Scalable, Graph-Based Network Vulnerability Analysis[*]

Paul Ammann
ISE Department, MS 4A4
Center for Secure Inf. Sys.
George Mason University
Fairfax, VA 22030, U.S.A.
+1 703 993 1660
pammann@gmu.edu

Duminda Wijesekera
ISE Department, MS 4A4
Center for Secure Inf. Sys.
George Mason University
Fairfax, VA 22030, U.S.A.
+1 703 993 1578
dwijesek@gmu.edu

Saket Kaushik
ISE Department, MS 4A4
George Mason University
Fairfax, VA 22030, U.S.A.
+1 703 993 1632
skaushik@gmu.edu

ABSTRACT

Even well administered networks are vulnerable to attack. Recent work in network security has focused on the fact that combinations of exploits are the typical means by which an attacker breaks into a network. Researchers have proposed a variety of graph-based algorithms to generate attack trees (or graphs). Either structure represents all possible sequences of exploits, where any given exploit can take advantage of the penetration achieved by prior exploits in its chain, and the final exploit in the chain achieves the attacker's goal. The most recent approach in this line of work uses a modified version of the model checker NuSMV as a powerful inference engine for chaining together network exploits, compactly representing attack graphs, and identifying minimal sets of exploits. However, it is also well known that model checkers suffer from scalability problems, and there is good reason to doubt whether a model checker can handle directly a realistic set of exploits for even a modest-sized network. In this paper, we revisit the idea of attack graphs themselves, and argue that they represent more information explicitly than is necessary for the analyst. Instead, we propose a more compact and scalable representation. Although we show that it is possible to produce attack trees from our representation, we argue that more useful information can be produced, for larger networks, while bypassing the attack tree step. Our approach relies on an explicit assumption of monotonicity, which, in essence, states that the precondition of a given exploit is never invalidated by the successful application of another exploit. In other words, the attacker never needs to backtrack. The assumption reduces the complexity of the analysis problem from exponential to polynomial, thereby bringing even very large networks within reach of analysis.

[*]This work supported in part by the National Science Foundation under grants CCR-99-01030, CCS-01113515 and CCR-TC-0208848.

Categories and Subject Descriptors

C.2.0 [**Computer and Communication Networks**]: General—*Security and Protection*; C.2.3 [**Computer and Communication Networks**]: Network Operations—*Network Monitoring*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive Software*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive Software, Unauthorized Access*; K.6.m [**Management of Computing and Information Systems**]: Miscellaneous—*Security*

General Terms

Security

Keywords

Network security, vulnerability, model checking, monotonic analysis, exploit, scalability

1. INTRODUCTION

Commercial vulnerability scanners are quite effective at what they do - namely identifying vulnerabilities in a specific host. However, a variety of authors have noted that identifying vulnerabilities in isolation is only a small part of securing a network, and that a significant issue is identifying which vulnerabilities an attacker can take advantage of through a chain of exploits. For example, an attacker might exploit a defect in a particular version of `ftp` to overwrite the `.rhosts` file on a victim machine. In the next step, the attacker could remotely log in to the victim. In a subsequent step, the attacker could use the victim machine as a base to launch another exploit on a new victim, and so on. There are numerous examples of such chains in the literature, and extensive databases of exploits tailored to specific software and services are available on the web.

Researchers and penetration testers often organize these chains of exploits into graphs or trees. In either case, a designated node (or set of nodes) represents the initial state(s), where a state is defined by assigning a set of values to relevant system attributes, including specific vulnerabilities on various hosts in the network, connectivity between hosts, and attacker access privileges on various hosts. Each transition in the tree (or graph) represents a specific exploit that an attacker can carry out. For example, the 'sshd buffer overflow' exploit, carried out from a specific host controlled by the attacker towards a victim host, lets the attacker obtain root access privileges on the victim, thereby changing the state of the system. Although the precise definitions of attack graph and attack tree vary by author, it is useful to think of an attack tree as a structure in which each possible exploit chain ends in a leaf state that satisfies the attacker's goal, and an attack graph as a consolidation of the attack tree in which some or all common states are merged.

The basic observation behind this paper is that attack graphs can easily be far too large to be practical. Paper [13] provides some support for our position on this: in a scaling exercise with 5 hosts, 8 exploits, and the vulnerabilities associated with those exploits, NuSMV reportedly took 2 hours to execute, with most of that time spent on graph manipulation. The resulting attack graph had 5948 nodes and 68364 edges. The state space in that example was represented with 229 bits.

By contrast, to encode such a problem with the methods presented in this paper, we need, at most, 229 nodes, one for each bit in the state representation. Each of these nodes must be able to store a constant amount of information about however many exploits can change the value of that particular node from 'false' to 'true'. It is clear that our structure is dramatically smaller, even for this relatively limited, from a real world perspective, example. However, we don't lose any of the information in our encoding. That is, we present an algorithm that explodes our structure into an attack tree. At the same time, we aren't required to generate an attack tree (or graph) to carry out our subsequent analysis. We give worst case bounds on our algorithms as we give our presentation.

The cost we pay in this paper for our dramatically smaller data structure is monotonicity. Simply stated, monotonicity means that no action an attacker takes interferes with the attacker's ability to take any other action. We return to the issue of monotonicity throughout the paper. Our basic position is that monotonicity is a reasonable modeling assumption in many network analysis situations.

2. AN SCALABLE, GRAPH-BASED ALGORITHM FOR NETWORK ANALYSIS

We treat vulnerabilities, attacker access privileges, and network connectivity in a way similar to other authors [13, 12], but with some simplification. A vulnerability is a fact about the system that, on the one hand, potentially enables some exploit to be carried out, and on the other, is the result of some exploit. A vulnerability might be running a particular version of some operating system on a given host. Attacker privileges and network connectivity are both straightforward to model; the fact that an attacker has a certain privilege level on a given host is an atomic fact, as is whether two hosts have a type of connectivity required for a given exploit. The simplification is that we group together attacker access privileges, network connectivity, and vulnerabilities into generic *attributes* in our model. Thus, if an attacker has `ftp` access to a given host, we model this as an atomic attribute. Similarly, if a '.rhosts' file includes a given host, we record that as an atomic attribute as well.

We model an *exploit* as an atomic transformation that, given a set of preconditions, estabilishes a set of postconditions. Both preconditions and postconditions in our model are simply attributes. An 'rcp' exploit is illustrated in figure 1. In this exploit, the malicious party needs four attributes on the 'attacker' machine:

1. rcp needs to be available on the attacker,
2. the victim needs to have a trust relationship (via a .rhosts file) with the attacker,
3. the malicious party has a shell on the attacker, and
4. the attacker and the victim need to be connected by the network.

The postcondition, shown on the top, is that the malicious party can move arbitrary files, including programs with which to carry out further exploits, from the attacker to the victim. This exploit model is closely related to the template model of Phillips and Swiler [10].

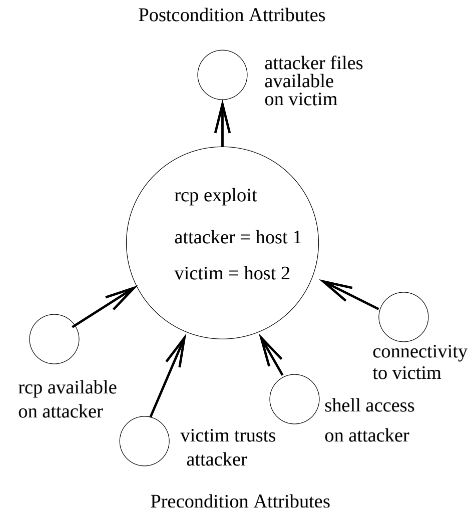

Figure 1: Example Exploit

Our explicit assumption of *monotonicity* has the following implications. First, the precondition of an exploit, once satisfied, never becomes unsatisfied. Said another way, attributes may start in the 'not satisfied' state and become transformed, via some exploit, to the 'satisfied' state, but never the reverse. Second, the monotonicity assumption requires that the negation operator is not used to express the precondition of any exploit. We assume that the preconditions of an exploit are conjoined; this assumption entails no loss of generality, since disjunctions could easily be handled by splitting the exploit into multiple exploits. We also assume that postconditions are conjoined, since it simplifies the algorithms.

One way of thinking about monotonicity is that it means the attacker never has to backtrack. Although there are certainly attacks where monotonicity does not strictly hold, for the most part, these can be *modeled*, with reasonable fidelity, as monotonic. An example may help here. Consider the 'port forward' exploit, in which an unwitting middleman host is used to forward communication from a compromised host to some host that trusts the middleman. To carry out a 'port forward' exploit, it is clearly necessary to have a free port on the middleman, and carrying out the exploit uses up that port. Hence that port is unavailable for a port forwarding attack on a different host; technically the 'port forward' attack is nonmonotonic. However, a clever attacker can often get by with a single port by merely switching back and forth between the two exploits, thereby justifying modeling the 'port forward' exploit monotonically.

Attacks which patch the vulnerability that enables the exploit, such as the 'code green' exploit, may be modeled as monotonic, even though they are clearly not. The rogue machine which initiates the exploit in 'code green' exploit uploads a patch that prevents other attackers from hacking into the host using 'code green'. As in 'port forward', a clever attacker is not impeded in any further attack by using 'code green'. Use of an exploit may lead to the attacker being caught and in such a scenario, the attacker can be prevented from compromising the system. We don't model such scenarios. As a final justification for monotonicity, the Center for Secure Information Systems (CSIS) laboratory has been encoding exploits for the model checker SMV as part of a project that has been ongoing for over a year. The encoded exploits come from training sessions for penetration testers. CSIS has yet to encounter an exploit where the monotonicity assumption wasn't at least as plausible as in the 'port forward' or 'code green' examples described above. While this experience is hardly a guarantee, it does seem to indicate that the assumption of monotonicity is plausible.

2.1 Model

We now turn to our formal model to describe exploits and vulnerabilities. Let $A = \{a_0, a_1, ..., a_N\}$ be the set of attributes. A forms the set of nodes in the graph that is the focus of our manipulations. Let $E = \{e_0, e_1, ..., e_M\}$ be the set of exploits of interest for that network. We consider exploits in a very concrete way, namely an exploit e_j is associated with typically 2 (sometimes 3) specific hosts in the network, where each of these hosts fulfills a particular role. For example, if

```
port-forward(attacker, middleman, victim)
```

is a generic exploit; in our model, concrete exploits would look like:

```
port-forward(Host 1, Maude, Ned)
port-forward(Host 1, Ned, Maude)
port-forward(Maude, Host 2, Ned)
...
```

where each of the three formal parameters binds with all possible hosts, as allowed by connectivity. Note that the 'attacker' may take the role of the 'middleman', or even the 'victim', if this proves useful in compromising the network. From a complexity perspective, the number of concrete exploits derived from a single generic exploit is quadratic in the number of hosts in the network for exploits with two hosts as parameters – a typical case, and cubic in the number of hosts in the network for exploits with three hosts as parameters – a minority case. We are not aware of any exploits that require more than three hosts simultaneously. Some exploits apply to a single host; for example, cracking passwords with a dictionary attack in the case where a shadow password file is not used on that host. In this paper, we count exploits in their instantiated versions, not their generic versions.

The set E forms the edges in our graph, but in a slightly complex way. The reason is that a given exploit can have both multiple preconditions and multiple postconditions. The number of edges labeled e_i is the product of the number of preconditions of e_i and the number of postconditions of e_i. That is, each attribute that is a precondition of e_i has an edge labeled e_i to each postcondition of e_i. The assignment of edges can be done statically, regardless of the initial state of the system. However, for a given initial state, only some of the exploits are feasible, and hence only some of the edges actually materialize.

To encode these edges in a compact way, for each exploit e_i, where e_i has attribute a_j as a postcondition, we label the node description of a_j with e_i. This records the fact that it is possible, under certain circumstances for exploit e_i to establish a_j.

For a given network configuration, we partition the set of attributes into two sets: those that are satisfied in the initial state, and those that require some successful exploit to enable them. The structure described so far appears in figure 2.

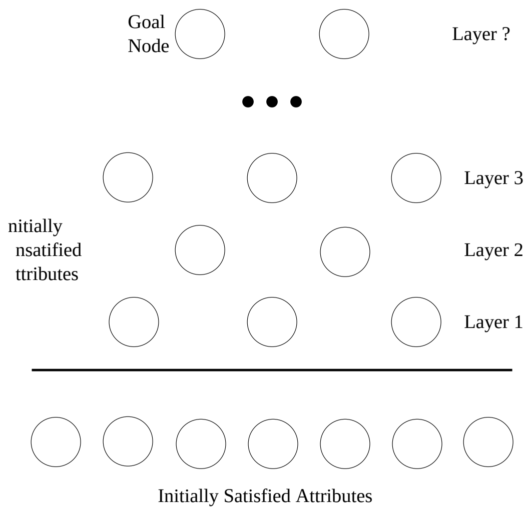

Figure 2: Arrangement of Attributes

The unsatisfied attributes are further structured into layers. Those that can be satisfied by applying a single exploit to the initial state are in layer 1. Those that require at least two exploits, chained together, are in layer 2, and so on. Computing the layer numbers is straightforward: it amounts to a breadth first search of the attributes, starting in the initial state, and adding a layer by considering just the exploits that are satisfied in the prior state. It is possible that some attributes cannot be satisfied by any combination of exploits; we ignore these in subsequent analysis. The layers are also shown in Figure 2.

We model the goal of the attacker as a single attribute, such as 'root access on host J'. Note that this attribute shows up at some layer in the graph, assuming the attacker can reach his or her goal, although not necessarily at the top layer. A generic goal attribute is identified in Figure 2. This is weaker than [13] with the temporal formula specification.

Our graph G is now almost complete; all that remains is to mark attributes with the step in the analysis with which a given exploit first satisfies that attribute. The algorithm is given below. The idea is to revisit the layering description from above. At each layer, the exploits that are 'active' at that layer are applied. If a postcondition attribute is not true, it is set to be true. If it is already true, it is left as true. *In either case*, the exploit label in that attribute is marked with the layer number. That is, if exploit e_7 establishes postcondition a_3 on the 4th layer, then the e_7 label in a_3 is marked 4, and a_3 is set to true, whether or not a_3 has already been set to true by some other exploit in some layer before or equal to 4. (Note that this fact will also be recorded in the label for the appropriate exploit in a_3.) We do not use the goal attribute as a precondition for any exploit. That is, we do not use the fact that the system is compromised by some sequence of exploits to search for further compromise. This restriction removes spurious exploits from subsequent analysis. The layer numbers themselves prove convenient in some of our later analysis algorithms.

```
markAttributes($S, att$):
Input: A set of attributes $A$, a set of exploits $E$
Input: A set of initially satisfied attributes $U \subseteq A$
Ouput: A graph with forward markings
Let $U_n$ be the set of attributes marked at layer $n$
1.Layer 1:
If $U_1 = U \neq \emptyset$, proceed.
    else halt.
2.Layer n:
Repeat
3.  { for each attribute $a_i$ and attack $e_j$
        If preConds($e_j$) $\subseteq U_{n-1}$ then
            If $a_k \in$ postConds($e_j$) and $a_k$ has not been
             marked with $e_j$ at a level $\leq n$.
            mark $a_k$ with attack $e_j$ at level $n$.
    Let all attributes marked at level $n$ be $V_n$.
    $U_n = U_{n-1} \cup V_n$.
    }
until ($V_n = \emptyset$)
Comment: preConds($e_j$), postConds($e_j$) are respectively
 the set of preconditions and postconditions of exploit $e_j$.
```

The runtime of `markAttributes` is $O(\mid A \mid^2 . \mid E \mid)$. This is due to the following reasons.

1. U_n is monotonic and $U_n \subseteq A$. Hence the algorithm converges in at most $\mid A \mid$ steps.

2. Attack e_j is monotonic and preconditions exclude all postconditions. $\texttt{preConds}(e_j) \cap \texttt{postConds}(e_j) = \emptyset$. Also $\texttt{preConds}(e_j) \cup \texttt{postConds}(e_j) \subseteq A$.
   Thus for each layer of the algorithm, step 3 takes at most $\mid A \mid . \mid E \mid$ computations.

2.2 Analysis

We are now ready to analyze our model. First, we present an algorithm to generate a single, minimal, attack from our structure. An *attack* is a sequence of exploits, such that the first exploit is enabled in the initial state, each subsequent exploit is enabled in it's prior state, and the postcondition of the last exploit achieves the attacker's goal. An attack is *minimal* if there is no exploit that can be deleted from the attack without causing some other exploit's preconditions to become false. Consequently, an attack is *minimum* if it has the minimum cardinality over all attacks. [13] showed that finding minimum attacks is NP complete in the number of exploits. Finding a minimal attack is, however, straightforward. Algorithm `findMinimal` finds a minimal attack with the call `findMinimal`($\{goal\}$,$\emptyset$)

```
findMinimal($S, att$):
Global: Attribute/exploit graph with forward markings
Input: A set of attributes to be satisfied $S$
Input: A partial attack $att$
Ouput: A minimal attack satisfying $S$
    If ($S \neq \emptyset$) then
        If there is a set of exploits $E_m = \{e_1, e_2, ...\}$ where
        each $s_i \in S$ is a postcondition of some $e_j \in E_m$.
1.          Choose a minimal set $E'_m \subseteq E_m$.
2.          Prepend each $e \in E'_m$ to $att$ to obtain $att'$.
3.          Select a minimal set $E''_m \subseteq E'_m$.
4.          $S'$ is the set of unsatisfied preconditions in $E''_m$
            return findMinimal($S', att'$);
        Else,
            break out of all recursions and report failure.
    else return $att$
```

Algorithm `findMinimal` deserves a few comments. First, the notion of choosing a minimal set of exploits for a given set of attributes is straight forward, if messy to implement. Each attribute is marked (from the Forward Marking phase), with exactly those exploits that can make it true. So, it is a simple matter to choose one exploit for each attribute. In some cases, an exploit might satisfy multiple attributes, in which case additional choices do not have to be made for those attributes. The partial attack encodes the exploits chosen so far, which in turn identifies attributes that are known to be true, namely the postconditions of the chosen exploits, along with any attributes that are true in the initial state. Note that the running time of `findMinimal` is quadratic in the cardinality of E, the number of exploits.[1] The reason for the quadratic time complexity is that computing a minimal set of exploits in steps 1 and 3 of `findMinimal` may take E^2 many steps in the worst case. In other words, finding a minimal attack is relatively cheap. The following result follows directly.

Result 1: The attack returned by `findMinimal` is a minimal attack.

The next item of interest is to determine all of the exploits that might participate in any attack. The result is algorithm `findAll`, a simple modification of algorithm `findMin`: Algorithm `findAll` finds all exploits that can lead to the attacker's goal with the call `findAll`($\{goal\}$,$\emptyset$)

The `findAll` algorithm doesn't miss any exploits because the forward marking phase guarantees the application of every feasible exploits. `findAll` also doesn't include any extra exploits, because it searches backwards from the goal state, thereby including only exploits whose postconditions advance the state of the attacker. Correspondingly, we have the following results:

Result 2: For any given exploit in an arbitrarily chosen minimal attack, algorithm *findAll* includes that exploit in its result.

Result 3: Every exploit identified by algorithm *findAll* does appear in some minimal attack.

[1]This assumes a constant bound on the number of preconditions or postconditions for a given exploit.

```
findAll($S$,$E_{all}$):
Global: Attribute/exploit graph with forward markings
Input: A set of attributes to be satisfied $S$
Input: A partial result $E_{all}$
Ouput: All exploits that might satisfy $S$
    If ($S \neq \emptyset$) then
        Choose all exploits $E_m$ not already in $E_{all}$
          that have any $s$ in $S$ as a postcondition.
        Add $E_m$ to $E_{all}$
        Let $S'$ be the set of unsatisfied
          preconditions for exploits in $E_{all}$.
        return findAll($S'$,$E_{all}$);
    else return $E_{all}$
```

Given a goal for an attacker, we enhance the algorithm `findMinimal` to find an attack that can be launched with the least number of steps from the initial state. The idea here is to compute the attack given by *minimal* levels provided by the forward marking algorithm.

```
findShort($S, att$):
Global: Attribute/exploit graph with forward markings
Input: A set of attributes to be satisfied $S$
Input: A partial attack set $att$.
Ouput: A short attack set satisfying $S$
  If ($S$) has an unmarked attribute, then abort.
    / $\star$ an attack set cannot be found $\star$ /
  Else for each $s \in S$ let $e_s$ be the attack marked with
   smallest level for $s$.
    let $att'$ be the collection of such smallest level
     attacks, and let $S'$ be the preconditions
     of attacks in $att'$.
  return findShort($S', att'$)
```

Notice that the set of attacks found by `findShort` may not be minimal, but would correspond to the *minimal* level that is necessary to achieve the end goal of the attacker. However, if we now find a minimal cover (as in the case of step 1 and 3 of the algorithm `findMinimal`, we obtain a minimal attack that corresponds to the shortest attack. The worst case run time of `findShort` is O(E. max{ min{ `length`(s):s } s∈S }). As stated, the cost of finding a minimal shortest cost is O(E^2).

2.3 Application

Given the graph structure defined above, what additional analyses are possible? The short answer is the 'standard' list is available. For example, it is possible to associate weights with exploits, thereby leading to notions of expensive (or likely) attack chains. Simple greedy modifications of our search algorithms can find desirable (but not necessarily optimal) attack chains with respect to the metrics.

It is also useful to think in terms of 'cut sets' of either exploits or attributes. These approaches ask the question: what set of exploits (edges) or attributes (nodes) in our graph must be removed to disconnect the goal state from the initial state. Standard graph analysis algorithms can be applied.

3. EXAMPLE

In this section, we consider an example closely related to that in [7]. We model a monotonic variant of that example with our approach. We enumerate attributes and exploits. We show the result of forward reachability analysis on the example, and then give the output of algorithm `findAll`.

The example has an attacker outside a firewall, a router behind the firewall, and two hosts behind the router. We consider three hosts: the attacker (host 0), and the two hosts behind the router (hosts 1 and 2). There is an intrusion detection system that monitors traffic going across the firewall. The result is that `rlogin` sessions from outside the firewall notify the IDS, but `rlogin` sessions between hosts behind the firewall do not.

There are 6 vulnerabilities considered in the example; each vulnerability is possible on each host. Hence, there are $6 * 3 = 18$ attributes in our model corresponding to vulnerabilities. The vulnerabilities are:

ssh_h**:** ssh service is running on host h

ftp_h**:** ftp service is running on host h

$data_h$**:** database is running on host h

$wdir_h$**:** ftp home directory is writable on host h

$fshell_h$**:** ftp user has executable shell on host h

$xterm_h$**:** xterm executable vulnerable to overflow on host h

There is a physical connectivity and two connectivity relations between each pair of hosts: one connectivity relation is for an ftp port and the second is for an $sshd$ port. Altogether, this is three attributes for each pair of hosts. The result is $3 * 3 * 3 = 27$ attributes. There is also a trust relationship between pairs of distinct hosts. In this example, 3 hosts yield 6 trust relationships.

There are three privilege levels for the attacker on any given machine: none, user, and root. For three machines, this yields $3 * 3 = 9$ attributes.

Thus our state space for this example has $18 + 27 + 6 + 9 = 60$ attributes. Most of these attributes do not change value as a result of the exploits. In fact, considering all of the attacks that achieve the goal of 'root privilege on the second host behind the firewall', only a small fraction of the attributes change value. We will enumerate these after we instantiate the exploits.

The example considers 4 generic exploits:

0: sshd buffer overflow

1: ftp .rhosts

2: rsh login

3: local setuid buffer overflow

Examining the first of these is illustrative from the perspective of monotonicity. As given in [7], the exploit is:

- preconditions:
    1. User level privilege for attacker on host S
    2. Less than root level privilege for attacker on host T
    3. Host T is running sshd

4. Host T reachable from S on sshd port

- postconditions:
  1. Root level privilege for attacker on host T
  2. Host T is not running sshd

It is clear that the 2nd precondition is not monotonic. However, modeling it as monotonic has no negative effect. That is, the exploit can be carried out whether or not the attacker already has root privilege on host T; in the former case there is simply no effect. The second postcondition is also not monotonic, since the *sshd* service, which was running, is now no longer running. Again it is reasonable to model the exploit as monotonic, since with root access on T, the attacker can usually restart *sshd* on T. Finally, we model postconditions such as 'root privilege' with multiple postconditions - in this case both 'root privilege' and 'user privilege'.

The first three of these exploits requires a pair of hosts; the last requires only one host. Including each machine paired with itself, each of these generic exploits results in $3 * 3 = 9$ instantiated exploits. In total, therefore, there are $3 * 9 + 3 = 30$ such exploits to consider. Of these, an analysis of the attack graph for this example shows that 8 turn out to be relevant to some minimal attack that achieves the goal. These 8 instantiated exploits are shown below, along with the attributes that each exploit affects.

1. The sshd buffer overflow exploit from hosts:
   - 0 to 1 (root privilege on host 1)
2. The ftp .rhosts exploit from hosts:
   - 0 to 1 (rsh trust from all other hosts to host 1)
   - 0 to 2 (rsh trust from all other hosts to host 2)
   - 1 to 2 (rsh trust from all other hosts to host 2)
3. The rsh rlogin exploit from hosts:
   - 0 to 1 (user privilege on host 1)
   - 0 to 2 (user privilege on host 2)
   - 1 to 2 (user privilege on host 2)
4. The local setuid buffer overflow exploit from hosts:
   - 2 to 2 (root privilege on host 2)

As can be seen from the table, the number of attributes that change value in some attack directed at the goal is quite small: 1 attribute for the first exploit, $2 * 2 = 4$ for the second, 2 for the third, and 1 for the fourth. The total is 8 distinct attributes out of the 54 attributes present. (Other attributes can change value as well, but these do not contribute towards the goal.)

The next step in the example is to show the effects of forward reachability analysis on these 8 attributes. For this purpose, we identify instantiated exploits by the generic exploit number (0 through 3) and the specific source host (0 through 2) and target host (also 0 through 2).

1. Attribute: Root privilege on host 1
   Precondition for: nobody.
   Achieved by $E_0(0,1)$ on round 1.

2. Attribute: User privilege on host 1
   Precondition for: $E_1(1,1)$, $E_1(1,2)$, and $E_2(1,2)$.
   Achieved by $E_0(0,1)$ on round 1.

3. Attribute: .rhosts file on host 1 trusts host 0
   Achieved by $E_1(0,1)$ on round 1.
   Precondition for: $E_2(0,1)$.

4. Attribute: .rhosts file on host 1 trusts host 2
   Achieved by $E_1(0,1)$ on round 1.
   Precondition for: nobody.

5. Attribute: .rhosts file on host 2 trusts host 0
   Precondition for: $E_2(0,2)$.
   Achieved by $E_1(0,2)$ on round 1.
   Achieved by $E_1(1,2)$ on round 2.

6. Attribute: .rhosts file on host 2 trusts host 1
   Precondition for: $E_2(1,2)$.
   Achieved by $E_1(0,2)$ on round 1.
   Achieved by $E_1(1,2)$ on round 2.

7. Attribute: User privilege on host 2
   Precondition for: $E_3(2,2)$.
   Achieved by $E_2(1,2)$ on round 2.

8. Attribute: Root privilege on host 2
   Goal node.
   Achieved by $E_3(2,2)$ on round 3.

We now turn to generating attack chains. We give the output of the algorithm `findAll`(root privilege on host 1, $\emptyset$). There are 3 chains generated. Each chain is minimal, meaning that there is no subset of exploits in the chain that also achieves the goal. Each chain is partially order, so that there is sometimes a choice as to which exploit to attempt next. Detectable exploits are marked. All stealthy attacks must include $E_0(0,1)$. All attacks must end with $E_3(2,2)$. The chains are:

1. $E_0(0,1)$, $E_1(1,2)$, $E_2(1,2)$, $E_3(2,2)$ // stealthy
2. $E_1(0,2)$, $E_2(0,2)$, $E_3(2,2)$ // detectable
3. $E_1(0,1)$, $E_2(0,1)$, $E_1(1,2)$ $E_2(1,2)$ $E_3(2,2)$ // detectable

4. RELATED WORK

There are a variety of commercial tools that scan single hosts for known vulnerabilities. Several prominent ones are COPS [2], SystemScanner [15], and CyberCop [4]. Such scanning tools form a necessary part of a complete system for network vulnerability analysis as they provide the input to our model. Each host on a network hosting network services would expose vulnerabilities to hosts outside the network. Such exposures could be benign or hazardous in nature. Plugging all the vulnerabilities based on the output of a scanning tool may render a network unusable to bonafide users. The focus of the work in this paper is on chaining together the vulnerabilities uncovered by such tools to uncover end-to-end attack scenarios and thus discover the nature of the exposed vulnerabilities.

In addition to tools cited above for analyzing single hosts, two distinct lines of work relate to our contribution here: model checking for network security and direct graph algorithms that manipulate attack graphs or attack trees. We start with the model checking work.

Ramakrishnan and Sekar used a model checker to analyze a single host systems with respect to combinations of unknown vulnerabilities [11]. Ritchey and Ammann used a model checker to provide single attack scenarios to test heterogeneous networks [12] with respect to known exploits such as one might find on `bugtraq`. The model checker SMV was used to produce a counterexample showing a single, possible attack. In an extension of the network vulnerability analysis of Ritchey and Ammann, Jha *et al* [8] and Sheyner *et al* [13] use model checking to analyze attack graphs on heterogeneous networks. As in [12], these authors consider interconnected network of computers with known vulnerabilities that can be combined by hackers in order attack one or more hosts. Every attack goal is encoded as an in computation tree logic (CTL) and model checked using NuSMV. If an attack fails, the authors provide a mechanism, implemented in NuSMV, to produce an attack graph, which includes *all* of the counterexamples that can be analyzed, thereby allowing the analyst to understand *all* possible attack scenarios. Various analyses are performed on the attack graphs. Ordered binary decision diagrams are used for compact representation of attack graphs, but the relevant point for our paper is that these graphs are still likely to be prohibitively large, since, as is usual in model checking applications, the state space is exponential in the number of system variables. The advantage of the monotonic approach proposed here is that the state space is linear in the number of system variables. In particular, it is possible to perform useful analyses without instantiating the full attack graph.

The other line of work addresses the problem of analyzing combinations of network exploits by explicit graph methods. The attack graphs generated in most of these works necessarily carry the problem of exponential size, and the authors are candid about the resulting effect on scalability. The earliest work in this line is the Kuang system [1], and it's extension to a network environment, the NetKuang system [17]. In these systems a backwards, goal-based search scans UNIX systems for poor configurations. The output is a combination of operations that lead to compromise.

The closest work to ours is that of Swiler *et al* [10], which has been implemented in [14]. The attack templates that they use correspond to our generic exploits, and the instantiation of an attack template on a specific network configuration corresponds to our instantiated exploits. Their attack graph is essentially that of Sheyner *et al* (cited above), with edges being steps in an attack, and nodes being valuations of system variables. Swiler *et al* eliminate redundant paths from their attack graphs with what amounts to an assumption of monotonicity, namely that the order of attacks does not matter. While their approach helps control the visited parts of the state space, it does not eliminate its fundamental exponential character. Swiler *et al* also consider cost functions, which we do not; it would be interesting to apply their techniques to our model.

Templeton and Levitt [16] develop a 'requires/provides' model for modeling chains of network attacks and reducing the complexity of such a model. The 'requires' part corresponds to our exploit preconditions, and the 'provides' part corresponds to our exploit postconditions. They provide a language for specifying exploits. Their ideas on reducing the complexity of the attack tree can be used within our model to further reduce the complexity of the attack graph. In future work, we would like to abstract multiple vulnerabilities on a host, with similar effects, to a generic vulnerability and hence simplify the analysis. Similarly, we can abstract a group of hosts with similar abstract vulnerabilities as a single node to simplify the attack graph. Dawkins *et al* [6] also provide a language for modeling exploits, and they use this language to provide a hierarchical view of attack trees. The hierarchy helps present information to the user in a more manageable way, but the approach does not have the 'goal' directed aspect of Sheyner *et al*, Swiler *et al*, and our work. That is, their search is a forward exploration of the state space, and hence includes all possible compromises, and not just those of interest to a specific attack goal.

Dacier *et al* [5] and Ortalo *et al* [9] also used graph analysis for network security evaluation. The result in their work is a notion of Mean Effort To security Failure (METF), a probabilistic metric based on assigning likelihoods to attacks. The work in Ortalo *et al* has a substantial implementation effort associated with it, and our methods may fit well into that framework.

Cuppens and Miege [3] use a similar approach to model the attacks in the *Intrusion Detection Framework.* They provide a formal basis to the chaining of attacks and go on to correlate alerts generated on that basis. They use `LAMBDA` as the modeling language to model their attacks which could be a good approach to make the analysis portable on different platforms and make the model generation easier.

Of course, graph analysis is a common computer science technique, and a variety of authors have developed graph based algorithms for different aspects of computer security. We have described the directly relevant cases above. Prior uses of graph-based approaches for other aspects of computer security are well described in the references listed above.

5. CONCLUSIONS

In this paper, we have used an assumption of monotonicity to obtain a concise, scalable graph-based representation for encoding attack trees. Without actually enumerating these trees, we can identify minimal attacks chains, including attack chains which are 'short' in the sense that they require the least number of (possibly parallel) steps from the initial state. We have achieved the above results in a prototype implementation with the algorithms discussed above implemented using Java programming language. Our representation avoids the exponential explosion problem associated with defining states as valuations of system variables, thereby making our approach a candidate for realistic networks with tens or hundreds of hosts.

6. REFERENCES

[1] R. Baldwin. Kuang: Rule based security checking. Technical report, MIT Lab for Computer Science, Programming Systems Research Group, May 1994.

[2] Computer Oracle and Password System (COPS). ftp.cert.org/pub/tools/cops.

[3] F. Cuppens and A.Miege. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (Oakland 2002)*, pages 187–200, Oakland, CA, May 2002.

[4] Network associates: CyberCop Scanner. www.nai.com/asp_set/products/tns/ccscanner_intro.asp.

[5] M. Dacier, Y. Deswartes, and M. Kaaniche. Quantitive assessment of operational security models and tools. Technical Report Research Report 96493, LAAS, May 1996.

[6] J. Dawkins, C. Campbell, and J. Hale. Modeling network attacks: Extending the attack tree paradigm. In *Workshop on Statistical and Machine Learning Techniques in Computer Intrusion Detection*, Johns Hopkins University, June 2002. Center for Information Security, University of Tulsa.

[7] S. Jha, O. Sheyner, and J. Wing. Minimization and reliability analysis of attack graphs. Technical Report CMU-CS-02-109, School of Computer Science, Carnegie Mellon University, February 2002.

[8] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *Proceedings of the 2002 Computer Security Foundations Workshop*, pages 45–59, Nova Scotia, June 2002.

[9] R. Ortalo, Y. Deswarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, September/October 1999.

[10] C. Phillips and L. Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the New Security Paradigms Workshop*, pages 71–79, Charlottesville, VA, 1998.

[11] C. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, September 1998.

[12] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (Oakland 2000)*, pages 156–165, Oakland, CA, May 2000.

[13] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (Oakland 2002)*, Oakland, CA, May 2002.

[14] L. Swiler, C. Phillips, D. Ellis, , and S. Chakerian. Computer-attack graph generation tool. In *Proceedings DISCEX '01: DARPA Information Survivability Conference & Exposition II*, pages 307–321, June 2001.

[15] Internet security systems: System scanner. www.iss.net.

[16] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the New Security Paradigms Workshop*, Cork, Ireland, September 2000. http://seclab.cs.ucdavis.edu/papers/NP2000-rev.pdf.

[17] D. Zerkle and K. Levitt. Netkuang - A multi-host configuration vulnerability checker. In *Proceedings of the 6th USENIX Unix Security Symposium*, San Jose, CA, 1996.