suit the need of the formal model of reasoning. For the work described in this dissertation, we use the OVAL scanner to report existence of software security bugs on a system, and a separate scanner to collect other configuration parameter. The combination of these two scanners are called a *MulVAL scanner*.

1.5 Policy-based analysis

Recent years have seen progress in policy-based network management. A policy is a set of directives that control access to resources. For example one may have a policy that says only corporate employees can read internal files stored on the file server. A policy is implemented by low-level mechanisms, such as file attributes in a file system.

While the separation of policy from mechanism is an important step towards eliminating human errors, an equally important question is how to make the policy itself less error-prone. A good policy language design should require little technical knowledge to write a "correct" policy. However, this is often hard to achieve, largely because many security problems are caused by complex interactions among network components. The correct behavior of a device is not only dependent on its own configuration, but also on the configuration of others in the network. Extensive research has been conducted to design proper abstractions to specify management policies [30, 7, 10, 13, 42, 29, 25, 33]. The goal is to push the policy to a higher level so that people can write down the ultimate goal of security management in a language that closely matches human intention. A mapping will translate high-level policy specifications to low-level mechanisms. Sometimes the mapping can be done at compile time (when configuring a network), sometimes the mapping has to be done at run time (when a request comes in). The person who writes down the policy does

not need to have deep knowledge in security interactions, whereas the people who define the mapping must be experts in this field.

One major problem in policy-based management is to make sure the mapping from policy to mechanism is done consistently so that nothing is lost in the translation. For example, if the policy says only corporate employees can access internal data, then after this high-level policy is mapped to the low-level mechanisms, such as file attributes, it must be the case that there is absolutely no way that people other than employees can read the data. For example, even though the local-file system access control on the file server where the data is stored is set up properly, the mapping also needs to make sure configurations in other places where people can indirectly access the data, such as from a web browser, are also set up in a way consistent with the high-level policy. Ioannidis's work [24] extensively studies the problem of consistent policy enforcement in a heterogeneous environment. The approach is a combination of compile-time and run-time checking. Two applications were described based on the approach — Virtual Private Services [25] and Cannon. These works aim at an overhaul of the security management today, which is often done in an ad hoc way across different layers in a system. We call this approach the *architectural approach*, because its application requires changing the architecture of network security management.

The work described in this thesis tries to improve security management from another angle. Instead of creating a new structure to replace what is commonly used today, we take the existing systems, model them formally, and analyze the security interactions in logic. The policy serves a different purpose here: instead of deriving low-level configurations from the policy, we *validate* the configuration against the policy. We call this approach *validation approach*. It does not require changes to the current security management framework, but adds an extra validation system to make sure high-level security goal will not be violated. Compared to the architectural

approach, the validation approach is easier to deploy in practice because people do not have to change the way they manage the network. However, in the long term, both approaches are essential to build a secure network. The validation approach will provide useful inputs for designing a better security architecture and gradually change the way people manage networks. Even after an architectural overhaul, it is still important to validate the new architecture formally to make sure it really meets the security needs.

It is important to note that the purpose of MulVAL policy is also different from some of the well-known security policy languages, such as PolicyMaker [9], KeyNote [8], SD3 [28], and Binder [15]. These policy languages are intended to be used to specify access control in distributed environment, or *trust management* (TM) [9]. In general, the safety property of a TM policy is hard to verify [32]. The security analysis discussed in this disseration does not address the analysis of TM, and policy used does not have features such as delegation in TM. Incorporating TM in the analysis is left for future work.

1.6 Contributions

In this thesis, I proposed a logical approach to network security analysis. We use Datalog — a logic programming language — as a uniform language to represent all relevant information needed in the reasoning. These include:

- Reasoning logic that captures generic security interactions, such as common attack scenarios, operating system semantics, and network traffic flow.

- Formal software vulnerability advisories that specifies the pre- and postconditions of exploits.

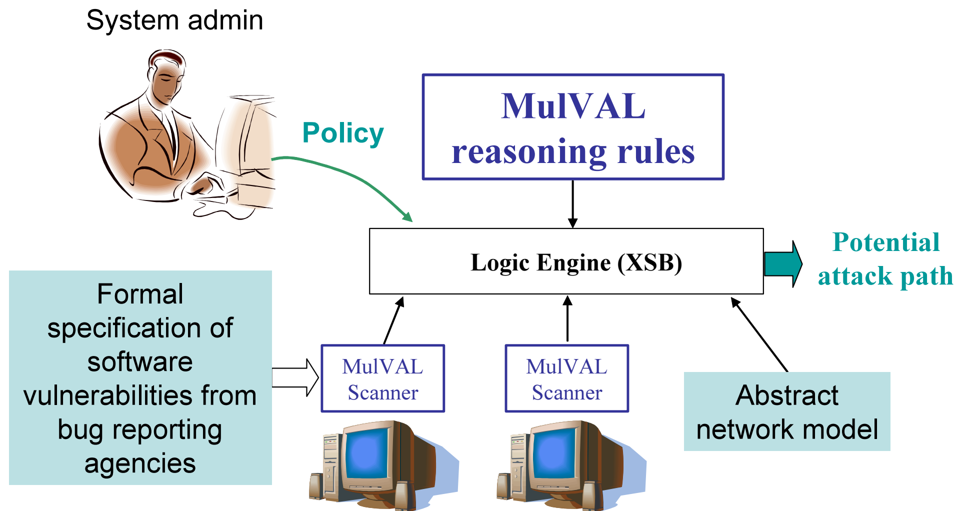

Figure 1.7: MulVAL Framework

- Output of host scanners that specify security-relevant configuration information.
- Output of network management tools that specify high-level network model.
- Security policies defined by system administrators that specifiy high-level goals of administration in the local site.

These information, formally specified in Datalog, can be put together in a standard logic-programming engine that can evaluate Datalog efficiently. The logic engine can then conduct exhaustive search to find out all possible multistage, multihost attack paths due to all possible interactions in the network. The framework is shown in Figure 1.7. The contribution is summarized as follows.

1. I have proposed a logic-programming approach for specifying and analyzing complex interactions among network elements, which has the advantages of clear specification, efficient execution, and expressive programming;
2. I have designed a formal model for reasoning about security interactions in networks of Unix-family machines; the formal model integrates information found

in existing vulnerability databases to provide exploit semantics, eliminating the need to manually provide them whenever a vulnerability is reported;

3. I have designed an end-to-end system, MulVAL (Multihost, multistage Vulnerability AnaLysis) [34], that incorporates OVAL vulnerability scanners and conduct security analysis on the network level;

4. I have designed and implemented algorithms for conducting various kinds of analysis in the MulVAL framework: checking network configurations against a high-level policy specification that captures data confidentiality and integrity, hypothetical analysis that assumes various vulnerability situations, and the generation of attack trees.

The major advantage of the logic-programming approach is its clear specification of reasoning logic and the separation of reasoning logic and the implementation of the reasoning engine, where the latter can be just a standard Prolog system. Clear specification makes it easier to incorporate third-party security knowledge, such as exploit semantics in vulnerability advisories. Such information has to be input manually in some existing vulnerability analysis tools, such as TVA (Topological Vulnerability Analysis) [27]. Since any security analyzer inevitably has false positives and false negatives, the clear sepecification of reasoning model in a formal logic makes it easier for the security community to audit, discuss, and augment the reasoning model and improve its accuracy and effectiveness over time, making it a viable approach to thwart the ever growing security threats that accompanies the ever growing use of computer networks.

Chapter 2

Formal model of reasoning

MulVAL adopts Datalog [11] as the language to model network elements and their interactions. We first review some terminologies.

2.1 Datalog review

Syntactically, Datalog is a subset of Prolog [12] with limited forms of 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 Prolog, a variable is an identifier that starts with an upper-case letter. A constant is one that starts with a lower-case letter. Let $L_0, L_1, ...L_n$ be literals, a Horn clause in Datalog has the form:

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

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*. A significant difference between Datalog and Prolog is that Datalog has a pure declar-

ative semantics. The order of clauses in a Datalog program is irrelevant to its logical meaning and evaluation result. Whereas in Prolog such order is important and affects the result of evaluation [12], due to the depth-first search strategy and side-effect operators like "cut".

Datalog is often used in *deductive databases*. In such settings, data tuples in the database are represented as Datalog facts, and the deductive engine is implemented as a Datalog program that runs on the inputs from the database. The Datalog facts representing the original database are called the *extensional database* (EDB), and the Datalog facts computed by the deductive engine are called the *intensional database* (IDB). The complexity of computing whether a literal is implied by a Datalog program from EDB input (*i.e.* whether the literal is in IDB) is polynomial in the size of the EDB [14]. In this dissertation, we call an EDB predicate a *primitive predicate* and an IDB predicate a *derived predicate*.

Datalog has also been used as a security language for expressing access control policies [15, 31]. The declarative semantics of Datalog makes specifying concepts such as delegation straightforward. The efficiency of Datalog and existing off-the-shelf Datalog evaluation engines [41, 51] make such languages readily usable in practice.

There are many advantages of using Datalog as the formal model of reasoning in the security analysis discussed in this dissertation. Compared with the exploit-dependency graph, Datalog is a formal declarative logic language, which provides a clear specification. Like in the model-checking approach, one can leverage an off-the-shelf logic engine to conduct the analysis. But unlike model-checking, the execution time of a Datalog program is polynomial in the size of data inputs. Logic engines have been optimized over decades to handle large datasets efficiently, which makes Datalog particularly suitable for analyzing security of large and complex networks.

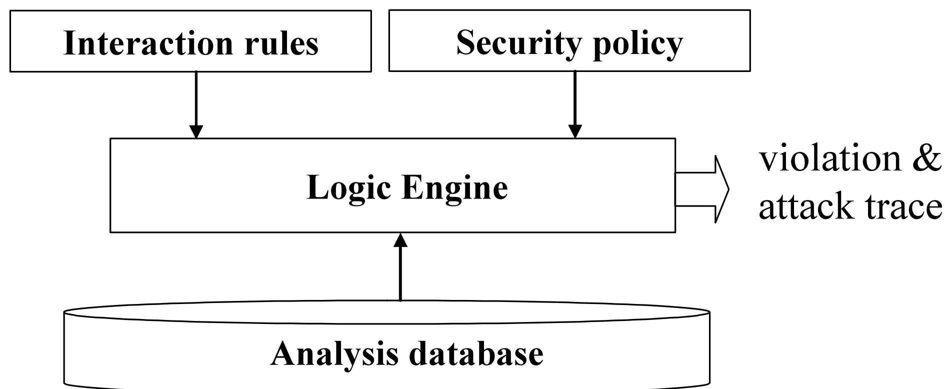

Figure 2.1: Analysis framework

2.2 Analysis framework

The MulVAL core analysis framework is shown in Figure 2.1. An *analysis database* is a collection of Datalog facts that represent the status of the network and the advisory information about software vulnerabilities. Chapter 3 will discuss in detail how to populate this database. The interaction rules are Datalog clauses that specify how different pieces of a network can interact and affect security. These are reasoning rules that can simulate what an attacker can do in the network, given the configuration information in the analysis database. The security policy specifies the ultimate property a system administrator wants to keep for the network. In MulVAL, the policy is simple Datalog tuples that list legal data accesses by principals.

2.3 Interaction rules

MulVAL interaction rules specify the semantics of: different kinds of vulnerabilities and their exploits, normal software behaviors that affect security, and multihop network access. Many of those rules are operating-system specific. The rules discussed in this dissertation apply to the Unix-family operating systems. Currently

there are about 20 rules in MulVAL. The MulVAL rules are carefully designed so that information about specific vulnerabilities is factored out. The interaction rules characterize general attack methodologies (such as "remote exploit of a buffer overrun bug," or "Trojan Horse client program"), not specific vulnerabilities. Thus the rules do not need to be changed frequently, even though new vulnerabilities are reported frequently. The rules are also independent of specific configurations of a particular network setting and thus can be applied across different sites.

2.3.1 Types of constants

In Datalog, a term is either a constant or a variable. Datalog is an untyped language, so a predicate can be applied to arbitrary terms. However, to make a Datalog sentence meaningful, the arguments to a predicate should take value from certain domains. This section lists the types used in the Datalog interaction rules.

1. Host.

   In this disseration, a host is represented as a symbolic name, such as `webServer` and `fileServer`. In the real implementation, it is represented as an IP-address range.

2. Protocol.

   A transport or application layer protocol, such as `tcp`, `udp`, and `rpc`.

3. Port.

   A number differentiating different services within the same protocol.

4. Principal.

   A symbolic name representing a certain group of people, such as `employee` and `attacker`.

5. Data.

   A symbolic name representing an abstract notion of a data item, such as `webPages` and `projectPlan`.

6. String.

   Single-quoted strings are used to represent file-system paths, vulnerability identification numbers, *etc.*

7. Exploit range.

   Either `localExploit` or `remoteExploit`.

8. Exploit consequence.

   One of four possibilities: `confidentiality`, `integrity`, `privilegeEscalation`, and `dos`.

9. Program

   The name of a program on a system, such as `httpd`.

10. User/Group

    The name of a user or group on a system.

11. Access

    `read`, `write`, or `exec`.

The next several sections describe interaction rules that capture various aspects of attack scenarios and operating-system semantics that affect security.

2.3.2 Vulnerability rules

A vulnerability is an unintended behavior in a software system that can be utilized by an attacker to compromise the security of a host. Following are the predicates involved in rules about vulnerabilities. The arguments of the predicates are presented as variables, although in a specific rule they can either be a variable or a constant.

`vulExists(Host, Program, ExploitRange, ExploitConsequence)` is a derived predicate specifying that a vulnerability exists in the `Program` on a `Host`, and it has specific `ExploitRange` and `ExploitConsequence`. This is a derived predicate. `Program` is the full path of the executable that contains the security bug. `ExploitRange` is either `local` or `remote`, indicating if the bug is locally exploitable or remotely exploitable. Two common values for `ExploitConsequence` are `privilegeEscalation`, meaning a successful exploit would enable an attacker to execute arbitrary code, and `dos`, meaning the attacker can crash the program (denial of service).

`vulExists(Host, ID, Program)` is a primitive predicate specifying that a vulnerability with identification `ID` exists in the `Program` on the `Host`.

`vulProperty(ID, ExploitRange, ExploitConsequence)` is a primitive predicate that specifies the exploitable range and consequence of the vulnerability with `ID`.

`bugHyp(Host, Program, Range, Consequence)` is a *dynamic* predicate that introduces a hypothetical bug in a `Program` on the `Host` which has `ExploitRange` and `ExploitConsequence`. More details about using dynamic predicates to conduct hypothetical analysis is discussed in Chapter 5.

`dependsOn(Host, Program, Library)` is a primitive predicate specifying that a `Program` on a `Host` depends on a `Library`, where the type of `Library` is also "Program".

Following are the rules computing vulnerablity information on a host. `%%` introduces a line of comment.

```
vulExists(H, Prog, Range, Consequence):-                    %%Advisory
                vulExists(H, ID, Prog),
                vulProperty(ID, Range, Consequence).

vulExists(H, Prog, Range, Consequence):-                    %%Hypothetical Bug
                bugHyp(H, Prog, Range, Consequence).

vulExists(H, Prog, Range, Consequence):-                    %%Library Bug
                vulExists(H, Library, Range, Consequence),
                dependsOn(H, Prog, Library).
```

2.3.3 Exploit rules

We first introduce several predicates that are used in the exploit rules.

`execCode(P,H,UserPriv)` is a derived predicate specifying that principal `P` can execute arbitrary code with privilege `UserPriv` on machine `H`.

`netAccess(P, Src, Dst, Protocol, Port)` is a derived predicate specifying that principal `P` can send packets from machine `Src` to `Port` on machine `Dst` through `Protocol`.

`networkService(H, Prog, Protocol, Port, User)` is a primitive predicate specifying that a service program `Prog` is running on host `H` as user `User`. It is listening on port `Port` of protocol `Protocol`. For example, `networkService(webServer, httpd, tcp, 80, apache)` means on machine `webServer`, a network service program `httpd` is running as user `apache` and listening on port `80` of the `tcp` protocol.

`setuidProgram(H, Prog)` is a primitive predicate specifying that `Prog` is a setuid[1]

[1] In a Unix system, a setuid program will have the privilege of the owner when executed.

program on host `H`. The executable file is owned by `User`.

`clientProgram(H, Prog)` is a primitive predicate specifying that `Prog` is a client program that when executed, may open a connection to a server over the network.

`malicious(P)` is a primitive predicate specifying that principal `P` would attack the network system to gain illegal privilege.

`incompetent(P)` is a primitive predicate specifying that principal `P` is not careful in using computers and its behavior may be utilized by a malicious attacker.

In theory, the preconditions for exploiting a particular software bug may be arbitrary. In practice, the vast majority of exploits happen in very similar manners. Most security bugs are caused by buffer overflows, where a malicious attacker construct a specially-crafted input that can overrun the memory boundary the program's stack or heap. By doing so the attacker can inject code into the memory and modify the return pointer in the stack to cause the program to jump to the injected code, which may be a shell program that will allow the attacker to execute arbitrary code. Even if the injected code cannot be executed, the attacker can still crash the program and thus cause a denial of service. If the input of the buggy program comes from the network, this kind of bug can be exploited remotely (called *remote privilege escalation*). Otherwise the attacker will need to first have some local privilege on the machine where the program is running. If the program is a *setuid* program, executing the program locally on a malicious input may enable the attacker to gain root(called *local privilege escalation*). Following are the two rules for remote and local privilege escalation.

```
execCode(Attacker, Host, User) :-            %%Rule-remote-privilege-escalation
          malicious(Attacker),
          vulExists(Host, Program, remoteExploit, privilegeEscalation),
          networkService(Host, Program, Protocol, Port, User),
```

```
        netAccess(Attacker, _AttackSrc, Host, Protocol, Port).
```

That is, if `Program` running on `Host` contains a remotely exploitable vulnerability whose consequence is privilege escalation, the buggy program is running as `User` and listening on `Protocol` and `Port`, and an attacker can send malicious packets to the service through the network, then the attacker can execute arbitary code on the machine as `User`. This rule can be applied to any vulnerability that matches the pattern. An underscore-led variable such as `_AttackSrc` is an anonymous variable in Datalog — one that appears only once in a clause, and thus whose value does not matter. In this rule, it indicates that the service program accepts packets from any client machine so one can launch an attack from any host that can send a packet to the server. This is a conservative approximation because some network services can restrict network accesses to certain client hosts, for example through TCP wrappers [50]. In such cases a more precise rule would need to specify the valid clients instead of using a wild cast.

```
execCode(Attacker, Host, User) :-              %%Rule-local-privilege-escalation
          malicious(Attacker),
          vulExists(Host, Prog, localExploit, privilegeEscalation),
          setuidProgram(Host, Prog),
          fileOwner(Host, Path, User),
          execCode(Attacker, Host, _SomeUser).
```

That is, if a malicious attacker can first compromise an account (`_SomeUser`) on a machine, and there is a locally exploitable privilege-escalation bug in a setuid program owned by `User`, then the attacker can gain the privilege of `User`. Again, the anonymous variable `_SomeUser` brings a conservative approximation into the rule. If the local user whose account is compromised by the attacker cannot execute the

setuid program in the first place, the exploit cannot happen at all. A more precise rule would include the executable check for the file `Prog`.

Another kind of exploit happens at the client side of a network application. A client program is one that establishes communication with a server, such as a web browser, a mail client, or a messenger program. In order to exploit a bug in those programs, the victim must first initiate a connection to a server that may provide malicious inputs from the attacker. For example, a compromised web page may contain Java programs that exploit the Java Virtual Machine in Internet Explorer, a mail server may deliver mails that contain worms exploiting vulnerabilities in Outlook Express, a MSN messenger server may pass on malicious requests trying to exploit vulnerabilities in a MSN client. For those exploits to happen, the victim needs to do something first: browse a compromised webpage, click a link in a malicious email, open the messenger client, and so on. A careful user can avoid such exploits by exercising precaution. For example, he never browses a webpage from unknown sources, he never clicks attachments in an unsolicited email, and he blocks messenger requests from unknown messenger users. We classify these careful people as "competent" users. Usually system administrators are considered competent. But an ordinary employee using a computer is considered "incomptetent." This classification can be provided by the system administrator and reflected in the security policy.

Following is the exploit rule for remote exploit of a client program.

```
execCode(Attacker, Host, User) :-             %%Rule-exploit-remote-client
          malicious(Attacker),
          vulExists(Host, Program, remoteExploit, privilegeEscalation),
          clientProgram(Host, Program),
          incompetent(P),
          hasAccount(P, User).
```

The body of the rule specifies that 1) the `Program` is vulnerable to a remote exploit; 2) the `Program` is a network client application; 3) an incompetent principal `P` has account `User` on the machine. The consequence of the exploit is that the attacker can execute arbitrary code with the privilege of the incompetent user.

After an exploit is successfully applied, an attacker can gain further privilege by exploiting more vulnerabilities. For example, after he compromises a local user account through a successful remote exploit, he can further exploit a local bug to become root. He can also do other things allowed by the operating system. For example, he can read or modify files on the machine or launch attacks to other machines from there. The following sections explain interaction rules that capture these scenarios.

2.3.4 File access

The following predicates are used in computing the file access a principal can have on a Unix machine.

`accessFile(P, H, Access, Path)` is a derived predicate specifying that principal `P` can `Access` the files specified by `Path` on machine `H`. `Access` can be either `read`, `write`, or `exec`.

`localFileProtection(H, User, Access, Path)` is a derived predicate specifying that the `User` on machine `H` can have the specified `Access` to the file `Path`.

`fileAttr(H, Path, R1,W1,X1,R2,W2,X2,R3,W3,X3)` specifies the UNIX file attribute bits. For example, if on machine `workStation` the file `/home/projectPlan.pdf`'s attribute is `rw-r-----`, the corresponding predicate would be

```
fileAttr(workStation, '/home/projectPlan.pdf', r,w,0,r,0,0,0,0,0).
```

Note that `r` and `w` are interpreted as `1` for the corresponding access bits.

The following rule says if an attacker `P` can execute arbitrary code on machine `H` with `User`'s privilege, he can have whatever access `User` has to files.

```
accessFile(P, H, Access, Path) :- execCode(P, H, User),
                                  localFileProtection(H, User, Access, Path).
```

Following are rules for computing file access rights on a UNIX system.

```
localFileProtection(H, User, Access, Path) :-
                 fileOwner(H, Path, User),
                 ownerAccessible(H, Access, Path).

localFileProtection(H, User, Access, Path) :-
                 inGroup(User, Group),
                 fileGroupOwner(H, Path, Group),
                 groupAccessible(H, Access, Path).

localFileProtection(H, User, Access, Path) :-
                 worldAccessible(H, Access, Path).

ownerAccessible(H, read, Path) :-
                 fileAttr(H, Path, r,_,_,_,_,_,_,_,_,_).

groupAccessible(H, read, Path) :-
                 fileAttr(H, Path, _,_,_,_,r,_,_,_,_,_).

worldAccessible(H, read, Path) :-
                 fileAttr(H, Path, _,_,_,_,_,_,_,r,_,_).

ownerAccessible(H, write, Path) :-
                 fileAttr(H, Path, _,w,_,_,_,_,_,_,_,_).
```

```
groupAccessible(H, write, Path) :-
                 fileAttr(H, Path, _,_,_,_,_,w,_,_,_,_).

worldAccessible(H, write, Path) :-
                 fileAttr(H, Path, _,_,_,_,_,_,_,_,w,_).

ownerAccessible(H, exec, Path) :-
                 fileAttr(H, Path, _,_,x,_,_,_,_,_,_,_).

groupAccessible(H, exec, Path) :-
                 fileAttr(H, Path, _,_,_,_,_,_,x,_,_,_).

worldAccessible(H, exec, Path) :-
                 fileAttr(H, Path, _,_,_,_,_,_,_,_,_,x).
```

The meaning of the rules are self-explanatory. The primitive predicate `fileOwner`, `fileGroupOwner`, and `fileAttr` can easily be gotten from the output of the “ls -l” command. The primitive predicate `inGroup` can be computed from the result of the “groups” command or from a directory database (such as OpenLDAP) if that is used by the system to maintain user and group information.

2.3.5 Trojan-horse programs

A Trojan-horse is a malicious program that masquerades as a benign application. For example, a Trojan-horse PDF file reader may communicate the content of the file to an adversary, and a Trojan-horse SSH client can steal the password or private key of a user. A Trojan horse can even install a back door on the system that allows the attacker to enter at a later time.

If the integrity of the file system on a machine is compromised by an attacker,

he can replace legitimate programs such as `Adobe Acrobat Reader` or `ssh` with his Trojan-horse version. When an innocent user executes such programs the attacker can gain privilege on the system:

```
execCode(Attacker, H, User) :- malicious(Attacker),
                               accessFile(Attacker, H, write, Path),
                               not setuidProgram(H, Path),
                               localFileProtection(H, User, exec, Path).

execCode(Attacker, H, Owner) :- malicious(Attacker),
                                accessFile(Attacker, H, write, Path),
                                setuidProgram(H, Path),
                                fileOwner(H, Path, Owner),
                                localFileProtection(H, User, exec, Path).
```

This is another example of conservative approximation in the interaction rules. Normally a user may inadvertantly execute a Trojan-horse program only if it is injected into a directory included in the user's "PATH" environment variable. But this rule specifies that if an attacker can modify files under any directory, he can gain the privileges of any user who can execute the code (even though it is not in the user's PATH diretories).

2.3.6 NFS semantics

There are also attacks that exploit normal software behaviors. For example, through talking to system administrators we have found that the security weaknesses in the NFS file-sharing system have contributed to many intrusions on our campus. NFS was not designed with security in mind, though now it is widely used and quite popular. Scripts to exploit the insecure file-sharing exist on the web and have been in use. NFS

is intended to be used in a local-area environment where the hosts trust each other, *i.e.* any file access request from a legitimate host on the server's export list is deemed valid. This host-based trust relationship works fine under normal conditions, but will cause severe problems when the network is facing malicious attacks. If the attacker can compromise one account on a client machine, he can send NFS requests to the server on behalf of any other user. Whether the requests will be honored depends on the NFS server's configuration. The semantics of NFS is specified by the following Datalog rules:

```
accessFile(P, Server, Access, Path) :-
        malicious(P),
        netAccess(P, Client, Server, rpc, 100003),
        nfsExportInfo(Server, Path, Client, Access, RootSquash, insecure),
        nfsUserMap(UserClient, UserServer, RootSquash),
        localFileProtection(Server, UserServer, Access, Path).

accessFile(P, Server, Access, Path) :-
        malicious(P),
        netAccess(P, Client, Server, rpc, 100003),
        execCode(P, Client, root),
        nfsExportInfo(Server, Path, Client, Access, RootSquash, secure),
        nfsUserMap(UserClient, UserServer, RootSquash),
        localFileProtection(Server, UserServer, Access, Path).

nfsUserMap(User, User, no_root_squash).

nfsUserMap(root, anonymous, root_squash).

nfsUserMap(User, User, root_sqush) :-
        nonvar(User),
```

```
    non_root_user(User).
```

NFS protocol is based on RPC (remote procedure call) and its assigned RPC number is 100003. The primitive predicate `nfsExportInfo` specifies the configuration on an NFS server — which path is exported to which client machine and under what options. In the first rule, the "secure" option of the exported share is off, which means a request does not have to come from a privileged port on the client machine [2] to be accepted by the server. In the second rule the "secure" option is on, so the attacking principal `P` must first get root on the client machine. The `nfsUserMap` maps a user in an NFS request to a user on the server. When the "root_squash" option is on, the root user will be mapped to the user `anonymous`. Otherwise the user id is be unchanged after the mapping.

Predicate `non_root_user` invokes a library function that tells if a user id is "root". From a formal point of view, a library function can be considered as a primitive predicate with a different physical realization than ordinary EDB predicates: they are not explicitly stored in the database but are implemented as procedures which are computed during the evaluation of a Datalog program. Before calling such a library, however, a ground check should be performed to guarantee the safety property [11], since calling the library with an uninstantiated variable could return an infinite number of results (*e.g.* there are infinite number of non-root user names). Such unsafe programs will typically throw an exception in the library function when evaluated.

On an NFS client machine, the remote file shares are mounted in the local file system so that legitimate users can access files on the server as if they are on the client machine. The security implication of mount is that if an attacker can compromise the integrity of files on an NFS server, the users on the client machine will be affected.

[2] A privileged port is one under 1024. Only the root user can bind to privileged ports.

Following is the rule for the semantics of mount.

```
accessFile(P, Client, Access, ServerPath) :-
        nfsMounted(Client, ClientPath, Server, ServerPath, Access),
        accessFile(P, Server, Access, ClientPath).
```

That is, when a principal accesses files in a portion that is exported to a client machine, he is also, in some sense, "accessing" files on a client that mounted that portion. `nfsMounted` is a primitive predicate provided by the scanner, which specifies that a portion on `Server` is mounted on a local path of machine `Client`.

2.3.7 User credentials

A credential is some qualification that enables a user to access a system. Passwords and user keys are two examples. Credentials are often stored on a machine with certain protections. In Unix, user passwords are cryptographically hashed and the hash value is stored in the "shadow" file readable only to the root user. Private keys for SSH sessions are stored in the user's home directory with passphrase protection. Once an attacker gets local access to a machine, he can steal user credentials and compromise more machines. There are a variety of ways to steal user credentials on a machine. If an attacker becomes root on the machine, he can get the shadow file and conduct a dictionary attack to guess ill-chosen passwords. If an attacker compromises one user account on a machine, he can get the private key of the user stored in his home directory. If the key is not passphrase-protected, he can log in to any machine where the corresponding public key is included in the SSH configuration. The attacker can even install a keystroke logger or Trojan-horse SSH program to record the password or passphrase of the user. Even worse, a careless user may use the same password across different sites. If his account is compromised at one site, his accounts will

also be compromised at the other sites. An attacker can also extract an incompetent user's password by social engineering.

Since there are so many situations for a user's credential to be compromised by an attacker, it may not be a good idea to precisely model the differences between all these scenarios. Rather, an approximate model can serve our purpose reasonably well. Predicate `principalCompromised(P1, P2)` is introduced to express that principal `P1`'s credential is compromised by principal `P2` (the attacker). The following two rules specify under what condition a principal's credential is compromised.

```
principalCompromised(Victim, Attacker) :- hasAccount(Victim, H, User),
                                          execCode(Attacker, H, root),
                                          malicious(Attacker).

principalCompromised(Victim, Attacker) :- incompetent(Victim),
                                          malicious(Attacker).
```

In the first rule, principal `Victim` has an account on host `H`. Malicious `Attacker` takes full control of the host as root. In this case `Victim`'s credential will be compromised by `Attacker`. Once an attacker completely compromises a machine, he can get the shadow file of the system, get the private key file of every user, or install key-stroke loggers or Trojan-horse SSH client. With all likelihood the credentials of the machine's user will be stolen.

The second rule encompasses the situations where an attacker does not need to get root to steal a credential. In this case the victim is "incompetent" from a security perspective. He may succumb to social engineering, use a easy-to-guess password, or not passphrase-protect his private-key file. For such a user, his account could be compromised at any time. So there is no other precondition for the rule.

The above two rules are conservative in the sense that what is inferred is the worst-

case scenario. Even if the root of a machine is compromised, a user's credential is not necessarily compromised if he excercises extreme vigilance and is lucky. Similarly, just because a person is classified as "incompetent" by the system administrator does not necessarily mean his credential must be compromised by a malicious attacker. Conservative approximation in the interaction rules leads to false positives in the analysis result. However, it simplifies the reasoning process and the resulting attack tree, leaving it to the system administrator to decide whether the attack tree is realistic.

Once the credential of a principal is compromised, an attacker can compromise accounts of the principal on any machine the attacker has access to.

```
execCode(Attacker, Host, User) :- principalCompromised(Victim, Attacker),
                                  hasAccount(Victim, Host, User),
                                  canAccessHost(Attacker, Host).

canAccessHost(P, H) :- execCode(P, H, _SomeUser).

canAccessHost(P, H) :- logInService(H, Protocol, Port),
                       netAccess(P, _AttackSrc, H, Protocol, Port).

logInService(H, Protocol, Port) :- networkService(H, sshd, Protocol, Port, _User).
```

To use the stolen credential to compromise another host, an attacker needs some access to the machine. Either he can get into the machine as another user and then execute the "su" command, or he is able to access the login service (such as `sshd`) on the host. The two rules for `canAccessHost(P, H)` specify these two cases.

2.4 Network topology

The packet flow in the network affects an attacker's ability to launch attacks. Packet flow is controlled by firewalls, routers, switches, and other aspects of network topology. MulVAL uses an abstract model, *host access control list* (HACL), to describe the topology of a network.

2.4.1 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 Datalog tuples of the following form:

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

It means machine `Source` can reach `DestPort` on machine `Destination` through `Protocol`. HACL is an abstraction of the ultimate effects of the physical topology, firewall rules, the configuration settings of routers and switches, and so on. It is compatible with the high-level specification used in many automatic network management tools [22, 23, 5, 10]. Those tools can be leveraged to provide this information.

Configuration aspects not captured by HACL

In a large enterprise network, many features of network configurations affect security. Following are some aspects not captured by HACL. We leave the modeling of these features as future work.

Spoofing is a common attack methodology. For exmaple by spoofing the source address of an NFS request packet an attacker can trick an NFS server into believing the request comes from a legitimate export client. Some network configuration mechanisms, such as access-control lists on switches, can prevent address spoofing to an

extend. In order to capture this configuration information in HACL, each `hacl` entry needs to be augmented with another field indicating the "true" source of packets.

Another aspect not captured by HACL is which zone a particular packet may pass through. This is relevant in eavesdropping attacks. If a company allows insecure communication inside the corporate network, it would be important to figure out whether an attacker can sniff sensitive information once he gets hold of one machine inside the enterprise network. However, HACL only specifies the end points of communication; the intermediate steps are not captured. One possible way to model this is by attaching to each `hacl` entry a list of zones the packet is allowed to traverse.

2.4.2 Multihop host access

Predicate `netAccess(P, Src, Dst, Protocol, Port)` specifies that principal `P` can send network packets from machine `Src` to `Port` on host `Dst` through `Protocol`. Following are the rules deriving the predicate.

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

If a principal `P` has local access on machine `H1` as some `_User` 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 rule allows for reasoning about multihost attacks, where an attacker first gains access on one machine inside a network and launches further attacks from there.

2.5 Policy specification

The security policy is the only piece of information that a local administrator needs to provide. In MulVAL, a security policy specifies which principal can access what

data. Each principal and piece of data is given a symbolic name, which is mapped to a concrete entity by the *binding* information. Each policy statement is of the form *allow(Principal, Access, Data)*. The arguments can be either constants or variables. Following is an example policy:

```
allow(_Everyone, read, webPages).
allow(employee, _Access, projectPlan).
allow(sysAdmin, _Access, Data).
```

`_Everyone` and `_Access` are anonymous variables. The policy says anybody can read `webPages`; `employee` 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 simple and easy to be specified correctly. However, the MulVAL reasoning system can handle more complex policies as well. For example, in MulVAL one can use general Prolog as the policy language. More discussions on policies and policy check can be found in Chapter 4.

2.5.1 Binding information

The principal and data items mentioned in the MulVAL policy are just symbolic names. They are mapped to concrete entities by *principal binding* and *data binding*.

Principal binding maps a principal symbol to its user accounts on network hosts, or a network zone from which the principal operates. The format of the binding information is

hasAccount(Principal, Host, Account), or

located(Principal, Zone),

where *Principal* is the symbolic name for a principal, *Account* is the name of a user account on *Host* that the principal can access, and *Zone* is the network zone a principal may operate from. Examples:

```
hasAccount(employee, workStation, ralph).
hasAccount(employee, workStation, james).
hasAccount(sysAdmin, webServer, root).
located(attacker, internet).
```

The account associated with a user does not necessarily correspond to a concrete account on the machine. It may stand for a group of accounts that have the same level of privilege. For example,

```
hasAccount(employee, workStation, employeeAccount).
hasAccount(sysAdmin, webServer, root).
located(attacker, internet).
```

Here `employeeAccount` represents any ordinaray user accounts on the system.

The principal binding may also contain information describing user behaviors. These are the `malicious` and `incompetent` predicates mentioned before. Example:

```
incompetent(employee).
malicious(attacker).
```

Data binding maps a data symbol to its physical location in the network, typically a file path on a machine. The format of the binding is

dataBind(Data, Host, Path).

Path could be a directory, in which case the *dataBindDir* tuple will mean "all files under the directory are bound to data symbol *Data*":

dataBindDir(Data, Host, DirPath).

Example data bindings:

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

2.6 Discussion

2.6.1 Using negations in the model

Pure Datalog is monotonic, *i.e.* adding more clauses will only increase the number of facts that can be derived. This corresponds to the monotonicity of attacks, which means gaining more privileges only increases an attacker's ability to launch more attacks. However, if negation is allowed in literals, Datalog is no longer a monotone logic. The introduction of negation will also affect the complexity of executing Datalog programs [14]. However, having negations is sometimes useful. For example, suppose there is an action that an attacker can take only if a certain configuration option of a program is *absent*. This could be modeled by the following Datalog rule:

```
action(Host, attacker) :- option_absent(Host, program, op).
```

This will require the scanner to report the absence of a configuration option, which means the scanner must have knowledge of the universe of all possible option values for every program. If negation is allowed, the rule can be rewritten as:

```
action(Host, attacker) :- not option(Host, program, op).
```

By allowing negations on the primitive predicate, the scanner will only need to report the options of a program that are set, yielding a much simpler design. This kind of negations are *stratified*. In a Datalog program with stratified negations, there exists a partial order on predicates, such that one predicate may depend on the negation of another predicate only if the former is strictly less than the latter. Such form of negation does not affect the polynomial complexity of Datalog [14].

2.6.2 Nonmonotonic attacks

Even with negation, Datalog still cannot model general nonmonotonic attacks. An attack is nonmonotonic if postconditions produced by an attack step may inhibit an attacker from launching more attacks. One example is attacks that compromise availability. An attacker can bring down a file server, but then he will not be able to install a Trojan horse on the fileserver and lure somebody to execute it. Such scenarios cannot be correctly modeled by Datalog, even with negation. Suppose the predicate `dos(H)` (denial of service) means the availability of host `H` is compromised. One naive attempt is to model the above scenario as the following rule.

```
execTrojanHorse(victim, H) :- trojanHorseInstalled(attacker, H),
                              not dos(H).
```

A victim may execute a Trojan-horse program on host `H`, if a Trojan-horse program is installed by `attacker` on `H`, and host `H` is still alive (*i.e.*, is not under denial of service attack). However, this is not a correct characterization of the precondition. `not dos(H)` means "the attacker cannot possibly cause a denial-of-service condition on `H`", whereas what we really want to say is "the attacker does not have to cause a denial-of-service condition on `H` to install the Trojan horse". Clearly the two are not equivalent.

Another example of nonmonotonic attack is when an attack step requires the absence of certain configuration options. If the option is set initially in the system, but the attacker can remove it, the attack can still succeed. However, in Datalog if something is known to be true it is impossible to introduce its negation without making the reasoning logic inconsistent. So one cannot model such scenarios directly in Datalog with negations.

The problem here is that under nonmonotonic attacks, the individual attack paths

become important. Conditions may change either way on a path: from false to true or from true to false. Model checking can conduct reasoning at the per-path level, although this introduces too much overhead for the more common monotonic attacks. For the simple nonmonotonic scenarios like the ones mentioned above, simple modification in the attack rules and some conservative approximation will suffice.

For the example of Trojan-horse installation, a conservative approximation of the rule looks as follows.

```
execTrojanHorse(victim, H) :- trojanHorseInstalled(attacker, H).
```

That is, we assume host `H` will be available after the attacker installed a Trojan-horse program on it. The attacker just chose not to compromise the availability of `H`, unless his ability to install the Trojan horse *depends* on his bringing down machine `H` in the first place. In that case, the rule will derive more than the attacker can actually achieve, which is a conservative approximation.

For the example of attacker changing configuration options, we use the example introduced in the last section and modify it as follows.

```
action(H, attacker) :- not_option(H, program, op).


not_option(H, Prog, Op) :- not option(H, Prog, Op).
not_option(H, Prog, Op) :- option_removed(H, Prog, Op).
```

That is, we introduce a new predicate, `not_option`, to represent the attacker's ability to make an option nonexistent. The two possibilities are: 1) the option is not set initially; 2) the option is removed by the attacker.

In summary, nonmonotonic attacks require more fine-grained analysis on individual attack paths and the temporal relations among attack steps. However, by conservative approximation many cases can still be modeled in Datalog.

Chapter 3

Analysis database

Chapter 2 described the Datalog rules for computing IDB (derived) predicates from the facts in the analysis database. These facts are represented by EDB (primitive) predicates in the following categories.

- Facts about software vulnerabilities (such as `vulExists` and `vulProperty`);
- Facts about machine configuration (such as `exports`, `fileAttr`, and `networkService`);
- Facts about network topology (host access control list);
- Facts about principal and data (the binding information).

This chapter discusses in detail where and how to obtain these Datalog tuples.

3.1 Vulnerability specification

Due to the large volume of security-relevant software bugs these days, many vulnerability-reporting agencies are starting to provide formal specification for reported vulnerabilities. Formal, machine-readable specification, along with tools that process them

automatically, makes it easier to exchange vulnerabilty reports and deal with them in a timely manner. The specification of a vulnerability consists of two parts: recognition specification and semantics specification.

3.1.1 Recognition specification

The recognition specification describes a set of machine configuration tests that, if true, show the existence of the vulnerability on the tested system. Multiple languages exist for this purpose, such as the *Nessus Attack Scripting Language* [6] (NASL), and the *Open Vulnerability Assessment Language* [52] (OVAL). MulVAL uses OVAL for recognition specification. An OVAL interpreter can take OVAL definitions and scan a machine for vulnerabilities. Figure 3.1 is an example OVAL definition for a vulnerability in the program "Ethereal," a network protocol analyzer.

Three tests must be performed and all must be true to conclude that the vulnerability exists on the machine. The three tests are enumerated in the `<criteria>` element and specified in the `<tests>` element. Each test consists of an `object` field, which identifies the piece of configuration information to be checked, and a `data` field, which specifies the value of the configuration parameter that will make the test true. An OVAL scanner will perform these tests according to the specification and output the result. In the example, if all three tests are true, the vulnerability, identified by CVE number `CVE-2003-0081`[1], exists on the machine being tested. Like the input, the output is also in a formally defined XML schema (OVAL Result Schema). It is easy to extract the test result from the output and transform it into Datalog tuples of the following form:

```
vulExists(Host, CVE_Id, Program).
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<oval xmlns="http://oval.mitre.org/XMLSchema/oval"  ...>
    ...
    <definitions>
        <definition id="OVAL54" class="vulnerability">
            <affected family="redhat">
                <redhat:platform>Red Hat Linux 9</redhat:platform>
                <product>Ethereal</product>
            </affected>
            ...
            <description>Format string vulnerability in packet-socks.c of
                         the SOCKS dissector for Ethereal 0.8.7 through
                         0.9.9 allows remote attackers to execute
                         arbitrary code via SOCKS packets containing
                         format string specifiers.
            </description>
            <reference source="CVE">CVE-2003-0081</reference>
            ...
            <criteria>
                <software operation="AND">
                    <criterion test_ref="rrt-201"
                               comment="Red Hat 9 is installed" />
                    <criterion test_ref="rut-201"
                               comment="ix86 architecture" />
                    <criterion test_ref="rvt-206"
                               comment="ethereal version is less
                                        than 0.9.11-0.90.1" />
                </software>
            </criteria>
        </definition>
    </definitions>
    <tests>
        <rpminfo_test id="rrt-201" check="at least one"
                      comment="Red Hat 9 is installed"
                      xmlns="http://oval.mitre.org/XMLSchema/oval#redhat">
            <object>
                <name operator="equals">redhat-release</name>
            </object>
            <data operation="AND">
                <version datatype="int" operator="equals">9</version>
            </data>
        </rpminfo_test>
        ...
    </tests>
</oval>
```

Figure 3.1: An OVAL definition

3.1.2 Semantics specification

The semantics specification describes the precondition to exploit the vulnerability and the consequence of the exploit. Unlike the recognition specification, the semantics specification of software vulnerabilities is far less developed and there is currently no common standard for the data format. The reason behind this is that there has not been much effort in the automatic reasoning of the effects of reported software vulnerabilities, even though there has been much work in the automatic recognition of them. Previous work in network vulnerability analysis relies on manually built exploit models, but it is infeasible to require the users of the tool to write down the pre and postconditions for the exploit of every reported bug. One major problem this thesis tries to solve is how to automatically incorporate the vulnerability semantics information from existing sources, eliminating the need for local administrators to provide the exploit semantics. If this attempt proves to be successful, more rigorous and formal semantics specification may emerge in the future that better suits the need of automatic vulnerability analysis.

Currently, some vulnerability databases contain information about the semantics of reported software security bugs. NVD (National Vulnerability Database), developed by the National Institute of Standards and Technology, is such a vulnerability database. Among the information provided by NVD, two attributes are related to the semantics specification: *exploitable range* and *loss type*.

Exploitable range

A vulnerability can enable either a "local" and/or "remote" attack. The definitions of local and remote attacks follow.

- *Local*:

Attacks that utilize the vulnerability can be launched directly on the system that is being attacked. The attacker must have some previous access to the system in order to launch an attack locally. An attack is still defined local if an attacker has a remote login shell to a host and initiates an attack on that host, as long as it is targeted to a component only visible to logged in users.

- *Remote*:

  Attacks that utilize the vulnerability can be launched across a network against a system without the user having previous access to the system.

Exploitable range is related to the precondition of exploits. As discussed in Section 2.3.3, these preconditions are categorized according to the bug's exploitable range and the nature of the buggy program (client or server).

Loss Type

The loss types take the form of the traditional three security properties ("availability", "confidentiality", and "integrity"), plus a new category called "security protection".

- *Confidentiality*

  A vulnerability is given the "confidentiality" label if it enables an attack that can result in the leaking of sensitive information on a system.

- *Integrity*

  A vulnerability is given the "integrity" label if it enables an attack that can result in the modification of sensitive information on a system.

- *Availability*

A vulnerability is given the "availability" label if it enables an attack that directly inhibits a user (human or machine) from accessing a particular system resource. This is also called *denial of service*.

- *Security Protection*

  A vulnerability is given the "security protection" label if it enables an attack that gives the attacker privileges in a system that the he is not supposed to have. This is also called *privilege escalation*. The "security protection" label may appear by itself or in three other variations: "security protection (gain superuser access)" when the attack allows a hacker complete control of a system, "security protection (gain user access)" when the attack allows a hacker partial control over a system, "security protection (other)" when the attack gives the hacker some other privilege on the system .

The availability, confidentiality, and integrity attributes are included in a vulnerability description only if exercising the vulnerability directly violates one of these properties. For example, if a vulnerability can give an attacker increased privilege thereby allowing the attacker to violate availability, only the "security protection" attribute would be true. However, if a single vulnerability enables two different attacks (as is typical with buffer overflow vulnerabilities), one of which violates security protection and the other availability directly, then both attributes would be true.

The loss type of a vulnerability indicates the consequence of an exploit. The meaning of the first two loss types are quite vague. It is not clear what information on the system may be leaked or changed. Fortunately, the two most common loss types: privilege escalation and denial of service[2], have relatively clear meanings and

[2] About 70% of vulnerabilities in NVD are labeled with only "security protection" or "availability".

they are modeled in MulVAL reasoning rules. MulVAL does not distinguish the three variants of security-protection vulnerabilities — gain superuser, user, or other access. Whether an exploit will enable an attacker to gain superuser access or just normal-user access normally depends on the configuration of the buggy program. If the program is running as root, then a successful exploit can potentially lead to root compromise. Otherwise the attacker can only gain normal-user access. The gain-other-privilege label itself does not provide enough semantic information. As a conservative approximation MulVAL treats it uniformly with the other two.

For the two loss types that do not have clear semantic meaning, a simple change in the NVD database would make them more useful. For many vulnerabilities labeled with "confidentiality" or "integrity," a successful exploit would enable an attacker to read or write all files on the system that the vulnerable program has access to. The NVD database could provide two subcategories for those two loss types: "all" and "other", where "all" means the loss of confidentiality (integrity) of all information the vulnerable program can access, and "other" means some other unspecified information. Then we can design MulVAL reasoning rules to handle the "all" case.

Modeling privilege separation

Privilege separation is a technique used to contain and restrict the effects of programming errors [39]. A network service program that runs as root can dispatch an unprivileged child process to handle network inputs. A bug in the unprivileged process does not result in the compromise of the privileged process. Most remote exploits depend on sending a maliciously crafted packet to the service program to induce a buffer overflow. The return address of the program is overwritten by the buffer overflow, causing the program to jump to the code injected by the malicious packet, typically a shell program. Privilege separation makes such compromises very

difficult if not impossible, because buffer overrun will happen in the unprivileged process, whose compromise will give an attacker very limited access to the system[3].

The rule `Rule-remote-privilege-escalation` in Section 2.3.3 does not take into consideration if a program adopts privilege-separation technology. The following modified rule does so:

```
execCode(Attacker, Host, User) :-     %%Rule-remote-privilege-escalation
      malicious(Attacker),
      vulExists(Host, VulID, Program),
      vulProperty(VulID, remoteExploit, privEscalation),
      networkService(Host, Program, Protocol, Port, User),
      not privilegeSeparated(Host, Program),
      netAccess(Attacker, Host, Protocol, Port).
```

This modification to take into account privilege separation highlights extensibility feature of MulVAL interaction rules. Security is characterised by the interactions between attacks and defence. The emergence of new attack methodologies and deployment of new defense techniques are eternal ongoing processes. This requires that the interaction rules be capable of adapting to changes in the most recent security arena, as illustrated by the example of privilege separation. Before this method was adopted, a successful exploit would enable an attacker to gain the privileges of the buggy program's process, which is captured by the old rule. With the new technology, this is not necessarily the case. But the modification of the rule is simple: adding a new predicate to tell if a program is running in privilege-separation mode. The scanner needs to get relevant configuration information to compute this predicate. Other than that, no other rules need to be changed.

[3]Typically the child process's file system access is restricted via chroot() to an unused portion, making it very difficult to invoke an executable on the system.

Exploit consequences other than privilege escalation

Another common exploit conseqence is denial of service, or loss of availability. This often happens as a buffer-overflow attack compromises the integrity of the program stack or heap and causes the program to crash. Compared with privilege escalation, this is a less severe consequence. But if availability is important for an application this may still be a concern to the system administrator.

Confidentiality and integrity loss are two less common exploit consequences. There is a great variety of information which can be leaked or modified, but there is no formal specification for it. So the exploit of vulnerabilities with only these two consequences are not modeled by the current MulVAL interaction rules. If MulVAL's interaction rules were to model those consequences using the information in the current databases, it would have to make a very conservative approximation. For example, a confidentiality loss would infer that all information stored on a machine that is accessible by the buggy program can be leaked to the attacker.

If practice shows that these unmodeled consequences do play important roles in a nonnegligible portion of attack cases, we will need to introduce new MulVAL predicates and rules to model and reason about them. Like in the case of introducing privilege escalation, this will only be a local change to the reasoning system and thus not error-prone. However, some additional information about vulnerabilties may be required that does not exist in the current bug databases. We view this as a proposal process to the bug-reporting community as to what information should be reported in a formal, machine-readable format.

Getting the semantics information from a vulnerability database is as easy as writing a database query to extract the corresponding attributes. The result can then be transformed to Datalog facts such as:

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

3.2 Host configuration

Various configuration information on a machine is needed for MulVAL to conduct its analysis. Most can easily be gotten by executing certain OS commands or looking at certain configuration files. Following is a list of predicates for the host configuration information and the corresponding ways to get them. The first parameter of all the predicates is the name of the host whose configuration is described.

1. `networkService(Host, Program, Protocol, Port, User)`

   Specifies the protocol and port number under which a network service program is listening, as well as the user that owns the process. This Datalog tuple is converted from the output of the "netstat" command. Figure 3.2 is an example output of executing "netstat -l -p" on a Linux machine:

```
Proto Recv-Q Send-Q Local Address Foreign Address State  PID/Program name
tcp    0     0      *:32768       *:*             LISTEN 1596/rpc.statd
tcp    0     0      localhost.localdo:32769  *:*  LISTEN 1715/xinetd
tcp    0     0      *:42117       *:*             LISTEN 9703/rpc.mountd
tcp    0     0      *:sunrpc      *:*             LISTEN 1577/portmap
tcp    0     0      *:x11         *:*             LISTEN 1943/X
tcp    0     0      *:ssh         *:*             LISTEN 29414/sshd
tcp    0     0      localhost.localdoma:ipp  *:*  LISTEN 1786/cupsd
tcp    0     0      localhost.localdom:smtp  *:*  LISTEN 1735/sendmail: acce
tcp    0     0      *:959         *:*             LISTEN 9683/rpc.rquotad
udp    0     0      *:32768       *:*                    1596/rpc.statd
udp    0     0      *:nfs         *:*                    -
udp    0     0      *:32770       *:*                    -
udp    0     0      *:32771       *:*                    9703/rpc.mountd
udp    0     0      *:924         *:*                    1596/rpc.statd
udp    0     0      *:956         *:*                    9683/rpc.rquotad
udp    0     0      *:sunrpc      *:*                    1577/portmap
udp    0     0      *:631         *:*                    1786/cupsd
```

Figure 3.2: Sample output of "netstat" command

Each row in the output can be converted to a Datalog tuple of the form `netstat-l-p(H,Prot,Recv-Q,Send-Q,Local,Foreign,State,PID,ProgName).`

However, there is no information about the user who owns the process. This requires executing a "ps -Af" command, whose outputs can be similarly converted to the Datalog tuples of the following format:

`ps-Af(H,UID,PID,PPID,C,STIME,TTY,TIME,CMD).`

Then the predicate `networkService` can be computed by the following simple Datalog program:

```
networkService(H, Program, Protocol, Port, User) :-
    netstat-l-p(H, Prot, _, _, Local, _, _, PID, Program),
    ps-A(H, User, PID, _, _, _, _, _, _).
```

2. `setuidProgram(Host, Program)`

   This information can be gotten by executing the following command under the root directory of the file system.

   "find . -perm +4000 -exec ls -l \;"

3. `clientProgram(Host, Program)`

   This predicate specifies that `Program` is a client program on `Host`. A client program is a network application that establishes connection with a server. Web browsers, email clients, and instant messengers are all example client programs. Since `Program` is the full path of the executable, this predicate can be computed by matching the last component of the path to a pre-defined list of known client-program names, such as "firefox, pine, gaim...." Specifically, for each element in the list[4], the following command will find the executable path of the program on the host (`NAME` is the executable names in the list).

[4] A more efficient way is to use a single find command to handle all the elements in the list.

```
find . -type f -perm +111 -name $NAME -print
```

4. `fileAttr(Host, Path, RO,WO,XO,RG,WG,XG,RW,WW,XW)`

   `fileOwner(Host, Path, Owner)`

   `fileGroupOwner(Host, Path, GroupOwner)`

   These are the result of conducting an “ls -l” command on the file specified by `Path`. The 9 attributes correspond to the file permission bits in Unix. For example, if exectuing command “ls -l /home/projectPlan.txt” on host `workStation` gets the following result:

```
-rw-r--r--    1 xou      grad        0 Mar 19 14:46 projectPlan.txt
```

   Then the corresponding Datalog tuples are:

   `fileAttr(workStation, ’/home/projectPlan.txt’, r,w,-,r,w,-,-,-,-)`

   `fileOwner(workStation, ’home/projectPlan.txt’, xou).`

   `fileGroupOwner(workStation, ’home/projectPlan.txt’, grad).`

   It is not necessary to store the “ls -l” information of every file on a machine. Only “important” files need to be scanned. These are the files that have a data binding on them (*i.e.* they are mapped by a data binding tuple to a data symbol), files that are executable binaries (normally found in directories containing a component “bin” in a Linux systems), and other files whose attribute is security relevant. The scanner can be given a list of such directories and perform the following command on a Linux system:

   “find `DIR` -exec ls -l \;”,

   where `DIR` is the diretory being scanned.

5. `nfsExportInfo(Server, PATH, Client, Access, RootSquash, Secure)`

   This tuple represents the NFS configuration on a server. The configuration file of kernel-based NFS in Linux is "/etc/exports". Each line in the file represents an export relation that can be specified in Datalog. For example, a line in "/etc/exports" may look like

   ```
   /public webServer(rw,async)
   ```

   It means the `/public` directory on the NFS server is exported to host `webServer` with read and write access. The "root_squash" and the "secure" options are on by default, otherwise they have to be explicitly specified as "no_root_squash" and "insecure". After a change in the exports file, an "exportfs" command must be executed to push the changes to the share table, which is typically located at "/var/lib/nfs/etab". Each row in this table contains all the options of an exported share in a fixed order and with explicit values. This table can be consulted to compute the `nfsExportInfo` predicate.

6. `nfsMounted(Client, ClientPath, Server, ServerPath, Access)`

   Specifies that `ServerPath` on `Server` is mounted on a `ClientPath` of machine `Client` as an NFS partition. This Datalog tuple can be gotten by executing the following command on the client machine:

   "df -t nfs -P"

   This predicate can also be computed from the raw content of "/etc/mtab".

What and when to scan Scanning machines is an important process in network security analysis. Since security is a complex problem, any piece of information about a machine can potentially be useful in determining an adversary's ability to carry out

attacks. However, it is infeasible to carry a complete image of a host for analysis, for reason of both scalability and privacy issues. A more practical approach is only to scan information required by the current interaction rules, and change the scanner when new attack methodologies emerge which need extra information for their analysis. Ideally, scanning is performed asynchronously on each individual host whenever its configuration changes in a significant way. When a new vulnerability report arrives, the analysis should be done on the data already gathered. However, under the current architecture, the recognition for vulnerability is done as part of the scanning process, not as part of the analysis on the database generated by the scanning. While this provides a better modularity (we leverage an off-the-shelf OVAL scanner to recognize vulnerabilities instead of writing our own OVAL-compatible vulerability recognizer), it also has several drawbacks. First, scanning of every host will be necessary whenever a new vulnerability report arrives. While the analysis in Datalog is fast, the scanning process takes much longer and does not scale in a large, wide-area network (see Chapter 6). Second, only the existence of a particular vulnerability on a machine will be visible from the attack tree generated by the MulVAL analysis engine. However, some recognition information of a bug, such as the version number of software installed on a host, configuration settings of a service proram, *etc.*, would be useful for system administrators to quickly understand what went wrong in the configuration. And third, the scanner needs to take inputs from third parties, increasing the chance of malicious inputs compromising machines. Thus, it will be advantageous to decouple the recognition process from the OVAL scanner and put it into the MulVAL analysis engine.

3.3 Network configuration

MulVAL models network configurations as abstract host access-control lists (HACL), which is a list of tuples of the following format:

`hacl(Source, Dest, Protocol, Port).`

It means machine in `Source` can reach `Dest` through a network service specified by `Protocol` and `Port`.

Study has been done in the past ten years in automatic network configuration mangagement [22, 23, 5, 10]. Some of this work has yielded tools that can manage routers, switches, and firewalls according to global policies similar to HACL [22, 5, 10, 26, 25]. MulVAL intends to integrate these tools as part of the information gathering process, providing the abstract HACL list automatically.

3.4 Binding information

In MulVAL the meanings of principal and data items in terms of concrete user accounts, file names, *etc.* are given by the principal binding and data binding information. Currently the binding information is part of the policy and must be provided manually by the system administrator. It is possible to put the principal binding information in an LDAP database and query it at the time of checking.

3.5 Putting everything together

Figure 3.3 illustrates the architecture of MulVAL with the data sources of the analysis database shown. MulVAL scanners, running on each individual host, provide machine configuration information. Smart Firewall [10] provides network configuration in terms of HACL tuples. LDAP provides principal binding information. The security

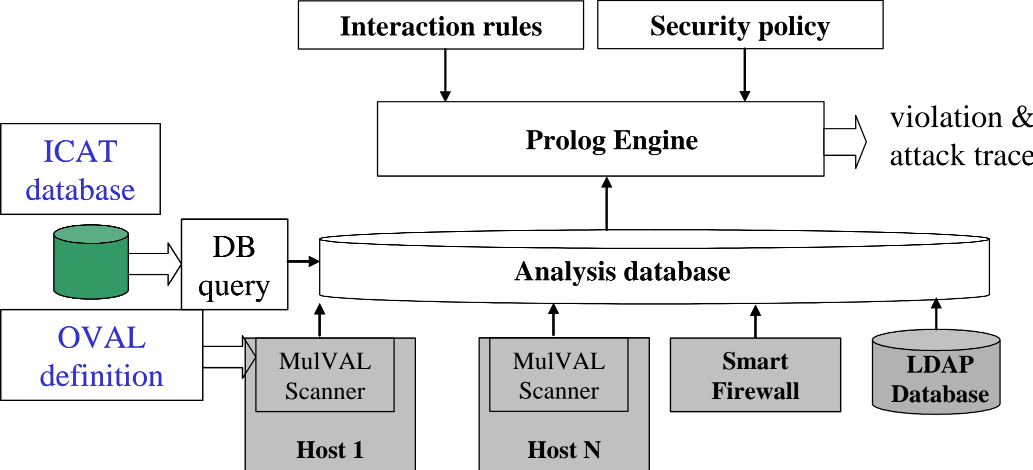

Figure 3.3: Complete MulVAL framework

policy is defined by the system administrator, who also needs to define data binding as part of the policy. Data sources on the left come from third-party bug-reporting agencies. They provide the formal specification of software bugs, in OVAL definitions and from the NVD database.

Chapter 4

Basic analysis

The reasoning model and the formal description of the configuration discussed in the previous two chapters provide a foundation for conducting various kinds of security analysis. One can view the interaction rules in the reasoning model as formalized expert knowledge on security interaction. The process of MulVAL analysis is in essense applying the security knowledge to configuration data and derive the properties of a network. This is a logical deduction process, and for the particular logical language used in MulVAL, the problem is reduced to Datalog evaluation. This chapter briefly reviews the evaluation strategies for Datalog and discusses two basic analysis — attack simulation and policy check.

4.1 Datalog evaluation and XSB

Datalog has two basic evaluation strategies: bottom-up evaluation and top-down evaluation [11]. In the bottom-up evaluation, the rules in the Datalog program are applied to the input facts in EDB to derive new facts in IDB, until no new facts can be derived. In the top-down evaluation, given a goal, the rules are applied backward

to find subgoals that must be true to satisfy the original goal, until all subgoals hit the input facts. Bottom-up evaluation has the advantage of computing each fact only once. For a Datalog program, there is at most a polynomial number of facts that can be derived. If each fact is computed only once, the evaluation is guaranteed to be polynomial. The top-down evaluation has the advantage that only facts related to the query goal are computed. However, a naieve implementation of the top-down evaluation may compute a fact mutiple times, leading to inefficiencies.

A standard Prolog system operates in a top-down manner: each rule is tried in order and so is each subgoal of a rule. It does not remember what facts have already been computed, so a fact may be computed multiple times if it is needed at different places in the depth-first search process. This may be a problem for performance. A more severe problem in those Prolog engines are that cycles in Datalog rules may lead to nonterminating execution, and the order of the clauses, as well as that of the subgoals within a clause, affects the result of execution. For example, following is a Datalog specification for computing transitive closure.

```
reachable(v1, v3) :- reachable(v1, v2), reachable(v2, v3).
reachable(v1, v2) :- edge(v1, v2).
```

Suppose the facts about `edge` are:

```
edge(node1, node2).
edge(node1, node3).
edge(node2, node3).
```

Executing the following query in a standard Prolog system will cause an infinite loop without outputing a single result.

```
| ?- reachable(node1,V).
```

If we switch the order of the two rules for `reachable`, the query will output three results (correctly) before going into an infinite loop.

```
| ?- reachable(node1,V).
V = node2 ? ;
V = node3 ? ;
V = node3 ? ;
```

Cycles are common when it comes to modeling computer attacks. For example, an attacker can modify a user's files if he can execute arbitrary code as the user. But it is possible that the reason he can execute arbitrary code as the user is because he modified some executables and installed a Trojan-horse program. In particular, the following two interaction rules may cause cycles in derivation.

```
accessFile(P, H, Access, Path) :- execCode(P, H, User),
                                  localFileProtection(H, User, Access, Path).

execCode(Attacker, H, User) :- malicious(Attacker),
                               accessFile(Attacker, H, write, Path),
                               not setuidProgram(H, Path),
                               localFileProtection(H, User, exec, Path).
```

The presence of cycles in interaction rules is completely legitimate in terms of the semantics of security interaction. Requiring interaction rules to be cycle-free is not only too restrictive, but also extremely hard, if not impossible. Unfortunately, these cycles will introduce infinite loops in a standard Prolog system, which views a Datalog program operationally rather than declaratively.

XSB [41] is a system that computes the well-founded semantics of logic programs [20]. XSB supports *tabled execution*, which is a kind of memoization techniques. Put in simple words, the computation of a tabled predicate is conducted

only once and the result is stored in a table for reuse. The effects of tabling are two-fold. First, it essentially implements a dynamic-programming algorithm so that facts about a tabled predicate will not be recomputed during the execution of a logic program. Second, if a tabled predicate is involved in a cycle during evaluation, XSB will detect it and not enter a loop. As a result, cycles in Datalog programs will not introduce nonterminating computation, and the order of clauses does not affect the result of execution. This advantage makes XSB an ideal candidate for the logic engine in MulVAL.

4.1.1 Properties of Datalog evaluation in XSB

Soundness and completeness Soundness and completeness state that 1) any result of the analysis should be a logical consequence of the MulVAL interaction rules and the input facts; 2) the analysis is able to compute all such logical consequences. There are different notions of semantics for Datalog that formally define what logical consequences mean [16, 20, 11]. These semantics coincide for Datalog programs with stratified negation — the only kind of negations used in MulVAL. The XSB system can efficiently compute the well founded semantics [20], which captures the intuitive bottom-up derivation semantics of Datalog programs. Since MulVAL uses XSB as its logic engine, the soundness and completeness of XSB in computing the well founded semantics ensures that the analysis in MulVAL is both sound and complete.

Complexity The complexity of MulVAL analysis is affected by the *data complexity* of the Datalog interaction rules. Data complexity is the evaluation time of a Datalog program with respect to the data input, with the Datalog program fixed. For a pure Datalog program, there is only a constant number of predicates, and the maximum arity of the predicates is also constant. Since an argument of a predicate can only

come from a input domain whose size is in proportion to the size of the data input, there is only a polynomial number of facts that can be possibly derived by the Datalog program. So the data complexity of pure Datalog is at most polynomial. Actually Datalog is data complete for P [14]. The introduction of stratified negation does not affect the polynomial complexity of Datalog [14].

In XSB, if every predicate is tabled, then every fact will be computed only once and the execution time of a Datalog program is guranteed to be polynomial. However, table manipulation also introduces overhead which may counteract the benefit brought by the dynamic programming. Currently we table only enough predicates to avoid infinite loops in programs. The precise complexity of MulVAL reaonsing process, however, depends on the interaction rules and input data. Section 6.2 shows some experimental results that illustrate the speed of the reasoning engine on large synthesized inputs.

4.2 Attack simulation

The goal of attack simulation is to find out what privileges an attacker could get by launching multistage, multihost attacks in a network. Let G be the attack goal, I be the MulVAL interaction rules, and D be the input data in the analysis database. The job of attack simulation is to determine if $I \wedge D \Rightarrow G$ is true. This can be computed by XSB by first loading I and D and then issuing a query: $? - G$. For exmaple, the following query answers the question of “whether principal `attacker` can get root permissions on any of the machines?”

```
?- execCode(attacker, H, root).
```

The XSB system will automatically find all possible ways this goal can be satisfied by applying the interaction rules on the input data. If the query succeeds, the

variables appearing in the query will have a solution — instantiations that make the query true. In this case, the solution for variable `H` will indicate the set of machines whose root can be compromised by `attacker`. XSB will output the solution like the following (the `no` in the last line indicates no more instantiations can be computed).

```
H = workStation;
H = fileServer;
H = webServer;
no
```

Since the policy model in MulVAL considers data confidentiality and integrity, we would like to compute an attacker's access to files that correspond to data symbols in the policy. The following Datalog program does that.

```
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`. To compute all the data access that could be obtained by launching multistage, multihost attacks, one only needs to issue the following query:

```
?- access(P, Access, Data).
```

4.3 Policy check

Attack simulation can compute all privileges an attacker could gain by launching multistage, multihost attacks. If we output all these privileges, the amount of information will be huge for a system administrator to digest. For one thing, not all privileges an attacker can get is harmful. And many times one privilege subsumes

many others. For example, if an attacker gets root on a machine, he can access every individual files on that machine. It is not very useful to output all the file accessses the attacker can have on the compromised system. In MulVAL, a policy is used to filter through the raw output from attack simulation and only output the *essential* undesirable privileges. These essential output captures the high-level goal of security administration. For example, if the ultimate goal of the security administration is to protect the confidentiality and integrity of sensitive information, the system administrator can define a data access policy in MulVAL and check the system against it. This can be done by the following simple Datalog program.

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

4.3.1 More policies

The data access policy is not the only policy MulVAL supports. One can view the MulVAL policy check as a two-step process. In the first phase, only attack simulation is conducted and it computes all privieleges an attacker could gain by lauching multistage, multihost attacks. This step has polynomial running time. In the second phase, the raw access data is compared with the given high-level policy to filter out essential undesirable access. This part can be done independently from the attack simulation and may have higher complexity than polynomial, depending on the expressive power of the policy language.

The MulVAL reasoning system supports general Prolog as the policy language. For example, in a law firm a policy may specify that nobody can access legal documents of two clients that have a conflict of interest. This can be specified by the following Prolog (actually, Datalog) program.

```
allow(Employee, read, Documents) :- belongTo(Documents, Client),
                                    not conflictOfInterest(Employee, Client).

conflictOfInterest(P1, P2) :- sue(P1, P2).

conflictOfInterest(P1, P2) :- represent(P1, P3),
                              conflictOfInterest(P3, P2).

conflictOfInterest(P1, P2) :- conflictOfInterest(P2, P1).
```

Intuitively, an employee can read a client's document if there is no conflict of interest between the employee and the client. Two principals have conflicts of interest if one sues the other, or one represents another principal in court who has a conflict of interest with the other principal. Suppose the following are facts about predicates `belongTo`, `sue`, and `represent`.

```
belongTo(docA, company1).
belongTo(docB, company2).
belongTo(docC, consumer1).

sue(consumer1, company1).
sue(company2, company1).
represent(lawyer1, company1).
represent(lawyer2, company2).
represent(lawyer3, consumer1).
```

`lawyer1` can access documents of `company1` but not those of `company2` or `consumer1`. `lawyer2` and `lawyer3` can access documents of both `company2` and `consumer1`, but not the documents of `company1`.

Should one need even richer policy specification, the attack simulation can still be performed efficiently and the output data access tuples can be sent to a policy

resolver that can handle the richer policy specification efficiently. However, it is not clear whether richer policy specification is useful in vulnerability analysis. It seems in most situations the hypothetical attacker comes from outside of the corporation, in which case what one cares about is just which accesses should be shielded from outsiders. Richer policy specification may be useful for analyzing insider threats, which is beyond the scope of this dissertation.

4.4 Attack-tree generation

The notion of attack tree was first introduced by Bruce Shneier [45]. In MulVAL, an attack tree is a trace that shows steps of a potential attack path that lead to a particular goal. Every leaf node of an attack tree is a Datalog tuple representing configuration information or initial privilege of the attacker. Every internal node is a Datalog tuple representing privilege the attacker could get by launching multistage, multihost attacks. Formally, let v be an internal node in the attack tree, and it has k children $v_1, ...v_k$. Then there must exist an interaction rule p :- $p_1, ...p_k$ and a substitution θ, such that $v = [\theta]p$, and $v_i = [\theta]p_k$ for $i = 1...k$. In other words, very internal node is derived from its children by applying one of the interaction rules in MulVAL.

Attack trees are important for system administrators to understand how an attacker can achieve his goal, and to decide upon remediation actions. Figure 4.1 shows an example attack tree.

This attack tree demonstrates the potential attack path in the example network in Figure 1.2. The name of the rule used in deriving each internal node is labeled in the graph. One can start from the bottom and follow the steps of an attacker. First by exploiting a bug in a server program, he gets local access on the server. Then

```
|-- policyViolation(attacker,read,projectPlan)
    |-- dataBnd(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,'/usr/local/share')
            Rule: NFS semantics
                |-- nfsMounted(fileServer,'/export',read,
                                      workStation, '/usr/local/share')
                    |-- nfsExportInfo(fileServer,/export,read,workStation)
                    |-- nfsMountInfo(workStation,/usr/local/share,
                                             fileServer,/export)
                |-- accessFile(attacker,fileServer,write,'/export')
                Rule: NFS shell
                    |-- malicious(attacker)
                    |-- 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)
                    |-- nfsExportInfo(fileServer,/export,write,webServer)
                    |-- hacl(webServer,fileServer,rpc,100003)
        |-- localFileProtection(workStation,root,read,/home)
    |-- not allow(attacker,read,projectPlan)
```

Figure 4.1: A MulVAL attack tree

he uses a program like NFS shell to access the NFS file server. By the semantics of NFS file sharing, once the attacker can modify files on the server, the client machine that mounts that server portion will also be affected. Thus the attacker has the ability to install a Trojan-horse program on the client machine. The machine will be compromised by the Trojan horse and confidential data stored on it will be leaked to the attacker.

In MulVAL, attack analysis is carried out by a Prolog program, thus an attack tree is a derivation tree, or proof, of a successful Prolog query. There are various ways to generate proofs from Prolog. MulVAL adopts a meta-programming approach in proof generation. The meta interpreter in MulVAL handles tabling so that even programs with side effects can run correctly in the interpreter. Details of the meta-interpreter can be found in Appendix B.

4.5 Attack-graph generation

While attack trees generated by the meta-interpreter serve the purpose of visualizing attack paths, the methodology also has several drawbacks. Meta-interpreting a Prolog program is one order of magnitude slower than executing it directly in Prolog. Moreover, even if there is only a polynomial number of facts that can be derived by a Datalog program, the number of proof trees generated could be exponential in the worst case.

The XSB system includes a *justifier* program [44, 36] that can compute evidence of derived literals while the program is running, thus eliminating the need for meta-interpreting. The evidence is stored in the Prolog database and can be extracted for visualization. According to test results [36], online justification only introduces 8% runtime overhead to the program, much better than meta-interpretation. To avoid

the exponential blow up of proof trees, an acyclic graph can be output visualizing the logical relationships among derived literals. The size of such graphs is polynomial to the size of the program. Currently attack-graph generation has not been implemented in MulVAL. For the attack graph generated to be more useful, a user-friendly tool that can explore the information provided by the justifier would be necessary. This is beyond the scope of this dissertation and is left for future work.

Chapter 5

Hypothetical analysis

The basic analysis described in the last chapter is conducted under the current snapshot of the network ocnfiguration. However, when it comes to security analysis, it is not sufficient to only consider the current situation. An important criterion in the security robustness of a network configuration is how much unknown threats it can withstand. An unknown threat may be a software vulnerability that exists in a piece of software but has not been reported to the public (zero-day vulnerability), or a compromised password of a user account not to the knowledge of either the user or the administrator. When configuring a network, a system administrator must take into account the possibility of these threats. Thus, a security analysis tool should be able to make statements with regard to unknown threats. For example, even though there is no known vulnerability on the web server, the system administrator may still want to know what would happen should a vulnerability be discovered. After all, the main purpose of having firewalls is to guard against unknown threats.

This chapter attempts to answer the question of how to conduct network security analysis with the existence of unknown threats. We call it hypothetical analysis.

5.1 Definition

A hypothetical analysis is one conducted under certain assumptions. Typically those assumptions are about hypothetical unknown threats existing in the network. Following are examples of hypothesis about unknown threats.

- There is an unknown remotely exploitable bug on the web server;
- An attacker has already taken over a particular machine in the network;
- There is a certain number of unknown vulnerabilities in the network, but it is not clear which software will be affected and what is the nature of the vulnerabilities.

Hypothetical analysis imposes stronger requirements on network security. It injects artificial adverse conditions and checks if the network is still secure under those faked conditions. Formally, let S be a logical statement about an attacker's potential privileges obtainable through launching multistage, multihost attacks in a network, and H be a set of statements of hypothetical conditions on the network, a hypothetical analysis answers the truthfulness of the formula $H \Rightarrow S$, where $\Rightarrow$ is the logical implication connective.

Some previous work in vulnerability analysis mentioned the "what if" analysis [27], which mainly considers what will happen if the system administrator makes some changes to the configuration. This is a different problem than the one addressed in this chapter. For this problem the algorithm discussed in the previous chapter will suffice. One can just construct the Datalog representation of the proposed new configuration and run the analysis on it. One may tend to think the same technique can be applied to answer the questions about unknown threats. However, depending on the kind of hypothesis assumed, the hypothetical analysis described in this chapter may not be reducable to Datalog evaluation. As we will see, the methodology introduced in this

chapter manipulate hypothetical assumptions dynamically, and as a result can answer a much richer set of questions than a Datalog program could. Since some of these questions are not reducable to Datalog evaluation, richer programming languages are necessary to conduct the analysis. The language we chose is Prolog, which has a clear logical meaning and the ability to dynamically operate hypothesis.

5.2 Conducting hypothetical analysis in Prolog

The hypothetical anlysis introduces unknown threats assumptions in a *dynamic predicate*. A dynamic clause in Prolog is one that can be added or retracted from the reasoning database at run time. To add a clause, the primitive `assert(C)` is called and clause C is then added into the database. To remove a clause, `retract(C)` is called and it recursively removes all clauses in the database that matches `C`.

Let `Statement` be the property the system administrator wants to verify, and `Hyps` is a list of statements on hypothetical conditions in the network. The following Prolog program computes the hypothetical analysis of `Hyps` ⇒ `Statement`.

```
with_hypothesis(Hyps, Statement) :- cleanState,
                                    assert_list(Hyps),
                                    Statement.
```

`assert_list` introduces each hypothetical condition dynamically into the Prolog database. Prolog does not automtically retract dynamic clauses when the execution backtracks, so these hypothetical conditions will remain in the database and may interfere with subsequent analysis. To avoid such interference, a program `cleanState` (whose implementation is omitted here) retracts all asserted hypothetical conditions and clean all the table entries that depend on them.

As an example, the following query will show whether the policy could be violated under the assumption that the attacker can take over machine `webServer` as user `apache`.

```
| ?- with_hypothesis([execCode(attacker,webServer,apache)],
                         policyViolation(Adversary, Access, Resources)).
```

We introduce a dynamic 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 following two clauses induct fake bugs into the reasoning process.

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

Using the `bugHyp` predicate, one can write queries to determine if the network is still secure even if unknown vulnerabilities with certain properties exist in the network. For example, the following query answers the question of "will the network still be secure if there is a remotely exploitable bug on the web server?".

```
| ?- with_hypothesis([bugHyp(webServer,Prog,remote,Consequence)],
                         policyViolation(Adversary, Access, Resources)).
```

Note that `Prog` is a variable, so the vulnerability can be in arbitrary number of programs as variable `Program` gets instantiated with different programs. If we want to restrict the hypothetical assumption to vulnerabilities in *exactly* one program, we can write a query like the following:

```
| ?- program(Prog),
     with_hypothesis([bugHyp(webServer, Prog, remote, Consequence)],
                         policyViolation(Adversary, Access, Resources)).
```

Primitive predicate `program` returns a concrete program installed in the network. After it is called, the variable `Prog` will have already been instantiated with a concrete term, so the hypothetical analysis will be conducted with the assumption that a vulnerability exists in this particular program. Prolog backtracking will cycle through all programs installed in the network, and the `cleanState` call at the beginning of `with_hypothesis` will guarantee that each time only vulnerabilities in one program is asserted in the current database.

Continuing along this idea, we consider the following hypothetical analysis problem:

"What will happen if there are N $(N = 1, 2, ...)$ unknown software vulnerabilities in the network?"

Like in the previous example, the assumptions do not specify which pieces of software are vulnerable. So we need to consider all possible combinations of N programs. If there are in total M programs installed in the network, there will be $\binom{M}{N}$ combinations to consider. The code below conducts the analysis for the case of $N = 2$.

```
with_two_advisories(Prog1, Range1, Consequence1, Prog2, Range2, Consequence2,
                                                                Analysis) :-
        program(Prog1),
        program(Prog2),
        Prog1 @< Prog2,
        with_hypothesis([bugHyp(H1, Prog1, Range1, Consequence1),
                         bugHyp(H2, Prog2, Range2, Consequence2)],
                      Analysis))).
```

The Prolog term-comparison operation `Prog1 @< Prog2` makes sure each combination is considered only once during backtracking. `Analysis` is the particular analysis program (*e.g.* `policyViolation(Adversary, Access, Resources)`) conducted under the assumptions and is passed as a parameter to function `with_two_advisories`.

The query below will check if the network can withstand two new advisories, *i.e.* the security policy will still be upheld even if there are two unknown software vulnerabilities in the network.

```
| ?- with_two_advisories(Prog1, Range1, Consequence1,
                              Prog2, Range2, Consequence2,
                              policyViolation(Adversary, Access Resources)).
```

If a policy violation is discovered, the information about the hypothetical software bugs that cause it will be output in the first six arguments of the `with_two_advisories` predicate.

Chapter 6

Practical Experience

We manually built analysis databases that reflect both real and synthesized networks and tested the interaction rules on those benchmarks. This chapter describes the preliminary results from those tests.

6.1 Experimental result on small networks

6.1.1 A small real-world example

The Princeton Computer Science Department has a small internal network used by several hundred users. In this benchmark, we modeled and analyzed a subset of the network that contains three public servers managed by the system administrators. The three machines have the same configuration. We ran OVAL scanners on the machines and got the following output after converting them to Datalog tuples.

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

The NVD database has the following information about the vulnerabilities:

```
vulProperty('CVE-2004-0427', localExploit, dos).
vulProperty('CVE-2004-0554', localExploit, dos).
vulProperty('CVE-2004-0495', localExploit, privEscalation).
vulProperty('CVE-2002-1363', remoteExploit, privEscalation).
```

The following configuration information about the host `publicServer` is true:

```
clientProgram(publicServer, '/usr/local/bin/mozilla').
dependsOn(publicServer, '/usr/local/bin/mozilla', libpng).
```

The principal binding information is as follows:

```
hasAccount(employee, publicServer, employeeAccount).
hasAccount(employee, userMachine, employeeAccount).
inCompetent(employee).
hasAccount(sysAdmin, publicServer, root).
located(attacker, internet).
malicious(attacker).
```

And the host-access control list (HACL) is

```
hacl(internet, publicServer, tcp, 22).
hacl(publicServer, AnyDestination, AnyProtocol, AnyPort).
hacl(H, H, AnyProtocol, AnyPort).
```

The last entry in the HACL list indicates any machine can talk to itself through any protocol and port.

Because our department does not have much confidential information, we do not specify the data access policy for this benchmark. Instead, we make the following query to see if an attacker from `internet` can get root on the public server.

```
|-- execCode(attacker,publicServer,root)
Rule: local exploit
     |-- malicious(attacker)
     |-- execCode(attacker,publicServer,employeeAccount)
     Rule: remote exploit for a client program
          |-- malicious(attacker)
          |-- vulExists(publicServer,/usr/local/bin/mozilla,
                                   remoteExploit,privEscalation)
          Rule: Library bug
               |-- vulExists(publicServer,libpng,
                                   remoteExploit,privEscalation)
                    |-- vulExists(publicServer,CVE-2002-1363,libpng)
                    |-- vulProperty(CVE-2002-1363,remoteExploit,privEscalation)
               |-- dependsOn(publicServer,/usr/local/bin/mozilla,libpng)
          |-- clientProgram(publicServer,/usr/local/bin/mozilla)
          |-- inCompetent(employee)
          |-- hasAccount(employee,publicServer,employeeAccount)
     |-- vulExists(publicServer,kernel,localExploit,privEscalation)
          |-- vulExists(publicServer,CVE-2004-0495,kernel)
          |-- vulProperty(CVE-2004-0495,localExploit,privEscalation)
     |-- setuidprogram(publicServer,kernel,root)
```

Figure 6.1: Attack tree for the benchmark in Section 6.1.1

```
| ?- execCode(attacker, publicServer, root).
yes
```

The meta-interpreter outputs an attack tree (shown in Figure 6.1). There is a remotely exploitable bug in the `libpng` library. Since a client program `mozilla` uses that library, the bug will also appear in `mozilla`. According to the rule on remote exploit client-side bugs, an incompetent but innocent user may execute the buggy program on an untrusted input and thus allows an attacker to execute arbitrary code on his behalf. Once the attacker gets local access to the system, he can exploit bug `CVE-2004-0495` in the kernel and escalate its privilege to superuser. In our model, the Linux kernel is treated as both a setuid program owned by root and a network service running as root.

This reasoning coincides with the response from our department's system administrator. When seeing the first three local vulnerabilities, he did not think immediate

action needed to be taken, because users in the department are trusted to not exploit those vulnerabilities. The malicious attacker comes from outside (`internet`) and he cannot directly exploit the bug without first getting access to a user account in the system. Since there is no immediate danger that can be caused by the local vulnerabilities, it is not necessary to patch the kernel right away, which requires rebooting the system and interrupting many people's work. However, once the `libpng` bug is seen, the system administrator determined that this bug must be patched immediately, for exactly the same reason as shown by the attack tree output from MulVAL. Since the bug in `libpng` is remotely exploitable, an attacker from outside may compromise an ordinary user account (we assume the system administrators are cautious and will not open webpages from suspicious sites, at least when logged in as "root.") Combined with the local vulnerabilities, the attacker can potentially get root on the public server, which will be a severe compromise.

Several features of MulVAL are highlighted in this application of the tool to a simple but real network. First, MulVAL can be used even without a formally defined security policy. Because the department is an academic institution and does not have much sensitive information, the data-access policy model is not suitable for the purposes of its administration. What the system administrators care about may be "no attacker should be able to compromise root on any managed servers." It is easy to check this statement by executing a simple Prolog query in MulVAL. This flexibility brought by the separation of attack simulation and policy check makes MulVAL a valuable tool for system administrators to check what might happen in the network.

Even though there is no formally defined data access policy in this example, the principal binding information still sheds light on some basic trust principles the system administrator follows in managing the network, which can be viewed as part of a policy statement. `incompetent(employee)` states that the system administrator does not

expect the users of the network to be security-vigilant. They may do stupid things, but are not intentionally malicious, for there is no clause of `malicious(employee)`. The only malicious attacker comes from outside (`located(attacker, internet)`).

Although there is only one machine under consideration in this example, the attack tree shows a multistage attack. Our system administrators are highly competent and only in rare occasions has an outside intruder successfully entered our network. When the scale of an enterprise network is large, both in size and complexity, we doubt even highly competent administrators can handle the management well without the aid of an automatic tool. MulVAL is designed to alleviate the system administrator's burden of the daily repetitive work of reading vulnerability reports, checking configurations, and figuring out all possible attack paths. With the help of MulVAL, the expensive human labor can be used to define sound and sensible security policies and to improve the overall design of the network infrastructure.

Once an attack tree shows that certain undesirable situation may happen and decisions have to be made on how to prevent them, there are always different alternatives to break the attack trace. In this example, one can choose to patch the local privilege escalation bug in the kernel, or the remote privilege-escalation bug in "libpng". The system administrator chooses the former, because 1) it happens earlier in the attack tree; 2) the patching does not require rebooting the system. It would be an interesting research topic to study how to provide automatic heuristics as to what countermeasures to apply. This is beyond the scope of this dissertation and is left as future work.

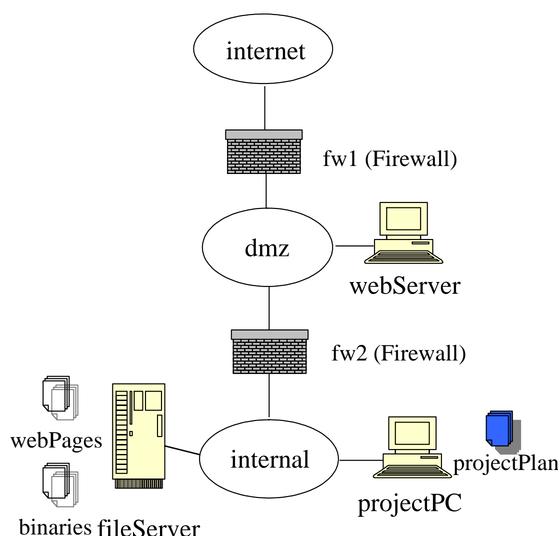

Figure 6.2: A real-world example

6.1.2 An example multihost attack

Figure 6.2 is the example small network mentioned in Chapter 1. In this section we show how we applied the MulVAL analysis engine to detect potential attack paths due to vulnerabilities (both real and hypothetical) on the machines. We first describe the various Datalog-tuple inputs to the analysis engine.

Network topology. There are three zones (`internet`, `dmz` and `internal`) separated by two firewalls (`fw1` and `fw2`). The administrators manage the `webserver`, the `projectPC` 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(projectPC, AnyHost, AnyProtocol, AnyPort).
```

```
hacl(H, H, AnyProtocol, AnyPort).
```

Vulnerability information

```
vulExists(fileServer, 'CVE-2003-0252', mountd).
vulExists(webServer, 'CVE-2002-0392', httpd).
vulProperty('CVE-2003-0252', remoteExploit, privEscalation).
vulProperty('CVE-2002-0392', remoteExploit, privEscalation).
```

There are two vulnerabilities reported by the OVAL scanner, on the machine `fileServer` and `webServer` respectively. The corresponding NVD database entries describing the exploitable range and consequences of these two bugs are also shown above.

Machine configuration Following are relevant Datalog tuples describing machine configurations output by the MulVAL scanners.

```
/* configuration information of fileServer */
networkServiceInfo(fileServer, mountd, rpc, 100005, root).
nfsExportInfo(fileServer, '/export', read, workStation).
nfsExportInfo(fileServer, '/export', write, workStation).
nfsExportInfo(fileServer, '/export', read, webServer).
nfsExportInfo(fileServer, '/export', write, webServer).

/* configuration information of webServer */
networkServiceInfo(webServer, httpd, tcp , 80 , apache).
nfsMounted(webServer, '/share', fileServer, '/export', read).
nfsMounted(webServer, '/share', fileServer, '/export', write).

/* configuration information of workStation */
nfsMounted(workStation, '/share', fileServer, '/export', read).
nfsMounted(workStation, '/share', fileServer, '/export', write).
```

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` 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` is also exported to `webServer` since the web pages are also stored on the file server.

Data binding.

```
dataBind(webPages, fileServer, '/export').
dataBind(projectPlan, workStation, '/home').
```

Two kinds of data are mentioned by the security policy: `webPages`, which is stored on the file server, and `projectPlan`, which is stored on the individual workstations.

Principals. The principal `sysAdmin` manages the machines with user name `root`. Since all other users in the corporation are treated equally for the purpose of this example, we model them as one principal `employee`. `employee` 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 is located in `internet`. The Datalog tuples for principal bindings are:

```
hasAccount(employee, workStation, employeeAccount).
hasAccount(sysAdmin, webServer, root).
hasAccount(sysAdmin, fileServer, root).
```

```
hasAccount(sysAdmin, workStation, root).
[ilocated(attacker, internet).
malicious(attacker).
```

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

```
allow(Anyone, read, webPages).
allow(employee, Acc, projectPlan).
allow(sysAdmin, Acc, AnyData).
```

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;
```

One trace of the first violation is shown in Figure 6.3. 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`, and the
`export` directory on `fileServer` is exported to `webServer`, the attacker can use a program such as "NFS Shell" to send NFS requests to the file server on behalf of any

```
|-- 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,'/share')
            Rule: NFS semantics
                |-- nfsMounted(workStation,'/share',fileServer,'/export',read)
                |-- accessFile(attacker,fileServer,write,'/export')
                Rule: NFS shell
                    |-- malicious(attacker)
                    |-- execCode(attacker,webServer,apache)
                    Rule: remote exploit of a server program
                        |-- malicious(attacker)
                        |-- vulExists(webServer,httpd,
                                      remoteExploit,privEscalation)
                            |-- vulExists(webServer,CVE-2002-0392,httpd)
                            |-- vulProperty(CVE-2002-0392, remoteExploit,
                                                         privEscalation)
                        |-- networkServiceInfo(webServer,httpd,tcp,80,apache)
                        |-- netAccess(attacker,webServer,tcp,80)
                            |-- located(attacker,internet)
                            |-- hacl(internet,webServer,tcp,80)
                    |-- nfsExportInfo(fileServer,/export,write,webServer)
                    |-- hacl(webServer,fileServer,rpc,100003)
        |-- canAccessFile(workStation,root,read,/home)
    |-- not allow(attacker,read,projectplan)
```

Figure 6.3: Attack tree for the benchmark in Section 6.1.2

user. So 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 and a Trojan-horse program can be installed. Eventually an innocent user will execute the Trojan-horse program; this will give the attacker complete control of `workStation` as the user. Thus the files stored on it would also be compromised.

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

6.1.3 Hypothetical analysis

To test the hypothetical analysis algorithm, we removed the Datalog tuples representing the two vulnerabilities. The policy check will not report a violation. But introducing one hypothetical bug would break it. We ran the hypothetical algorithm and it produces the expected result (Figure 6.4). This trace is different from the one in Section 6.1.2 only in that the software vulnerabilities are introduced by hypothetical bugs.

6.2 Performance and Scalability

The running time of MulVAL consists of two parts: time for the scanner to collect configuration information and time for the reasoning engine to analyze the collected data. We measured the performance of the MulVAL 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 reasoning engine runs on a Windows PC with 2.8GHz Pentium 4 processor with 513MB RAM.

```
|-- with_one_advisory(httpd,_h95,_h109,policyViolation(attacker,read,projectplan))
    |-- with_hypothesis([bugHyp(_h574,httpd,_h95,_h109)],
                        policyViolation(attacker,read,projectplan))
        |-- 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: NFS shell
                            |-- malicious(attacker)
                            |-- execCode(attacker,webServer,apache)
                            Rule: remote exploit of a server program
                                |-- malicious(attacker)
                                |-- vulExists(webServer,httpd,
                                              remoteExploit,privEscalation)
                                Rule: Introducing hypothetical bug
                                    |-- bugHyp(webServer,httpd,
                                              remoteExploit,privEscalation)
                                |-- networkServiceInfo(webServer,httpd,tcp,80,apache)
                                |-- netAccess(attacker,webServer,tcp,80)
                                    |-- located(attacker,internet)
                                    |-- hacl(internet,webServer,tcp,80)
                            |-- nfsExportInfo(fileServer,/export,write,webServer)
                            |-- hacl(webServer,fileServer,rpc,100003)
                |-- canAccessFile(workStation,root,read,/home)
            |-- not allow(attacker,read,projectplan)
```

Figure 6.4: Attack tree for the hypothetical analysis in Section 6.1.3

One needs to run the MulVAL scanner on each host in the network and transfer the results to the host running the analysis engine. The scanners can execute asynchronously in parallel on multiple machines. The analysis engine then operates on the data collected from all hosts. Since we did not have access to a real network with thousands of machines, we constructed benchmarks of synthesized networks with different number of hosts. The synthesized networks are similar to the one in Section 6.1.2, but with different numbers of web servers, file servers, and workstations (approximately one-third web servers, one-third file servers, and one-third project PCs). The benchmark is just a collection of Datalog tuples representing the configuration of the synthesized networks. In generating the Datalog tuples, which includes configuration of every machine, with host access rules, vulnerability information, *etc.*, we make sure the number of attack paths grows in proportion to the size of the network. The running times (in seconds) are shown in Table 6.2.

MulVAL scanner		236 s	#attack paths
MulVAL reasoning engine	§6.1.1	0.08	1
	1 host	0.08	1
	200 hosts	0.22	135
	400 hosts	0.75	269
	1000 hosts	3.85	669
	2000 hosts	15.8	1335

Table 6.1: Performance data of MulVAL

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 §6.1.1 is the real-world network described in section 6.1.1. Each benchmark labeled "n hosts" consists of n similar Linux hosts; the corresponding number of attack paths is also listed in the table. The running time shows that the reasoning engine scales well with the size of the network. It can handle the benchmark with thousands of

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

Table 6.2: Input size to MulVAL reasoning engine

hosts in just seconds. The input size to the MulVAL reasoning engine, in terms of number of Datalog tuples, is shown in Table 6.2.

The running times as shown in the table actually grow quadratically. Since the complexity of a particular Datalog program depends both on the rules and inputs, it would be useful to study if the quadratic growth is due to the way the benchmark is constructed, or something inherent in the reasoning model. This is left as future work.

A typical network might have a dozen kinds of hosts: many web servers, many file servers, many compute servers, and 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 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 the 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 6.5 shows the performance with regard to different numbers of hosts, programs, and bugs. The running time

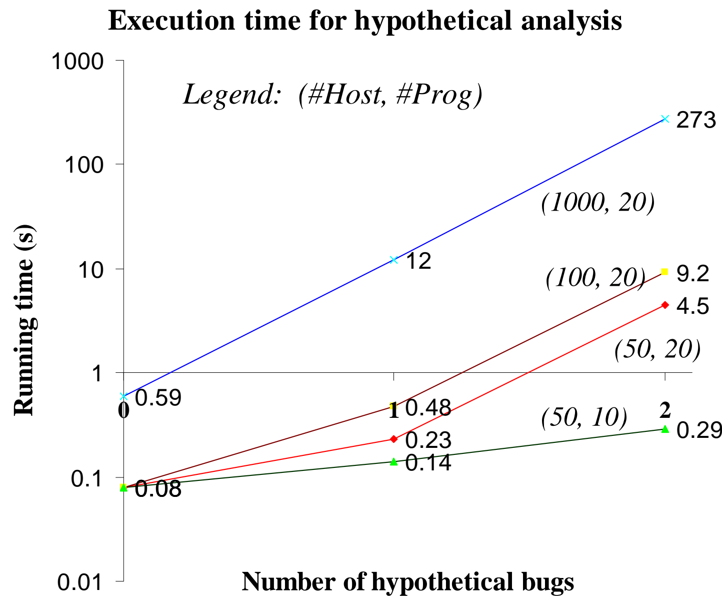

Figure 6.5: Hypothetical analysis.

increases with the number of hypothetical bugs, because the analysis engine must 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 degrades significantly with the increase of k. But it still takes only 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 as important to have fast real-time response. The degraded performance is acceptable. Figure 6.5 shows MulVAL can perform this analysis in a reasonable time frame for a big network. 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.

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 [37]. 47 hosts are selected in such a way as to get geographical diversity (U.S., Canada, Switzerland, Germany, Spain, Israel, India, Hong Kong, Korea, Japan). We were able to log into 39 of these hosts; of these, we successfully installed the scanner on 33 hosts.[2] 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, when its configuration is known to have changed, or periodically, should scan and report configuration information. Then, whenever newly scanned data arrives or new vulnerability data is obtained from OVAL or NVD, the reasoning engine can be run within seconds.

[2]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, real root privileges are advisable.

Chapter 7

Conclusions

This dissertation describes a preliminary attempt at applying logic-programming methodologies in network security analysis. Several other approaches have been proposed in the past. The approach proposed in this thesis has the advantage of both clear declarative specification and efficient execution. Declarative specification brings good modularity, crucial for integrating security knowledge from third parties. Efficiency is important for practical use. The logic-programming approach also provides the expressiveness for programming various analysis algorithms, as demonstrated by the four algorithms already implemented in MulVAL: attack simulation, policy check, hypothetical analysis, and attack-tree generation.

This dissertation makes available a formal attack model for networks of Unix-family machines. The interaction rules discussed in Chapter 2 gives detailed Datalog rules for attack methodologies and security-related operating-system semantics. These rules have been developed based on the knowledge of security and study of real network attack events. However, more research is needed to reveal the accuracy and effectiveness of this reasoning model. A necessary step in making the tool described in this thesis useful in practice is to refine the formal attack model through empirical

study on more real data.

This dissertation describes an end-to-end system, MulVAL, that puts various off-the-shelf tools and data sources — formal software vulnerability advisories, security scanners, and network configuration tools — together and conducts analysis according to the reasoning model. Preliminary experimental study shows that it has the potential to conduct the analysis in seconds on a network of thousands of machines.

This disseration explains various analysis algorithms that are implementated in MulVAL. In particular, the hypothetical analysis is important to show the security robustness of a network by assuming adverse vulnerability conditions. This makes MulVAL useful even when there is no reported software vulnerabilities in the network. The disseration also presents various ways of attack-tree generation in MulVAL.

A limitation of the logic-programming approach is that it cannot specify general nonmonotonic attacks and temporal constraints on attack steps. Model checking is capable of both tasks, but its scalability has not been attested. More research is needed to study what is the best approach to reason about these attacks. It would be useful to conduct research on other logics that can both express such attack scenarios and be evaluated efficiently.

Network security analysis is only the first step towards complete automation of network security management. Once a problem is found, the next question to ask is how to change the configuration such that the problem will be fixed and normal operations will not be impeded. The efficient reasoning engine in MulVAL provides a promising outlook for automating this reconfiguration process.

Appendix A

Interaction Rules for Unix-family Platform

Each interaction rule is introduced by an `explain_rule` clause. The first argument of `explain_rule` is a plain-text explanation of the interaction rule. The second argument is the rule itself. There are also fact clauses, and they are introduced by the `load_clause` statement. The interaction clauses will be loaded dynamically into the Prolog database so that the meta-interpreter introduced in Appendix B can use them.

```
/****** Section execCode *******/
explain_rule(
   'account user can execute arbitrary code',
   (execCode(P, H, Perm) :-
        hasAccount(P, H, Perm))
   ).

explain_rule(
```

```
   'When a principal is compromised, any machine he has
    an account on will also be compromised',
   (execCode(P1, Host, Perm) :-
        principalCompromised(P2, P1),
        hasAccount(P2, Host, Perm),
        canAccessHost(P1, Host))
   ).

explain_rule(
  'local exploit',
  (execCode(P, Host, Perm) :-
        malicious(P),
        execCode(P, Host, Perm2),
        vulExists(Host, Software, localExploit, privEscalation),
        setuidProgram(Host, Software, Perm))
  ).

explain_rule(
  'remote exploit of a server program',
  (execCode(Attacker, H, Perm) :-
        malicious(Attacker),
        vulExists(H, Software, remoteExploit, privEscalation),
        networkService(H, Software, Protocol, Port, Perm),
        netAccess(Attacker, _, H, Protocol, Port))
  ).

explain_rule(
  'remote exploit for a client program',
  (execCode(Attacker, H, Perm) :-
        malicious(Attacker),
        vulExists(H, Software, remoteExploit, privEscalation),
```

```
        clientProgram(H, Software),
        incompetent(P),
        hasAccount(P, H, Perm))
  ).

explain_rule(
  'Trojan horse installation',
  (execCode(Attacker, H, root) :-
        malicious(Attacker),
        accessFile(Attacker, H, write, Path))
  ).



/******** Section netAccess ********/
explain_rule(
  'multi-hop access',
  (netAccess(P, H1, H2, Protocol, Port) :-
        execCode(P, H1, Perm),  /* Any permission level */
        hacl(H1, H2, Protocol, Port))
  ).

explain_rule(
  'direct access',
  (netAccess(P, H, Protocol, Port) :-
        located(P, Zone),
        hacl(Zone, H, Protocol, Port))
  ).
```

```
/****** Section canAccessHost ******/
explain_rule(
  'direct access to hosts',
  (canAccessHost(P, H) :-
        execCode(P, H, Perm))
  ).

explain_rule(
  'access a host through a log in service',
  (canAccessHost(P, H) :-
        logInService(H, Protocol, Port),
        netAccess(P, _, H, Protocol, Port))
  ).



/******** Section accessFile ********/
explain_rule(
  'execCode implies file access',
  (accessFile(P, H, Access, Path) :-
        execCode(P, H, Usr),
        localFileProtection(H, Usr, Access, Path))
  ).



/****** Section principalCompromised ******/
explain_rule(
  'password sniffing',
  (principalCompromised(Victim, Attacker) :-
        hasAccount(Victim, H, Perm),
        execCode(Attacker, H, root),
```

```
        malicious(Attacker))
  ).

explain_rule(
  'incompetent user',
  (principalCompromised(P1, P2) :-
        incompetent(P1),
        malicious(P2))
  ).



/***************** Section ssh *********************/
explain_rule(
  'ssh is a log in service',
  (logInService(H, Protocol, Port) :-
        networkService(H, sshd, Protocol, Port, _))
  ).



/**************** Section  nfs ****************/
/* Principal P can access files on a NFS server if the files
   on the server are mounted at a client and he can access the
   files on the client side */
explain_rule(
  'NFS semantics',
  (accessFile(P, Server, Access, ServerPath) :-
        nfsMounted(Client, ClientPath, Server, ServerPath, Access),
        accessFile(P, Client, Access, ClientPath))
  ).
```

```
/* Principal P can access files on a NFS client if the files
   on the server are mounted at the client and he can access the
   files on the server side */
explain_rule(
  'NFS semantics',
  (accessFile(P, Client, Access, ClientPath) :-
        nfsMounted(Client, ClientPath, Server, ServerPath, read),
        accessFile(P, Server, Access, ServerPath))
  ).


explain_rule(
  'NFS shell',
  (accessFile(P, Server, Access, Path) :-
        malicious(P),
        netAccess(P, Client, Server, rpc, 100003),
        nfsExportInfo(Server, Path, Access, Client).
    ).


/****************** Section misc ******************/
explain_rule(
  'root has arbitrary access',
  (localFileProtection(H, root, Access, Path))
  ).


/* Kernel is both a network service and a setuid program */
load_clause(setuidProgram(Host, kernel, root)).
load_clause(networkService(Host, kernel, _, _, root)).
```

```
explain_rule('Scanner reports security bug',
             (vulExists(H, ID, Sw, Range, Consequence):-
                vulExists(H, ID, Sw),
                vulProperty(ID, Range, Consequence))
             ).



explain_rule('Introducing hypothetical bug',
             (vulExists(H, ID, Sw, Range, Consequence):-
                bugHyp(H, Sw, Range, Consequence))
             ).



explain_rule('Library bug',
             (vulExists(H, ID, Sw, Range, Consequence):-
                vulExists(H, ID, Library, Range, Consequence),
                dependsOn(H, Sw, Library))
             ).
```

Appendix B

Meta-programming in XSB

A Prolog meta-interpreter is a Prolog program that can execute other Prolog programs. While interpreting a Prolog program, a proof tree can be generated to show the derivation steps that lead to a successful query.

B.1 A meta-interpreter for definite Prolog programs

Following is a simple meta-interpreter for definite (negation free) Prolog programs:

```
:- table trace/1.

trace(true) :- !.

trace(A ',' B) :- !, trace(A), trace(B).

trace((A ';' B)) :- trace(A).

trace((A ';' B)) :- trace(B).
```

```
trace((A ';' B)) :- !, fail.

trace(A) :- clause(A, B), trace(B).
```

`trace` is the meta-interpreter program. The order of clauses is important here (unlike a Datalog program). The Prolog cut ('!') operator is used to make sure once a clause is matched to the cut point, no alternatives before the cut point will be tried. The `fail` literal will always fail.

The interesting case is the last rule of the interpreter. `clause(A,B)` returns through backtracking all dynamic clauses in the Prolog run-time environment whose head matches `A` and body matches `B`. The interpreter recursively calls itself on the body of the clause. For facts the body is `true`, and the interpreter will return. The body of a rule may be a single clause or a composite one constructed by the 'and" (',') and "or"(';') operators. These patterns are all handled by the appropriate interperter rules.

There are two kinds of Prolog clauses in Prolog: dynamic clauses and compiled clauses. The `clause` predicate only works for dynamic clauses, which requires us all interaction rules be loaded dynamically. Preliminary performance tests have not discovered much difference between the two.

An interesting observation is that `trace/1` is tabled, which means all predicates that are executed by the interpreter are automatically tabled, eliminating infinite loops in the execution. This simple meta-interpreter just runs a Prolog program, without outputing a derivation tree.

B.2 A meta-interpreter for generating proofs

The following program `trace/2` is augmented to output proof trees in its second argument.

```
:- table trace/2.

trace(true, empty) :- !.

trace((A ',' B), and(PfA, PfB)) :- !, trace(A, PfA), trace(B, PfB).

trace((A ';' B), PfA) :- trace(A, PfA).

trace((A ';' B), PfB) :- trace(B, PfB).

trace((A ';' B), _) :- !, fail.

trace(A, because(A, rule((A:-B)), PfB)) :- clause(A, B),
                   trace(B, PfB),
                   loop_detection(A, PfB).

loop_detection(A, because(B, _, C)) :- !,
                     A \== B,
                     loop_detection(A, C).

loop_detection(A, and(B,C)) :- !,
             loop_detection(A,B),
             loop_detection(A,C).

loop_detection(A, B) :- A \== B.
```

The proof term is constructed by the **`and`**, **`because`**, and **`empty`** functions. The

`because` function takes three parameters: the conclustion, the rule applied, and the reason. If there is a clause that matches rule `A :- B`, and `B` is shown to be true with proof `PfB`, then `A` can be shown to be true with proof `because(A, rule((A:-B)), PfB)`. Cycles in the rules of the program being interpreted will lead to cyclic proofs. Unlike the simple interpreter without proof generation, the tabling mechanism cannot prevent nontermination caused by cyclic proofs. This is because the proof term is the second parameter of predicate `trace/2`, thus cyclic proofs will create infinite number of table entries.

To avoid cyclic proofs, the program `loop_detection(A, B)` checks that literal `A` does not already appear in proof `PfB` before returning the proof term `because(A, rule((A:-B)), PfB)`. This guarantees that no cyclic proof will be output as a result, and thus no nontermination in the meta interpreter will be caused by them [1]. While this loop checking is necessary, it does significantly increase the complexity of the meta-interpreter. One can avoid the quadratic blow-up by using dynamic clauses to mark visited proof nodes. However, executing Prolog programs in the meta-interpreter is already one order of magnitude slower than running them diretly in XSB. This may affect the usage of the proof generator in practice.

B.3 Dealing with negation and side effects

Both negations and side effects are used in the MulVAL analysis algorithm. For a negated literal, the proof is the "nonexistence" of derivations. It is not clear what is a good way to encode this meta-logic argument as a proof witness. MulVAL chose to output the proof tree of a negated literal as the literal itself, as illustrated by the following interpreter rule:

[1] Nontermination will still occur if the original program does not terminate when executed directly in XSB with tabling enabled.

```
trace((not A), (not A)) :- !, not A.
```

Since we do not explain why `not A` is true, the subgoal is not executed in the meta interpreter but rather directly called by the Prolog environment.

Side effects pose a bigger problem for the meta-interpreter. There are two aspects of interaction between side-effects and tabling that may affect the correctness of the meta-interpretation. For one thing, tabled results depending on dynamic clauses must be voided once some or all the dynamic clauses are retracted. In the hypothetical analysis, any IDB predicate may depend on the hypothetical bug. So their interpreted results must also be removed from the table once the hypothetical bug is retracted from the database. On the other hand, clauses having side effects, such as asserting a dynamic clause, should not be tabled at all. There are also some auxiliary predicates, such as the `program` predicate in the hypothetical analysis, which are not necessary to show in the proof tree. Thus, the proof-generating meta-interpreter in MulVAL distinguishes those predicates and does not interprete them, avoiding the above mentioned problems:

```
:- table ttrace/2.

trace(true, empty) :- !.

trace(A, empty) :- dontShowInTrace(A), !, A.

trace((not A), (not A)) :- !, not A.

trace((A ',' B), and(PfA, PfB)) :- !,
                   trace(A, PfA),
                   trace(B, PfB).
```

```
trace((A ';' B), PfA) :- trace(A, PfA).

trace((A ';' B), PfB) :- trace(B, PfB).

trace((A ';' B), _) :- !, fail.

trace(A, Tr) :- ttrace(A, Tr).

ttrace(A, because(A, rule((A:-B)), PfB)) :- clause(A, B),
                    trace(B, PfB),
                    loop_detection(A, PfB).
```

Predicate `dontShowInTrace` specifies computations whose proof trees are not interesting and thus not interpreted by `trace`. These include clauses with side effects and auxiliary clauses that do not shed light on how attacks happen. The meta-interpreter `trace` is not tabled, guaranteeing the side effects in programs will be executed whenever the clause is called. Another interpreter `ttrace`, which is mutually recursive to `trace`, is tabled. Thus we can have a fine-grained control of what program to table and what not. However, at least the `because` case of proof generation needs to be tabled, otherwise cycles in program rules will lead to nonterminating computation (even the `loop_detection` function does not help without tabling).

Bibliography

[1] Rajeev Alur, Thomas A. Henzinger, F.Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *Proceedings of the Tenth International Conference on Computer-aided Verification (CAV 1998),*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[2] 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.

[3] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33:52–59, 2000.

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

[5] 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.

[6] Jay Beale, Haroon Meer, Roelof Temmingh, Charl Van Der Walt, and Renaud Deraison. *Nessus Network Auditing*, chapter 11. Syngress Publishing, 1998.

[7] 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.

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

[9] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 17th IEEE Symp. on Security and Privacy*, pages 164–173, 1996.

[10] 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.

[11] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1:146 – 166, March 1989.

[12] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag New York, Inc., 1987.

[13] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder: A language for specifying security and management policies for distributed systems. Technical report, Imperial College, October 2000.

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

[15] 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.

[16] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733 – 742, 1976.

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

[18] 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.

[19] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110 – 121, Long Beach, California, USA, 2005.

[20] 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.

[21] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174 – 186, Paris, France, 1997.

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

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

[24] Sotiris Ioannidis. *Security policy consistency and distributed evaluation in heterogeneous environments*. PhD thesis, University of Pennsylvania, 2005.

[25] Sotiris Ioannidis, Steven M. Bellovin, John Ioannidis, Angelos Keromytis, and Jonathan M. Smith. Design and implementation of virtual private services. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (Workshop on Enterprise Security, Special Session on Trust Management in Collaborative Global Computing)*, June 2003. Earlier version available as U Penn. CIS Technical Report MS-CIS-01-13.

[26] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[27] 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.

[28] Trevor Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, May 2001.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Stefan Miltchev, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Secure and flexible global file sharing. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track.*, June 2003.

[34] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In *14th USENIX Security Symposium*, Baltimore, MD, USA, August 2005.

[35] Robert Palmer and Ganesh Gopalakrishnan. Partial order reduction assisted parallel model-checking. In *Preliminary proceedings of Parallel and Distributed Model Checking*, Brno, Czech Republic, August 2002.

[36] 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.

[37] 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.

[38] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *NSPW '98: Proceedings of the 1998 workshop on New security paradigms*, pages 71–79. ACM Press, 1998.

[39] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

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

[41] 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.

[42] 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.

[43] 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.

[44] Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Principles and Practice of Declarative Programming*, pages 178–189, 2000.

[45] Bruce Schneier. *Secrets & Lies: Digital Security in a Networked World*, chapter 21. John Wiley & Sons, 2000.

[46] 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.

[47] Laura P. Swiler, Cynthia Phillips, David Ellis, and Stefan Chakerian. Computer-attack graph generation tool. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, June 2001.

[48] 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.

[49] T. Tidwell, R. Larson, K. Fitch, and J. Hale. Modeling internet attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY, June 2001.

[50] Wietse Venema. Tcp wrapper: Network monitoring, access control, and booby traps, July 1992.

[51] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 2004)*, Washington, DC, USA, June 2004.

[52] 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.

[53] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Operating System Design and Implementation (OSDI)*, 2004.

[54] 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.