Tools for Generating and Analyzing Attack Graphs

Oleg Sheyner[1] and Jeannette Wing[2]

[1] Carnegie Mellon University, Computer Science Department
5000 Forbes Avenue, Pittsburgh, PA 15213
oleg@cs.cmu.edu
[2] Carnegie Mellon University, Computer Science Department
5000 Forbes Avenue, Pittsburgh, PA 15213
wing@cs.cmu.edu

Abstract. Attack graphs depict ways in which an adversary exploits system vulnerabilities to achieve a desired state. System administrators use attack graphs to determine how vulnerable their systems are and to determine what security measures to deploy to defend their systems. In this paper, we present details of an example to illustrate how we specify and analyze network attack models. We take these models as input to our attack graph tools to generate attack graphs automatically and to analyze system vulnerabilities. While we have published our generation and analysis algorithms in earlier work, the presentation of our example and toolkit is novel to this paper.

1 Introduction

As networks of hosts continue to grow, it becomes increasingly more important to automate the process of evaluating their vulnerability to attack. When evaluating the security of a network, it is rarely enough to consider the presence or absence of isolated vulnerabilities. Large networks typically contain multiple platforms and software packages and employ several modes of connectivity. Inevitably, such networks have security holes that escape notice of even the most diligent system administrator.

1.1 Vulnerability Analysis and Attack Graphs

To evaluate the security of a network of hosts, a security analyst must take into account the effects of interactions of local vulnerabilities and find global security holes introduced by interconnection. A typical process for vulnerability analysis of a network is shown in Figure 1. First, scanning tools determine vulnerabilities of individual hosts. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, the analyst produces an *attack graph*. Each path in an attack graph is a series of exploits, which we call *actions*, that leads to an undesirable state. An example of an undesirable state is a state where the intruder has obtained administrative access to a critical host.

A typical result of such efforts is a floor-to-ceiling, wall-to-wall "white board" attack graph, such as the one produced by a Red Team at Sandia National Labs for DARPA's CC20008 Information battle space preparation experiment and shown in Figure 2. Each

F.S. de Boer et al. (Eds.): FMCO 2003, LNCS 3188, pp. 344–371, 2004.

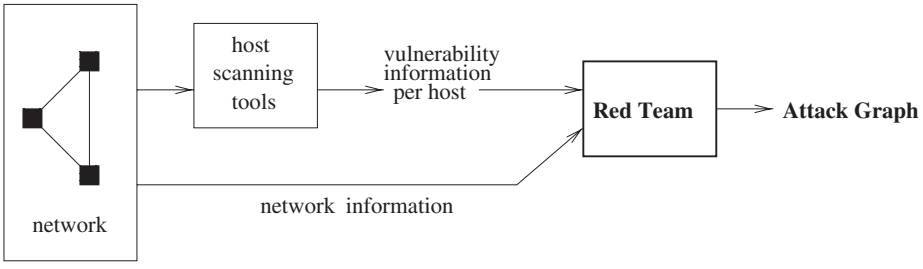

Fig. 1. Vulnerability Analysis of a Network

box in the graph designates a single intruder action. A path from one of the boxes at the top of the graph to one of the boxes at the bottom is a sequence of actions corresponding to an attack scenario. At the end of any such scenario, the intruder has broken the network security in some way. The graph is included here for illustrative purposes only, so we omit the description of specific details.

Attack graphs can serve as a useful tool in several areas of network security, including intrusion detection, defense, and forensic analysis. System administrators use attack graphs for the following reasons:

- To gather information: Attack graphs can answer questions like "What attacks is my system vulnerable to?" and "From an initial configuration, how many different ways can an attacker reach a final state to achieve his goal?"
- To make decisions: Attack graphs can answer questions like "Which set of actions should I prevent to ensure the attacker cannot achieve his goal?" or "Which set of security measures should I deploy to ensure the attacker cannot achieve his goal?"

1.2 Prior Work and Contributions of this Paper

In practice, attack graphs, such as the one shown in Figure 2, are drawn by hand. In earlier work, we show how we can use model checking techniques to generate attack graphs automatically [11, 17]. Our techniques guarantee that attack graphs are sound (each scenario depicted is a true attack), exhaustive (no attack is missed), and succinct (only states and state transitions that participate in an attack are depicted) [16].

In earlier work, we also have presented algorithms for analyzing attack graphs that answer questions such as those posed above [17, 9, 10]. For example, to help system administrators determine how best to defend their system, we cast the decision-making questions in terms of finding a minimum set of actions to remove (or minimum set of measures to deploy) to ensure the attacker cannot achieve his goal. We reduce this NP-complete problem to the Minimum Hitting Set problem [16], which can be reduced to the Minimum Set Cover problem [2], and we then use standard textbook algorithms to yield approximate solutions [3].

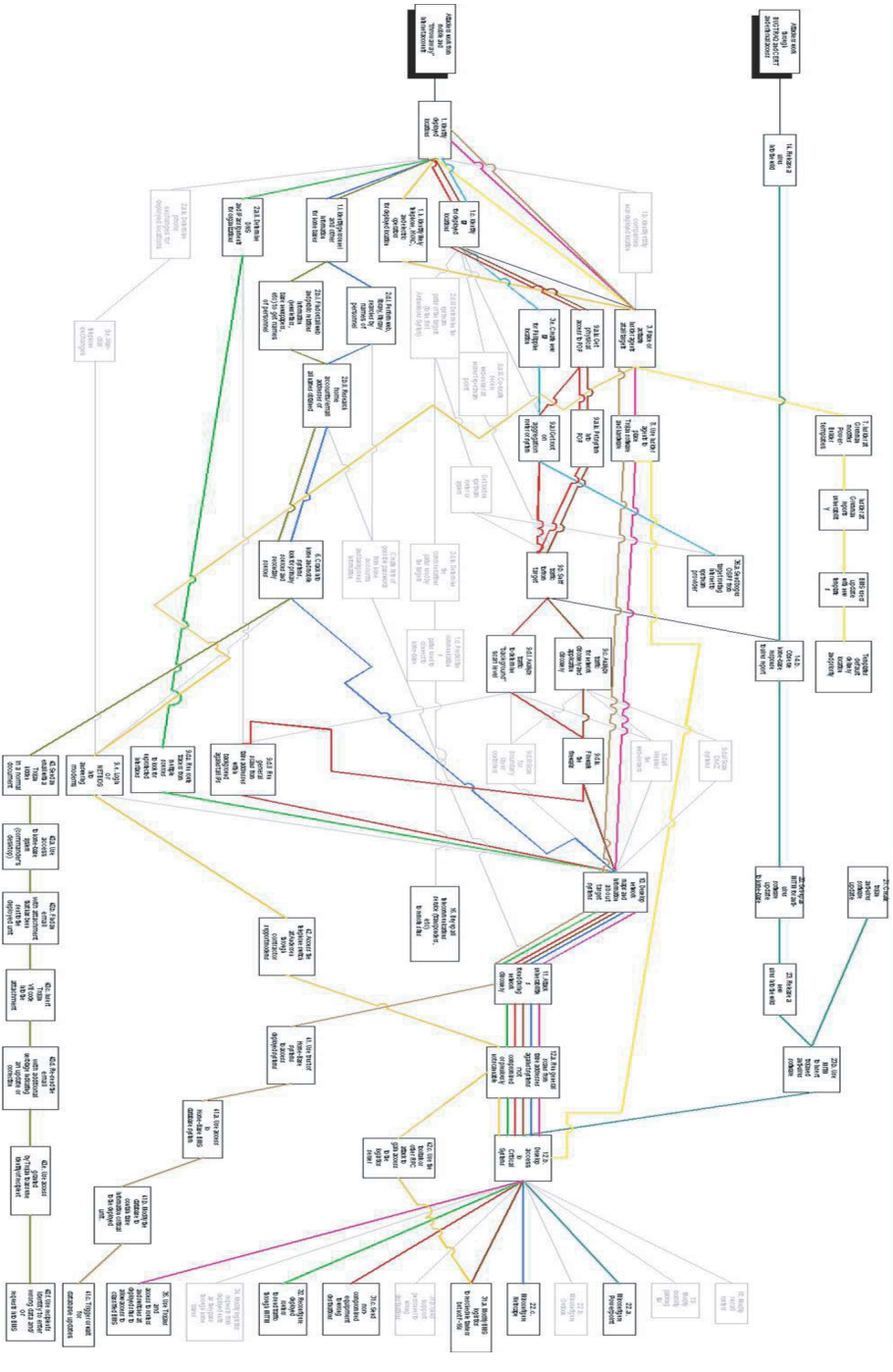

Fig. 2. Sandia Red Team Attack Graph

In this paper, we present the complete details of an example. We use this example to illustrate:

- How we specify network attack models;
- The results of our automatic attack graph generation algorithms;
- The results of our minimization analyses;
- How to use our attack graph toolkit, including how we integrated tools from external sources with ours.

The presentation of this example and our toolkit is novel to this paper.

In Sect. 2 we give general definitions for *attack models* and *attack graphs*. Section 3 narrows the definitions specifically to the domain of network security. Section 4 illustrates the definitions with a small example network. Section 5 focuses on the practical aspects of building a usable attack graph tool. We discuss several approaches to collecting the data necessary to build the network model. Finally, we review related work in Sect. 6.

2 Attack Models and Graphs

Although our primary interest is in multi-stage cyber-attacks against computer networks, we define attack graphs abstractly with respect to models where agents attack and defend a complex system.

Definition 1. *An* attack model *is a finite automaton $M = (S, \tau, s_0)$, where S is a set of states, $\tau \subseteq S \times S$ is a transition relation, and $s_0 \in S$ is an initial state. The state space S represents a set of three agents $\mathcal{I} = \{E, D, T\}$. Agent E is the attacker, agent D is the defender, and agent N is the system under attack. Each agent $i \in \mathcal{I}$ has its own set of possible states S_i, so that $S = \times_{i \in \mathcal{I}} S_i$.*

Definition 2. *A* finite execution *of an attack model $M = (S, \tau, s_0)$ is a finite sequence of states $\alpha = s_0 s_1 \ldots s_n$,such that for all $0 \leq i \leq n$, $(s_i, s_{i+1}) \in \tau$. An* infinite execution *of an attack model $M = (S, \tau, s_0)$ is an infinite sequence of states $\beta = s_0 s_1 \ldots s_n \ldots$, such that for all $i \geq 0$, $(s_i, s_{i+1}) \in \tau$.*

With each agent $i \in \mathcal{I}$ we associate a set of actions A_i, so that the total set of actions in the model is $A = \bigcup_{i \in \mathcal{I}} A_i$. The single root state s_0 represents the initial state of each agent before any action has taken place. In general, the attacker's actions move the system "toward" some undesirable (from the system's point of view) state, and the defender's actions attempt to counteract that effect. For instance, in a computer network the attacker's actions would be the steps taken by the intruder to compomise the network, and the defender's actions would be the steps taken by the system administrator to disrupt the attack.

The specifics of how each agent is represented in an attack model depend on the type of the system that is under attack. In Sect. 3 we specify the agents more precisely for network attack models. Sheyner presents a more formal definition of attack models in his Ph.D. thesis [16].

An attack model is a general formalism suitable for modeling a variety of situations. The system under attack can be virtually anything: a computer network under attack by hackers, a city under siege during war, an electrical grid targeted by terrorists, etc. The attacker is an abstract representation of a group of agents who seek to move the system to a state that is inimical to the system's interests. The defender is an agent whose explicit purpose, in opposition to the attacker, is to prevent this occurrence. The system itself is oblivious to the fact that it is under attack. It goes through its normal routine according to its purpose and goals regardless of the actions of the active agents.

Abstractly, an attack graph is a collection of scenarios showing how a malicious agent can compromise the integrity of a target system. With a suitable model of the system, we can use model checking techniques to generate attack graphs automatically [11, 17, 16]. In this context, correctness properties specify the negation of the attacker's goal: an execution is correct with respect to the property if the attacker does not achieve his goal for that execution. We call such properties *security properties*. An example of a security property in a computer network would be a statement like "the intruder cannot get root access on the web server."

Definition 3. *Given an attack model M, a* security property *P is a subset of the set $L(M)$ of executions of M.*

Definition 4. *An execution $\alpha \in L(M)$ is* correct *with respect to a security property P iff $\alpha \in P$. An execution α is* failing *with respect to P (*violates *P) iff $\alpha \notin P$.*

We say that an attack model M *satisfies* a security property P if it does not have any failing executions (that is, if $L(M) \subset P$). If, however, M does have some failing executions, we say that the set of such executions makes up an attack graph.

Definition 5. *Given an attack model M and a security property P, an* attack graph *of M with respect to P is the set $L(M) \backslash P$ of failing executions of M with respect to P.*

For the remainder of this paper, we restrict the discussion to attack graphs comprised of finite executions only. For a more comprehensive treatment of finite and infinite failing executions we refer the reader to Sheyner [16].

3 Network Attack Graphs

Network attack graphs represent a collection of possible penetration scenarios in a computer network. Each penetration scenario is a sequence of actions taken by the intruder, typically culminating in a particular goal—administrative access on a particular host, access to a database, service disruption, etc. For appropriately constructed network models, attack graphs give a bird's-eye view of every scenario that can lead to a serious security breach.

3.1 Network Attack Model

A *network attack model* is an attack model where the system N is a computer network, the attacker E is a malicious agent trying to circumvent the network's security, and the

defender D represents both the system administrator(s) and security software installed on the network. A state transition in a network attack model corresponds to a single action by the intruder, a defensive action by the system administrator, or a routine network action.

Real networks consist of a large variety of hardware and software pieces, most of which are not involved in cyber attacks. We have chosen six network components relevant to constructing network attack models. The components were chosen to include enough information to represent a wide variety of networks and attack scenarios, yet keep the model reasonably simple and small. The following is a list of the components:

1. *H*, a set of hosts connected to the network
2. *C*, a connectivity relation expressing the network topology and inter-host reachability
3. *T*, a relation expressing trust between hosts
4. *I*, a model of the intruder
5. *A*, a set of individual actions (exploits) that the intruder can use to construct attack scenarios
6. *Ids*, a model of the intrusion detection system

We construct an attack model M based on these components. Table 1 defines each agent i's state S_i and action set A_i in terms of the network components. This construction gives the security administrator an entirely passive "detection" role, embodied in the *alarm* action of the intrusion detection system. For simplicity, regular network activity is omitted entirely.

Table 1. Network attack model

Agent $i \in \mathcal{I}$	S_i	A_i
E	I	A
D	*Ids*	$\{alarm\}$
N	$H \times C \times T$	$\oslash$

It remains to make explicit the transition relation of the attack model M. Each transition $(s_1, s_2) \in \tau$ is either an action by the intruder, or an *alarm* action by the system administrator. An *alarm* action happens whenever the intrusion detection system is able to flag an intruder action. An action $a \in A$ requires that the preconditions of a hold in state s_1 and the effects of a hold in s_2. Action preconditions and effects are explained in Sect. 3.2.

3.2 Network Components

We now give details about each network component.

Hosts. Hosts are the main hubs of activity on a network. They run services, process network requests, and maintain data. With rare exceptions, every action in an attack

scenario will target a host in some way. Typically, an action takes advantage of vulnerable or misconfigured software to gain information or access privileges for the attacker. The main goal in modeling hosts is to capture as much information as possible about components that may contribute to creating an exploitable vulnerability.

A host $h \in H$ is a tuple *(id, svcs, sw, vuls)*, where

- *id* is a unique host identifier (typically, name and network address)
- *svcs* is a list of service name/port number pairs describing each service that is active on the host and the port on which the service is listening
- *sw* is a list of other software operating on the host, including the operating system type and version
- *vuls* is a list of host-specific vulnerable components. This list may include installed software with exploitable security flaws (example: a *setuid* program with a buffer overflow problem), or mis-configured environment settings (example: existing user shell for system-only users, such as *ftp*)

Network Connectivity. Following Ritchey and Ammann [15], connectivity is expressed as a ternary relation $C \subseteq H \times H \times P$, where P is a set of integer port numbers. $C(h_1, h_2, p)$ means that host h_2 is reachable from host h_1 on port p. Note that the connectivity relation incorporates firewalls and other elements that restrict the ability of one host to connect to another. Slightly abusing notation, we say $R(h_1, h_2)$ when there is a network route from h_1 to h_2.

Trust. We model trust as a binary relation $T \subseteq H \times H$, where $T(h_1, h_2)$ indicates that a user may log in from host h_2 to host h_1 without authentication (i.e., host h_1 "trusts" host h_2).

Services. The set of services S is a list of unique service names, one for each service that is present on any host on the network. We distinguish services from other software because network services so often serve as a conduit for exploits. Furthermore, services are tied to the connectivity relation via port numbers, and this information must be included in the model of each host. Every service name in each host's list of services comes from the set S.

Intrusion Detection System. We associate a boolean variable with each action, abstractly representing whether or not the IDS can detect that particular action. Actions are classified as being either *detectable* or *stealthy* with respect to the IDS. If an action is detectable, it will trigger an alarm when executed on a host or network segment monitored by the IDS; if an action is *stealthy*, the IDS does not see it.

We specify the IDS as a function *ids:* $H \times H \times A \rightarrow \{d, s, b\}$, where *ids(*$h_1, h_2, a$*)* = d if action a is *d*etectable when executed with source host h_1 and target host h_2; *ids(*h_1, h_2, a*)* = s if action a is *s*tealthy when executed with source host h_1 and target host h_2; and *ids(*h_1, h_2, a*)* = b if action a has *b*oth detectable and stealthy strains, and success in detecting the action depends on which strain is used. When h_1 and h_2 refer to the same host, *ids(*h_1, h_2, a*)* specifies the intrusion detection system component (if any) located on that host. When h_1 and h_2 refer to different hosts, *ids(*h_1, h_2, a*)* specifies the intrusion detection system component (if any) monitoring the network path between h_1 and h_2.

Actions. Each action is a triple (r, h_s, h_t), where $h_s \in H$ is the host from which the action is launched, $h_t \in H$ is the host targeted by the action, and r is the rule that describes how the intruder can change the network or add to his knowledge about it. A specification of an action rule has four components: *intruder preconditions*, *network preconditions*, *intruder effects*, and *network effects*. The *intruder preconditions* component places conditions on the intruder's store of knowledge and the privilege level required to launch the action. The *network preconditions* specifies conditions on target host state, network connectivity, trust, services, and vulnerabilities that must hold before launching the action. Finally, the *intruder* and *network effects* components list the action's effects on the intruder and on the network, respectively.

Intruder. The intruder has a *store of knowledge* about the target network and its users. The intruder's store of knowledge includes host addresses, known vulnerabilities, user passwords, information gathered with port scans, etc. Also associated with the intruder is the function *plvl: Hosts* $\rightarrow \{$*none, user, root*$\}$, which gives the level of privilege that the intruder has on each host. For simplicity, we model only three privilege levels. There is a strict total order on the privilege levels: *none* $\leq$ *user* $\leq$ *root*.

Omitted Complications. Although we do not model actions taken by user services for the sake of simplicity, doing so in the future would let us ask questions about effects of intrusions on service quality. A more complex model could include services provided by the network to its regular users and other routine network traffic. These details would reflect more realistically the interaction between intruder actions and regular network activity at the expense of additional complexity.

Another activity worth modeling explicitly is administrative steps taken either to hinder an attack in progress or to repair the damage after an attack has occurred. The former corresponds to transitioning to states of the model that offer less opportunity for further penetration; the latter means "undoing" some of the damage caused by successful attacks.

4 Example Network

Figure 3 shows an example network. There are two target hosts, `Windows` and `Linux`, on an internal company network, and a Web server on an isolated "demilitarized zone" (DMZ) network. One firewall separates the internal network from the DMZ and another

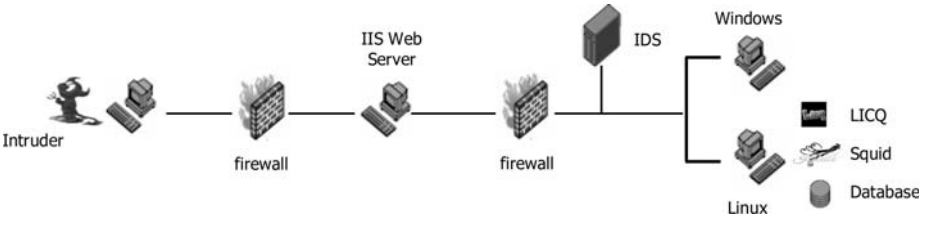

Fig. 3. Example Network

firewall separates the DMZ from the rest of the Internet. An intrusion detection system (IDS) watches the network traffic between the internal network and the outside world.

The `Linux` host on the internal network is running several services—Linux "I Seek You" (*LICQ*) chat software, *Squid* web proxy, and a *Database*. The *LICQ* client lets Linux users exchange text messages over the Internet. The *Squid* web proxy is a caching server. It stores requested Internet objects on a system closer to the requesting site than to the source. Web browsers can then use the local *Squid* cache as a proxy, reducing access time as well as bandwidth consumption. The host inside the DMZ is running Microsoft's Internet Information Services (IIS) on a Windows platform.

The intruder launches his attack starting from a single computer, which lies on the outside network. To be concrete, let us assume that his eventual goal is to disrupt the functioning of the database. To achieve this goal, the intruder needs root access on the database host `Linux`. The five actions at his disposal are summarized in Table 2.

Each of the five actions corresponds to a real-world vulnerability and has an entry in the Common Vulnerabilities and Exposures (CVE) database. CVE [22] is a standard list of names for vulnerabilities and other information security exposures. A CVE identifier is an eight-digit string prefixed with the letters "CVE" (for accepted vulnerabilities) or "CAN" (for candidate vulnerabilities).

Table 2. Intruder actions

Action	**Effect**	**Example CVE ID**
IIS buffer overflow	remotely get root	CAN-2002-0364
Squid port scan	port scan	CVE-2001-1030
LICQ gain user	gain user privileges remotely	CVE-2001-0439
scripting exploit	gain user privileges remotely	CAN-2002-0193
local buffer overflow	locally get root	CVE-2002-0004

The IIS buffer overflow action exploits a buffer overflow vulnerability in the Microsoft IIS Web Server to gain administrative privileges remotely.

The *Squid* action lets the attacker scan network ports on machines that would otherwise be inaccessible to him, taking advantage of a misconfigured access control list in the *Squid* web proxy.

The *LICQ* action exploits a problem in the URL parsing function of the *LICQ* software for Unix-flavor systems. An attacker can send a specially-crafted URL to the *LICQ* client to execute arbitrary commands on the client's computer, with the same access privileges as the user of the *LICQ* client.

The scripting action lets the intruder gain user privileges on Windows machines. Microsoft Internet Explorer 5.01 and 6.0 allow remote attackers to execute arbitrary code via malformed Content-Disposition and Content-Type header fields that cause the application for the spoofed file type to pass the file back to the operating system for handling rather than raise an error message. This vulnerability may also be exploited through HTML formatted email. The action requires some social engineering to entice a user to visit a specially-formatted Web page. However, the action can work against

firewalled networks, since it requires only that internal users be able to browse the Web through the firewall.

Finally, the local buffer overflow action can exploit a multitude of existing vulnerabilities to let a user without administrative privileges gain them illegitimately. For the CVE number referenced in the table, the action exploits a buffer overflow flaw in the *at* program. The *at* program is a Linux utility for queueing shell commands for later execution.

Some of the actions that we model have multiple instantiations in the CVE database. For example, the local buffer overflow action exploits a common coding error that occurs in many Linux programs. Each program vulnerable to local buffer overflow has a separate CVE entry, and all such entries correspond to the same action rule. The table lists only one example CVE identifier for each rule.

4.1 Example Network Components

Services, Vulnerabilities, and Connectivity. We specify the state of the network to include services running on each host, existing vulnerabilities, and connectivity between hosts. There are five boolean variables for each host, specifying whether any of the three services are running and whether either of two other vulnerabilities are present on that host:

Table 3. Variables specifying a host

variable	meaning
w3svc_h	IIS web service running on host h
squid_h	*Squid* proxy running on host h
licq_h	*LICQ* running on host h
scripting_h	HTML scripting is enabled on host h
vul-at_h	*at* executable vulnerable to overflow on host h

The model of the target network includes connectivity information among the four hosts. The initial value of the connectivity relation R is shown the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). IIS and *Squid* listen on port 80 and the *LICQ* client listens on port 5190, and the connectivity relation specifies which of these services can be reached remotely from other hosts. Each entry consists of three boolean values. The first value is 'y' if h_1 and h_2 are connected by a physical link, the second value is 'y' if h_1 can connect to h_2 on port 80, and the third value is 'y' if h_1 can connect to h_2 on port 5190.

We use the connectivity relation to reflect the settings of the firewall as well as the existence of physical links. In the example, the intruder machine initially can reach only the Web server on port 80 due to a strict security policy on the external firewall. The internal firewall is initially used to restrict internal user activity by disallowing most outgoing connections. An important exception is that internal users are permitted to contact the Web server on port 80.

In this example the connectivity relation stays unchanged throughout an attack. In general, the connectivity relation can change as a result of intruder actions. For example,

Table 4. Connectivity relation

Host	Intruder	IIS Web Server	Windows	Linux
Intruder	y,y,y	y,y,n	n,n,n	n,n,n
IIS Web Server	y,n,n	y,y,y	y,y,y	y,y,y
Windows	n,n,n	y,y,n	y,y,y	y,y,y
Linux	n,n,n	y,y,n	y,y,y	y,y,y

an action may enable the intruder to compromise a firewall host and relax the firewall rules.

Intrusion Detection System. A single network-based intrusion detection system protects the internal network. The paths between hosts `Intruder` and `Web` and between `Windows` and `Linux` are not monitored; the IDS can see the traffic between any other pair of hosts. There are no host-based intrusion detection components. The IDS always detects the *LICQ* action, but cannot see any of the other actions. The IDS is represented with a two-dimensional array of bits, shown in the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). The value is 'y' if the path between h_1 and h_2 is monitored by the IDS, and 'n' otherwise.

Intruder. The intruder's store of knowledge consists of a single boolean variable 'scan'. The variable indicates whether the intruder has successfully performed a port scan on the target network. For simplicity, we do not keep track of specific information gathered by the scan. It would not be difficult to do so, at the cost of increasing the size of the state space.

Initially, the intruder has root access on his own machine `Intruder`, but no access to the other hosts. The 'scan' variable is set to *false*.

Actions. There are five action rules corresponding to the five actions in the intruder's arsenal. Throughout the description, S is used to designate the source host and T the target host. $R(S, T, p)$ says that host T is reachable from host S on port p. The abbreviation $plvl(X)$ refers to the intruder's current privilege level on host X.

Recall that a specification of an action rule has four components: *intruder preconditions*, *network preconditions*, *intruder effects*, and *network effects*. The *intruder preconditions* component places conditions on the intruder's store of knowledge and the privilege level required to launch the action. The *network preconditions* component specifies

Table 5. IDS locations

Host	Intruder	IIS Web Server	Windows	Linux
Intruder	n	n	y	y
IIS Web Server	n	n	y	y
Windows	y	y	n	n
Linux	y	y	n	n

conditions on target host state, network connectivity, trust, services, and vulnerabilities that must hold before launching the action. Finally, the *intruder* and *network effects* components list the effects of the action on the intruder's state and on the network, respectively.

Sometimes the intruder has no logical reason to execute a specific action, even if all technical preconditions for the action have been met. For instance, if the intruder's current privileges include root access on the Web Server, the intruder would not need to execute the IIS buffer overflow action against the Web Server host. We have chosen to augment each action's preconditions with a clause that disables the action in instances when the primary purpose of the action has been achieved by other means. This change is not strictly conservative, as it prevents the intruder from using an action for its secondary side effects. However, we feel that this is a reasonable price to pay for removing unnecessary transitions from the attack graphs.

IIS Buffer Overflow. This remote-to-root action immediately gives a remote user a root shell on the target machine.

action IIS-buffer-overflow **is**
intruder preconditions
$plvl(S) \geq$ user — *User-level privileges on host S*
$plvl(T) <$ root — *No root-level privileges on host T*
network preconditions
w3svc_T — *Host T is running vulnerable IIS server*
$R(S, T, 80)$ — *Host T is reachable from S on port 80*
intruder effects
$plvl(T) :=$ root — *Root-level privileges on host T*
network effects
$\neg\text{w3svc}_T$ — *Host T is not running IIS*
end

Squid Port Scan. The *Squid* port scan action uses a misconfigured *Squid* web proxy to conduct a port scan of neighboring machines and report the results to the intruder.

action squid-port-scan **is**
intruder preconditions
$plvl(S) =$ user — *User-level privileges on host S*
$\neg$scan — *We have not yet performed a port scan*
network preconditions
squid_T — *Host T is running vulnerable* Squid *proxy*
$R(S, T, 80)$ — *Host T is reachable from S on port 80*
intruder effects
scan — *We have performed a port scan on the network*
network effects
$\oslash$ — *No changes to the network component*
end

LICQ Remote to User. This remote-to-user action immediately gives a remote user a user shell on the target machine. The action rule assumes that a port scan has been performed previously, modeling the fact that such actions typically become apparent to the intruder only after a scan reveals the possibility of exploiting software listening on lesser-known ports.

action LICQ-remote-to-user **is**
intruder preconditions
$plvl(S) \geq$ user — *User-level privileges on host S*
$plvl(T) =$ none — *No user-level privileges on host T*
scan — *We have performed a port scan on the network*
network preconditions
licq_T — *Host T is running vulnerable* LICQ *software*
$R(S, T, 5190)$ — *Host T is reachable from S on port 5190*
intruder effects
$plvl(T) :=$ user — *User-level privileges on host T*
network effects
$\oslash$ — *No changes to the network component*
end

Scripting Action. This remote-to-user action immediately gives a remote user a user shell on the target machine. The action rule does not model the social engineering required to get a user to download a specially-created Web page.

action client-scripting **is**
intruder preconditions
$plvl(S) \geq$ user — *User-level privileges on host S*
$plvl(T) =$ none — *No user-level privileges on host T*
network preconditions
scripting_T — *HTML scripting is enabled on host T*
$R(T, S, 80)$ — *Host S is reachable from T on port 80*
intruder effects
$plvl(T) :=$ user — *User-level privileges on host T*
network effects
$\oslash$ — *No changes to the network component*
end

Local Buffer Overflow. If the intruder has acquired a user shell on the target machine, this action exploits a buffer overflow vulnerability on a *setuid root* file (in this case, the *at* executable) to gain root access.

action local-setuid-buffer-overflow **is**
intruder preconditions
$plvl(T) =$ user — *User-level privileges on host T*

network preconditions
vul-at$_T$ *There is a vulnerable* at *executable*
intruder effects
$plvl(T) := \text{root}$ *Root-level privileges on host* T
network effects
$\oslash$ *No changes to the network component*
end

4.2 Sample Attack Graphs

Figure 4 shows a screenshot of the attack graph generated with our attack graph toolkit for the security property

$$\mathbf{G}\ (intruder.privilege[lin] < root)$$

which states that the intruder will never attain root privileges on the `Linux` host. In Figure 4, a sample attack scenario is highlighted with solid square nodes, with each attack step identified by name and CVE number. Since the external firewall restricts most network connections from the outside, the intruder has no choice with respect to the initial step—it must be a buffer overflow action on the IIS Web server. Once the intruder has access to the Web server machine, his options expand. The highlighted scenario is the shortest route to success. The intruder uses the Web server machine to launch a port scan via the vulnerable *Squid* proxy running on the Linux host. The scan discovers that it is possible to obtain user privileges on the Linux host with the *LICQ* exploit. After that, a simple local buffer overflow gives the intruder administrative control over the `Linux` machine. The last transition in the action path is a bookkeeping step, signifying the intruder's success.

Any information explicitly represented in the model is available for inspection and analysis in the attack graph. For instance, with a few clicks we are able to highlight

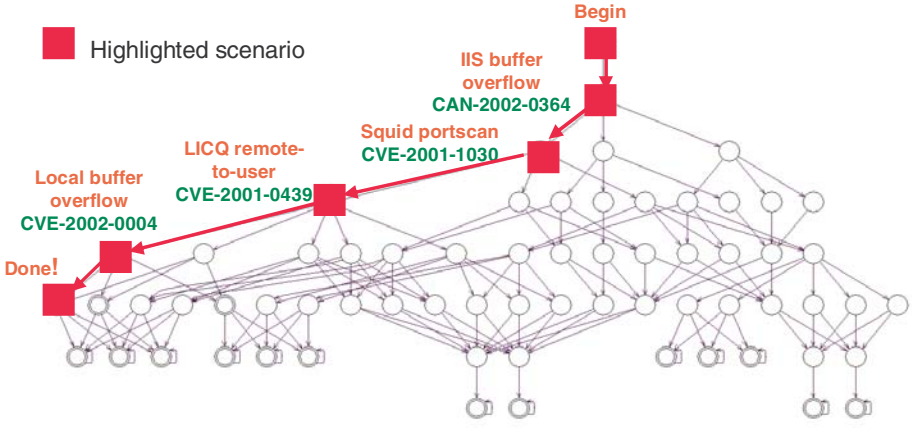

Fig. 4. Example Attack Graph

portions of the graph "covered" by the intrusion detection system. Figure 5 shades the nodes where the IDS alarm has been sounded. These nodes lie on paths that use the *LICQ* action along a network path monitored by the IDS. It is clear that while a substantial portion of the graph is covered by the IDS, the intruder can escape detection and still succeed by taking one of the paths on the right side of the graph. One such attack scenario is highlighted with square nodes in Figure 5. It is very similar to the attack scenario discussed in the previous paragraph, except that the *LICQ* action is launched from the internal `Windows` machine, where the intrusion detection system does not see it. To prepare for launching the *LICQ* action from the `Windows` machine, an additional step is needed to obtain user privileges in the machine. For that, the intruder uses the client scripting exploit on the Windows host immediately after taking over the Web machine.

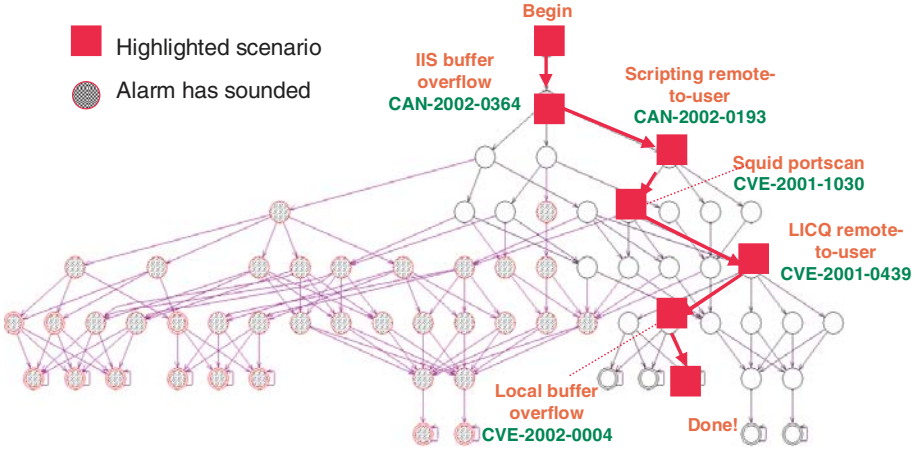

Fig. 5. Alternative Attack Scenario Avoiding the IDS

4.3 Sample Attack Graph Analysis

After generating an attack graph, we can use it to analyze potential effectiveness of various security improvements [16]. To demonstrate the analysis techniques, we expand the example from Sect. 4.1 with an extra host `User` on the external network and several new actions. An authorized user W of the internal network owns the new host and uses it as a terminal to work remotely on the internal `Windows` host. The new actions permit the intruder to take over the host `User`, sniff user W's login credentials, and log in to the internal `Windows` host using the stolen credentials. We omit the details of the new actions, as they are not essential to understanding the examples. Figure 6(a) shows the full graph for the modified example. The graph is significantly larger, reflecting the expanded number of choices available to the intruder.

Single Action Removal. A simple kind of analysis determines the impact of removing one action from the intruder's arsenal. Recall from Sect. 3 that each action is a triple (r, h_s, h_t), where $h_s \in H$ is the host from which the attack is launched, $h_t \in H$ is

the host targeted by the attack, and r is an action rule. The user specifies a set A_{rem} of action triples to be removed from the attack graph. The toolkit deletes the transitions corresponding to each triple in the set A_{rem} from the graph and then removes the nodes that have become unreachable from the initial state.

As demonstrated in Figure 6, this procedure can be repeated several times, reducing the size of the attack graph at each step. The full graph in Figure 6(a) has 362 states. Removing one of two ways the intruder can sniff user W's login credentials produces the graph in Figure 6(b), with 213 states. Removing one of the local buffer overflow actions produces the graph in Figure 6(c), with 66 states. At each step, the user is able to judge visually the impact of removing a single action from the intruder's arsenal.

Critical Action Sets. Once an attack graph is generated, an approximation algorithm can find an approximately-optimal *critical set of actions* that will completely disconnect the initial state from states where the intruder has achieved his goals [16]. A related algorithm can find an approximately-optimal set of security measures that accomplish the same goal. With a single click, the user can invoke both of these exposure minimization algorithms.

The effect of the critical action set algorithm on the modified example attack graph is shown in Figure 7(a). The algorithm finds a critical action set of size 1, containing the port scan action exploiting the *Squid* web proxy. The graph nodes and edges corresponding to actions in the critical set computed by the algorithm are highlighted in the toolkit by shading the relevant nodes. The shaded nodes are seen clearly when we zoom in to inspect a part of the graph on a larger scale (Figure 7(b)).

Since the computed action set is always critical, removing every action triple in the set from the intruder's arsenal is guaranteed to result in an empty attack graph. In the example, we might patch the `Linux` machine with a new version of the *Squid* proxy, thereby removing every action triple that uses the *Squid* port scan rule on the `Linux` machine from the intruder's arsenal.

5 Attack Graph Toolkit

We have implemented a toolkit for generating and exploring attack graphs, using network attack models defined in Sect. 3. In this section we describe the toolkit and show several ways to integrate it with external data sources that supply information necessary to build a network attack model. Specifically, it is necessary to know the topology of the target network, configuration of the network hosts, and vulnerabilities present on the network. In addition, we require access to a database of attack rules to build the transition relation of the attack model. We could expect the user to specify all of the necessary information manually, but such a task is tedious, error-prone, and unrealistic for networks of more than a few nodes.

We recommend deploying the attack graph toolkit in conjunction with information-gathering systems that supply some of the data automatically. We integrated the attack graph generator with two such systems, MITRE Corp's Outpost and Lockheed Martin's ANGI. We report on our experience with Outpost and ANGI in Sections 5.4 and 5.5.

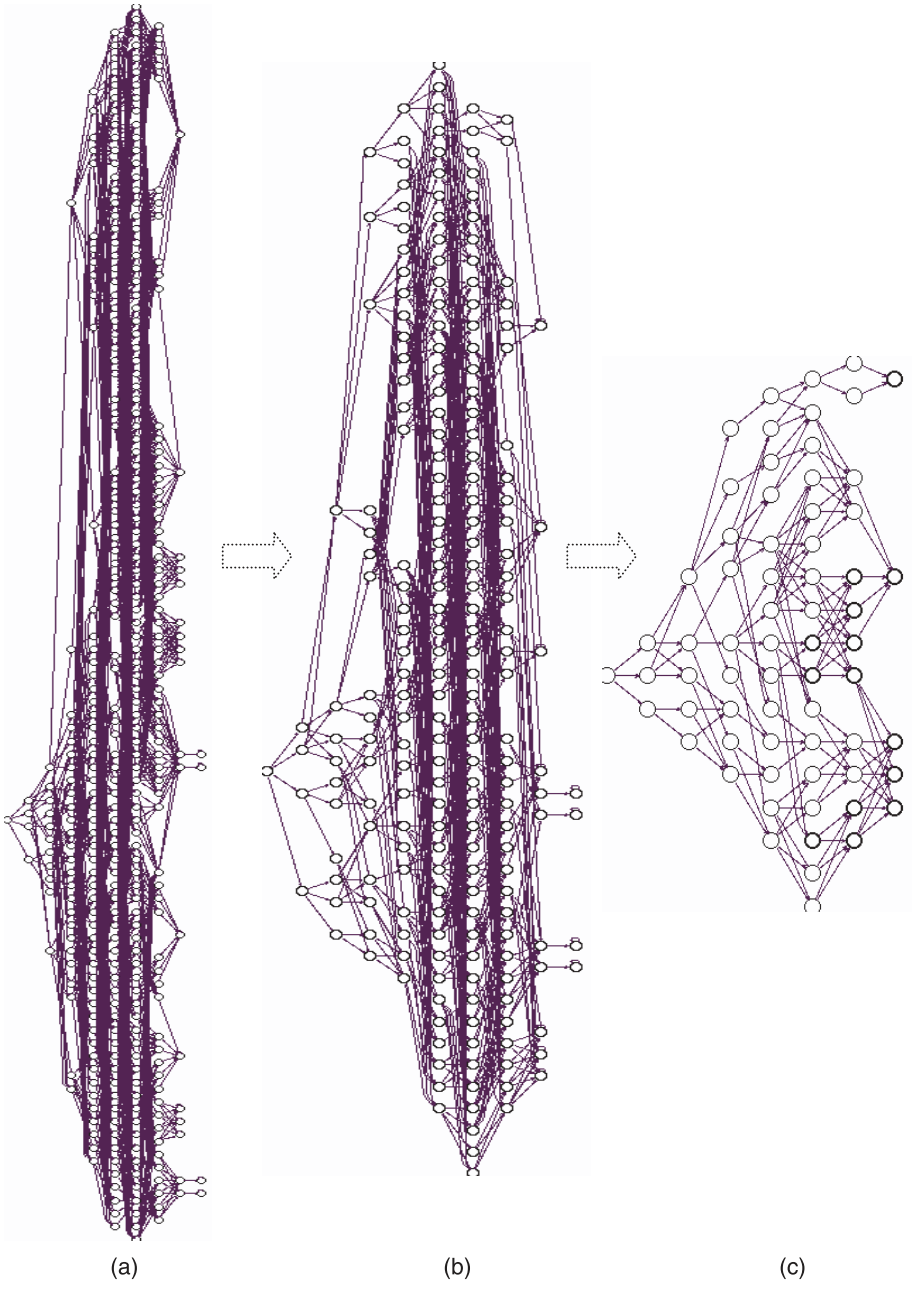

Fig. 6. Reducing Action Arsenal

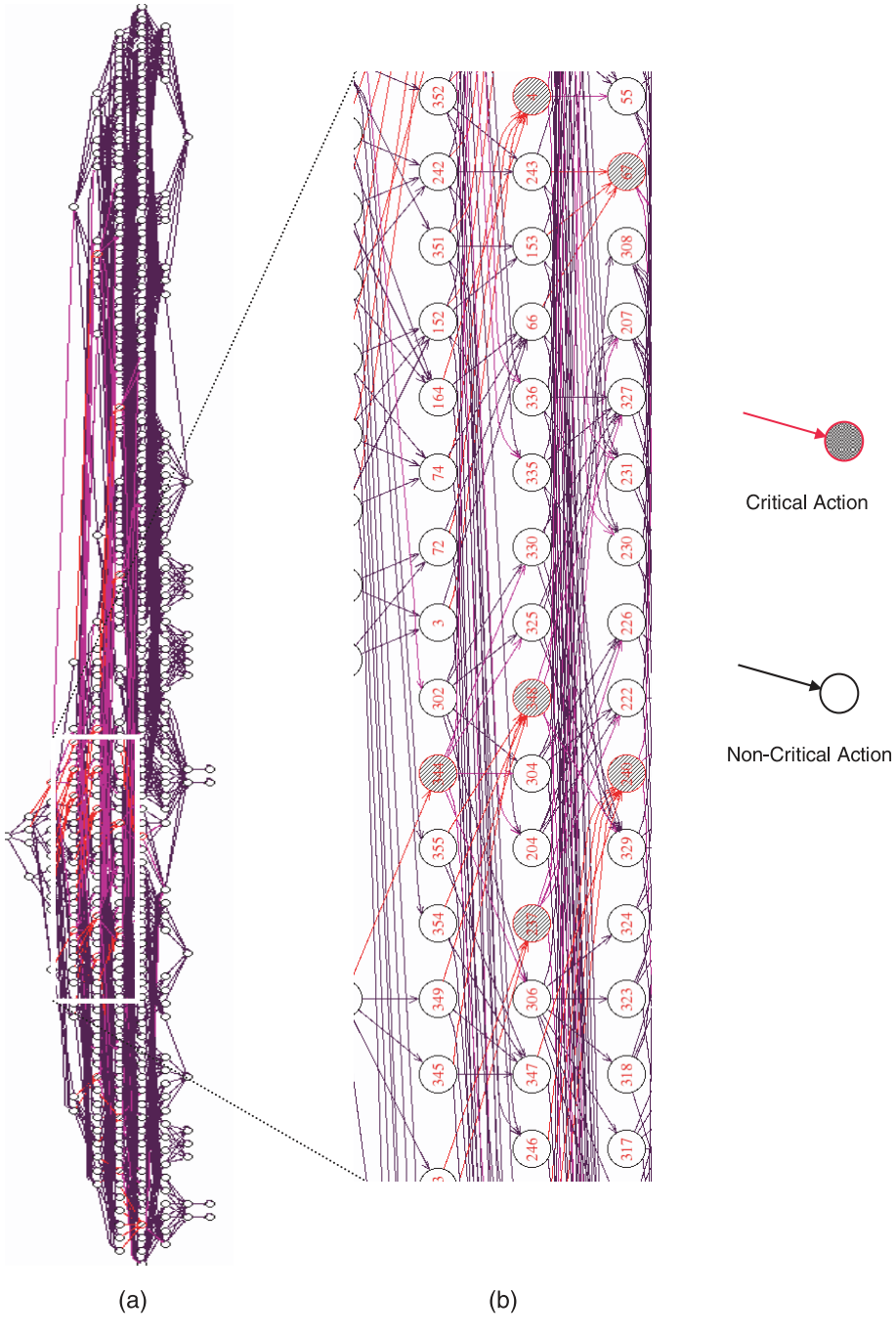

Fig. 7. Finding Critical Action Sets

5.1 Toolkit Architecture

Figure 8 shows the architecture of the attack graph toolkit. There are three main pieces: a *network model builder*, a *scenario graph generator*, and a *graphical user interface* (GUI). The network model builder takes as input information about network topology, configuration data for each networked host, and a library of attack rules. It constructs a finite model of the network suitable for automated analysis. The model is augmented with a security specification, which spells out the security requirements against which the attack graph is to be built. The model and the security specification then go to the second piece, the scenario graph generator.

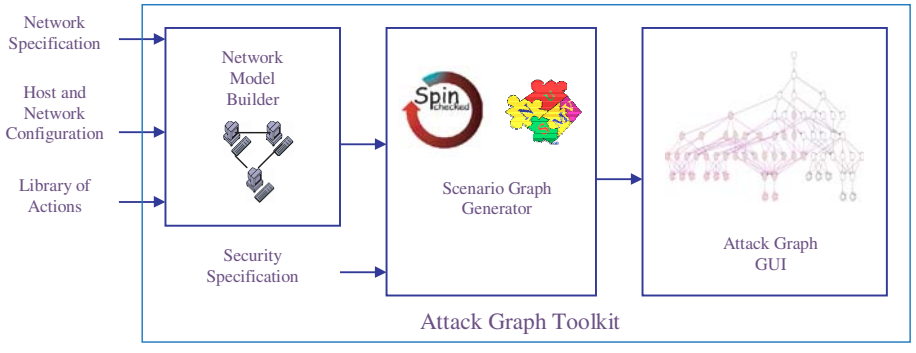

Fig. 8. Toolkit Architecture

The scenario graph generator takes any finite model and correctness specification and produces a graph composed of possible executions of the model that violate the correctness specification. The model builder constructs the input to the graph generator so that the output will be the desired attack graph. The graphical user interface lets the user display and examine the graph.

The model builder's running time is linear in the size of the input specification, typically written in the XML format specified in Sect. 5.2. The algorithm in the scenario graph generator is linear in the size of the output scenario graph [16]. The slowest part of the toolkit is the algorithm that lays out the attack graph on screen. The algorithm uses the network simplex method to find optimal x-coordinates. The simplex method has exponential worst-case performance. The rest of the layout algorithm has cubic complexity. Thus, for large graphs it is sometimes necessary to run analysis algorithms without displaying the full graph on screen.

5.2 The Model Builder

Recall from Sect. 3 that a network attack model consists of six primary components:

1. H, a set of hosts connected to the network
2. C, a connectivity relation expressing the network topology and inter-host reachability
3. T, a relation expressing trust between hosts

4. *I*, a model of the intruder
5. *A*, a set of individual attack actions
6. *Ids*, a model of the intrusion detection system

To construct each of the six components, the model builder needs to collect the following pieces of information. For the entire network, we need:

1. The set of hosts H
2. The network topology and firewall rules, which together induce the connectivity relation C
3. The initial state of the trust relation T: which hosts are trusted by other hosts prior to any intruder action

Several pieces of data are required for each host h in the set H:

4. A unique host identifier (usually name and network address)
5. Operating system vendor and version
6. Active network services with port numbers
7. Common Vulnerabilities and Exposures IDs of all vulnerabilities present on h
8. User-specific configuration parameters

Finally, for each CVE vulnerability present on at least one host in the set H, we need:

9. An attack rule with preconditions and effects

We designed an XML-based format covering all of the information that the model builder requires. The XML format lets the user specify each piece of information manually or indicate that the data can be gathered automatically from an external source. A typical description of a host in XML is as follows:

```
1  <host id="typical-machine" ip="192.168.0.1">
2
3    <services>
4      <ftp port="21"/>
5      <W3SVC port="80"/>
6    </services>
7
8    <connectivity>
9      <remote id="machine1" <ftp/> <W3SVC/> </remote>
10     <remote id="machine2"> <sshd/> <W3SVC/> </remote>
11     <remote id="machine3"> <sshd/> </remote>
12   </connectivity>
13
14   <cve>
15     <CAN-2002-0364/>
16     <CAN-2002-0147/>
17   </cve>
18
19 </host>
```

The example description provides the host name and network identification (line 1), a list of active services with port numbers (lines 3-6), the part of the connectivity relation that involves the host (lines 8-12), and names of CVE and CVE-candidate (CAN) vulnerabilities known to be present on the host (lines 14-17). Connectivity is specified as a list of services that the host can reach on each remote machine. Lines 9-11 each specify one remote machine; e.g., `typical-machine` can reach `machine1` on ports assigned to the *ftp* and *W3SVC* (IIS Web Server) services.

It is unrealistic to expect the user to collect and specify all of the data by hand. In Sections 5.3-5.5 we discuss three external data sources that supply some of the information automatically: the Nessus vulnerability scanner, MITRE Corp.'s Outpost, and Lockheed Martin's ANGI. Whenever the model builder can get a specific piece of information from one of these sources, a special tag is placed in the XML file. If Nessus, Outpost and ANGI are all available at the same time as sources of information, the above host description may look as follows:

```
<host id="typical-machine" ip="|Outpost|">

  <services source="|Outpost|"/>
  <connectivity source="|ANGI|"/>
  <cve source="|Nessus|"/>

</host>
```

The model builder gets the host network address and the list of running services from Outpost, connectivity information from ANGI, and a list of existing vulnerabilities from Nessus. Once all of the relevant information is gathered, the model builder creates a finite model and encodes it in the input language of the scenario graph generator. The scenario graph generator then builds the attack graph.

5.3 Attack Graphs with Nessus

A savvy attacker might use one of the many widely available vulnerability scanners [4] to discover facts about the network and construct an attack scenario manually. Similarly, an attack graph generator can use a scanner to construct such scenarios automatically. Our attack graph toolkit works with the freeware vulnerability scanner Nessus [8] to gather information about reachable hosts, services running on those hosts, and any known exploitable vulnerabilities that can be detected remotely.

The scanner has no internal knowledge of the target hosts, and will usually discover only part of the information necessary to construct a graph that includes every possible attack scenario. Using only an external vulnerability scanner to gather information can lead the system administrator to miss important attack scenarios.

Nevertheless, the administrator can run vulnerability scanners against his own network to find out what a real attacker would discover. In the future, sophisticated intruders are likely to use attack graph generators to help them devise attack scenarios. As a part of network security strategy, we recommend running a vulnerability scanner in conjunction with an attack graph generator periodically to discover avenues of attack that are most likely to be exploited in practice.

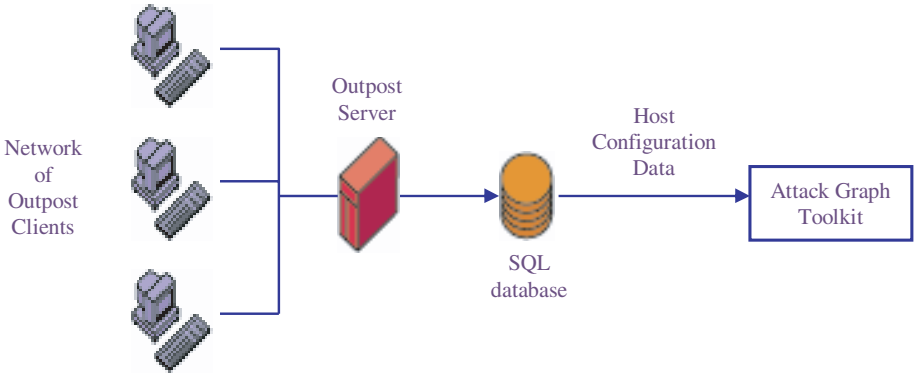

Fig. 9. Outpost Architecture

5.4 Attack Graphs with MITRE Outpost

MITRE Corporation's Outpost is a system for collecting, organizing, and maintaining security-related information on computer networks. It is a suite of inter-related security applications that share a common data model and a common data collection infrastructure. The goal of Outpost is to provide a flexible and open environment for network and system administrators to monitor, control, and protect computer systems.

At the center of the Outpost System is a data collection/probe execution engine that gathers specific configuration information from all of the systems within a network. The collected data is stored in a central database for analysis by the Outpost applications. Outpost collects data about individual hosts only, so it cannot provide information about network topology or attack rules. Since Outpost stores all of the data in a network-accessible SQL database, we retrieve the data directly from the database, without talking to the Outpost server, as shown in Figure 9.

Currently Outpost works with SQL databases supported by Microsoft and Oracle. Both of these packages use a proprietary Application Programming Interface. The model builder includes an interface to each database, as well as a generic module that uses the Open DataBase Connectivity interface (ODBC) and works with any database that supports ODBC. Furthermore, it is easy to add a capability to interface with other types of databases.

An Outpost-populated database contains a list of client hosts monitored by the Outpost server. For the model builder, the Outpost server can provide most of the required information about each individual host h, including:

1. A unique host identifier (usually name and network address)
2. Operating system vendor and version
3. Active network services with port numbers
4. Common Vulnerabilities and Exposures IDs of all vulnerabilities present on h
5. User-specific configuration parameters (e.g., is Javascript enabled for the user's email client?)

Outpost's lists of CVE vulnerabilities are usually incomplete, and it does not keep track of some of the user-specific configuration parameters required by the attack graph

toolkit. Until these deficiencies are fixed, the user must provide the missing information manually.

In the future, the Outpost server will inform the attack graph toolkit whenever changes are made to the database. The tighter integration with Outpost will enable attack graph toolkit to re-generate attack graphs automatically every time something changes in the network configuration.

5.5 Attack Graphs with Lockheed's ANGI

Lockheed Martin Advanced Technology Laboratory's (ATL) Next Generation Infrastructure (ANGI) IR&D project is building systems that can be deployed in dynamic, distributed, and open network environments. ANGI collects local sensor information continuously on each network host. The sensor data is shared among the hosts, providing dynamic awareness of the network status to each host. ANGI sensors gather information about host addresses, host programs and services, and network topology. In addition, ANGI supports vulnerability assessment sensors for threat analysis.

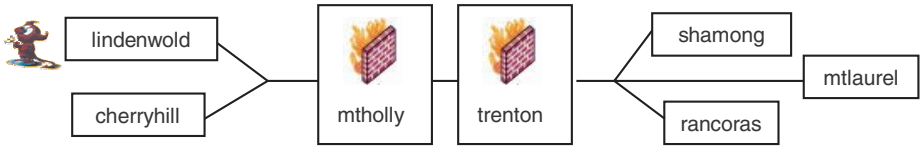

Fig. 10. ANGI Network

Two distinguishing features of ANGI are the ability to discover network topology changes dynamically and focus on technologies for pro-active, automated repair of network problems. ANGI is capable of providing the attack graph model builder with network topology information, which is not available in Outpost and is not gathered by Nessus.

We tested our attack graph toolkit integrated with ANGI on a testbed of five hosts with combinations of the five CVE vulnerabilities specified for the example model in Chapter 4 (p. 352), and one adversary host. Figure 10 is a screenshot of the testbed network schematic. The intruder resides on the host `lindenwold`. Hosts `trenton` and `mtholly` run firewalls, which are initially disabled. We assume that the target of the intruder is the host `shamong`, which contains some critical resource.

ANGI combines information about each host with data from firewall configuration files into a single XML document. To convert firewall rules into a reachability relation C accepted by the attack graph toolkit, ANGI uses a package developed at MITRE Corp. that computes network reachability from packet filter data [14]. The XML file specifies explicitly five attack rules corresponding to the CVE vulnerabilities present on the hosts. ANGI then calls the model builder with the XML document and a security property as inputs. The security property specifies a guarantee of protection for the critical resource host `shamong`:

$$\mathbf{G}(intruder.privilege[shamong] < root)$$

The attack graph generator finds several potential attack scenarios. Figure 11 shows the attack graph as it is displayed by the graphical user interface. The graph consists of 19 nodes with 28 edges.

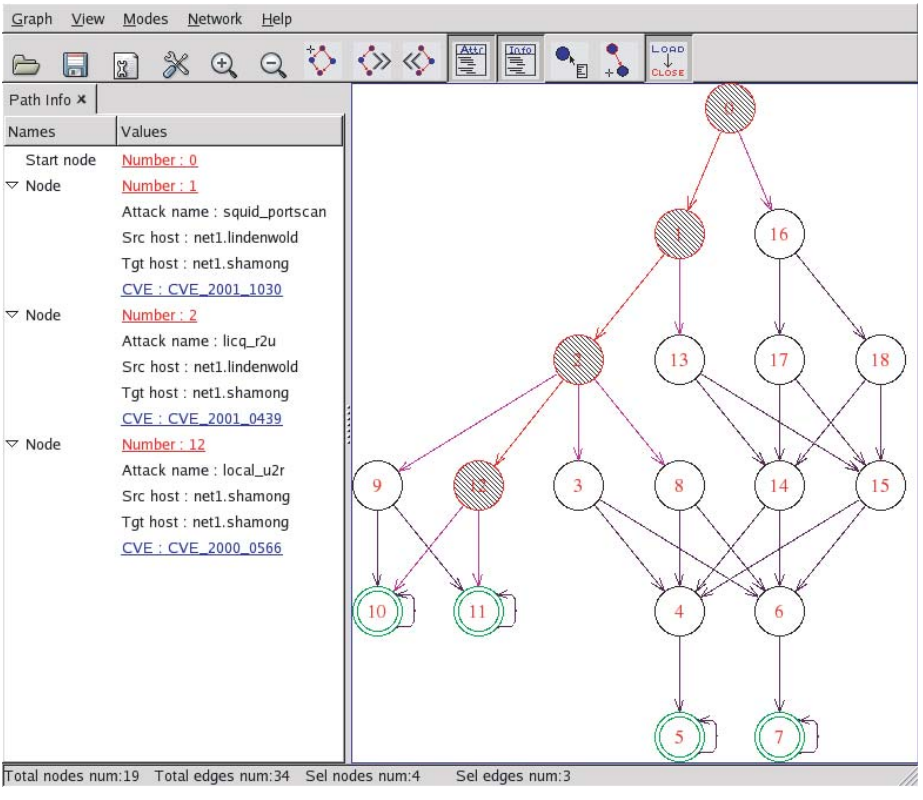

Fig. 11. ANGI Attack Graph - No Firewalls

Exploring the attack graph reveals that several successful attack scenarios exploit the *LICQ* vulnerability on the host `shamong`. One such attack scenario is highlighted in Figure 11. As indicated in the "Path Info" pane on the left of Figure 11, the second step of the highlighted scenario exploits the *LICQ* vulnerability on `shamong`. This suggests a possible strategy for reducing the size of the graph. Using the ANGI interface, we enable the firewall on the host `trenton`, and add a rule that blocks all external traffic at `trenton` from reaching `shamong` on the *LICQ* port. ANGI then generates a new XML model file reflecting this change. The new graph demonstrates a significant reduction in network exposure from this relatively small change in network configuration. The modification reduces graph size to 7 nodes and 6 edges with only two possible paths. (Contrast this new graph with the attack graph shown in Figure 11, which has 19 nodes, 28 edges, and 20 paths.)

Looking at the scenarios in this new graph, we discover that the attacker can still reach `shamong` by first compromising the web server on `cherryhill`. Since we do not want to disable the web server, we enable the firewall on `mtholly` and add a rule specifically blocking `cherryhill`'s access to the *LICQ* client on `shamong`. Yet another invocation of the attack graph generator on the modified model produces an empty attack graph and confirms that we have successfully safeguarded `shamong` while retaining the full functionality of the network.

6 Related Work

Many of the ideas that we propose to investigate have been suggested or considered in existing work in the intrusion detection field. This section surveys recent related work.

Phillips and Swiler [13] propose the concept of attack graphs that is similar to the one described here. However, they take an "attack-centric" view of the system. Since we work with a general modeling language, we can express in our model both seemingly benign system events (such as failure of a link) and malicious events (such as attacks). Therefore, our attack graphs are more general than the one proposed by Phillips and Swiler. Swiler et al. describe a tool [19] for generating attack graphs based on their previous work. Their tool constructs the attack graph by forward exploration starting from the initial state.

The advantage of using model checking instead of forward search is that the technique can be expanded to include liveness properties, which can model service guarantees in the face of malicious activity. For example, a model of a banking network could have a liveness security property such as

$$\mathbf{G}\ (\mathit{CheckDeposited}\ \rightarrow\ (\mathbf{F}\ \mathit{CheckCleared}))$$

which specifies that every check deposited at a bank branch must eventually clear.

Templeton and Levitt [20] propose a requires/provides model for attacks. The model links atomic attacks into scenarios, with earlier atomic attacks supplying the prerequisites for the later ones. Templeton and Levitt point out that relating seemingly innocuous system behavior to known attack scenarios can help discover new atomic attacks. However, they do not consider combining their attack scenarios into attack graphs.

Cuppens and Ortalo [6] propose a declarative language (LAMBDA) for specifying attacks in terms of pre- and post-conditions. LAMBDA is a superset of the simple language we used to model attacks in our work. The language is modular and hierarchical; higher-level attacks can be described using lower-level attacks as components. LAMBDA also includes intrusion detection elements. Attack specifications includes information about the steps needed to detect the attack and the steps needed to verify that the attack has already been carried out. Using a database of attacks specified in LAMBDA, Cuppens and Miege [5] propose a method for alert correlation based on matching post-conditions of some attacks with pre-conditions of other attacks that may follow. In effect, they exploit the fact that alerts about attacks are more likely to be related if the corresponding attacks can be a part of the same attack scenario.

Dacier [7] proposes the concept of privilege graphs. Each node in the privilege graph represents a set of privileges owned by the user; edges represent vulnerabilities. Privi-

lege graphs are then explored to construct attack state graphs, which represents different ways in which an intruder can reach a certain goal, such as root access on a host. He also defines a metric, called the *mean effort to failure* or METF, based on the attack state graphs. Orlato *et al.* describe an experimental evaluation of a framework based on these ideas [12]. At the surface, our notion of attack graphs seems similar to the one proposed by Dacier. However, as is the case with Phillips and Swiler, Dacier takes an "attack-centric" view of the world. As pointed out above, our attack graphs are more general. From the experiments conducted by Orlato *et al.* it appears that even for small examples the space required to construct attack state graphs becomes prohibitive. By basing our algorithm on model checking we take advantage of advances in representing large state spaces and can thus hope to represent large attack graphs.

Ritchey and Ammann [15] also use model checking for vulnerability analysis of networks. They use the (unmodified) model checker SMV [18]. They can obtain only one counter-example, i.e., only one attack corresponding to an unsafe state. In contrast, we modified the model checker NuSMV to produce attack graphs, representing all possible attacks. We also described post-facto analyzes that can be performed on these attack graphs. These analysis techniques cannot be meaningfully performed on single attacks.

Graph-based data structures have also been used in network intrusion detection systems, such as *NetSTAT* [21]. There are two major components in NetSTAT, a set of probes placed at different points in the network and an analyzer. The analyzer processes events generated by the probes and generates alarms by consulting a network fact base and a scenario database. The network fact base contains information (such as connectivity) about the network being monitored. The scenario database has a directed graph representation of various atomic attacks. For example, the graph corresponding to an IP spoofing attack shows various steps that an intruder takes to mount that specific attack. The authors state that "in the analysis process the most critical operation is the generation of all possible instances of an attack scenario with respect to a given target network."

Ammann et. al. present a scalable attack graph representation [1]. They encode attack graphs as dependencies among exploits and security conditions, under the assumption of monotonicity. Informally, monotonicity means that no action an intruder can take interferes with the intruder's ability to take any other actions. The authors treat vulnerabilities, intruder access privileges, and network connectivity as atomic boolean attributes. Actions are treated as atomic transformations that, given a set of preconditions on the attributes, establish a set of postconditions. In this model, monotonicity means that (1) once a postcondition is satisfied, it can never become 'unsatisfied', and (2) the negation operator cannot be used in expressing action preconditions.

The authors show that under the monotonicity assumption it is possible to construct an efficient (low-order polynomial) attack graph representation that scales well. They present an efficient algorithm for extracting minimal attack scenarios from the representation, and suggest that a standard graph algorithm can produce a critical set of actions that disconnects the goal state of the intruder from the initial state.

This approach is less general than our treatment of attack graphs. In addition to the monotonicity requirement, it can handle only simple safety properties. Further, the compact attack graph representation is less explicit, and therefore harder for a human to

read. The advantage of the approach is that it has a worst-case bound on the size of the graph that is polynomial in the number of atomic attributes in the model, and therefore can scale better than full-fledged model checking to large networks.

7 Summary and Current Status

We have designed, implemented, and tested algorithms for automatically generating attack graphs and for performing different kinds of vulnerability analyses on them. We have built an attack graph toolkit to support our generation and analysis algorithms. The toolkit has an easy-to-use graphical user interface. We integrated our tools with external sources to populate our network attack model with host and vulnerability data automatically.

We are in the process of specifying a library of actions based on a vulnerability database provided to us by SEI/CERT. This database has over 150 actions representing many published CVEs. We have preliminary results in using a subset of 30 of these actions as input to our model builder, allowing us to produce attack graphs with over 300 nodes and 3000 edges in just a few minutes. Most telling, is that once graphs are that large, automated analysis, such as the kind we provide, is essential.

With our current toolkit and our growing library of actions, we are now performing systematic experiments: on different network configurations, with different subsets of actions, and for different attacker goals. The ultimate goal is to help the system administrator—by giving him a fast and completely automatic way to test out different system configurations (e.g., network connectivity, firewall rules, services running on hosts), and by finding new attacks to which his system is vulnerable.

References

1. Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *9th ACM Conference on Computer and Communications Security*, pages 217–224, 2002.
2. G. Ausiello, A. D'Atri, and M. Protasi. Structure preserving reductions among convex optimization problems. *Journal of Computational System Sciences*, 21:136–153, 1980.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1985.
4. Cotse.net. Vulnerability Scanners. *http://www.cotse.com/tools/vuln.htm*.
5. Frederic Cuppens and Alexandre Miege. Alert correlation in a cooperative intrusion detection framework. In 23^{rd} *IEEE Symposium on Security and Privacy*, May 2002.
6. Frederic Cuppens and Rodolphe Ortalo. Lambda: A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID)*, number 1907 in LNCS, pages 197–216. Springer-Verlag, 2000.
7. M. Dacier. *Towards Quantitative Evaluation of Computer Security*. PhD thesis, Institut National Polytechnique de Toulouse, December 1994.
8. Renaud Deraison. Nessus Scanner. *http://www.nessus.org*.
9. Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Minimization and reliability analyses of attack graphs. Technical Report CMU-CS-02-109, Carnegie Mellon University, February 2002.

10. Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Two formal analyses of attack graphs. In *Proceedings of the15th IEEE Computer Security Foundations Workshop*, pages 49–63, Nova Scotia, Canada, June 2002.
11. Somesh Jha and Jeannette Wing. Survivability analysis of networked systems. In *Proceedings of the International Conference on Software Engineering*, Toronto, Canada, May 2001.
12. R. Ortalo, Y. Dewarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, September/October 1999.
13. C.A. Phillips and L.P. Swiler. A graph-based system for network vulnerability analysis. In *New Security Paradigms Workshop*, pages 71–79, 1998.
14. John Ramsdell. Frame propagation. MITRE Corp., 2001.
15. R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–165, May 2001.
16. Oleg Sheyner. *Scenario Graphs and Attack Graphs*. PhD thesis, Carnegie Mellon University, 2004.
17. Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.
18. SMV. SMV: A Symbolic Model Checker. `http://www.cs.cmu.edu/~modelcheck/`.
19. L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, June 2000.
20. Steven Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the New Security Paradigms Workshop*, Cork, Ireland, 2000.
21. G. Vigna and R.A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1), 1999.
22. Common Vulnerabilities and Exposures. http://www.cve.mitre.org.