Available online at www.sciencedirect.com

ScienceDirect

Computer Communications 29 (2006) 3812–3824

computer communications

www.elsevier.com/locate/comcom

Minimum-cost network hardening using attack graphs

Lingyu Wang *, Steven Noel, Sushil Jajodia

Center for Secure Information Systems, George Mason University, Fairfax, VA 22030-4444, USA

Received 11 May 2006; received in revised form 22 June 2006; accepted 26 June 2006
Available online 18 July 2006

Abstract

In defending one's network against cyber attack, certain vulnerabilities may seem acceptable risks when considered in isolation. But an intruder can often infiltrate a seemingly well-guarded network through a multi-step intrusion, in which each step prepares for the next. *Attack graphs* can reveal the threat by enumerating possible sequences of exploits that can be followed to compromise given critical resources. However, attack graphs do not directly provide a solution to remove the threat. Finding a solution by hand is error-prone and tedious, particularly for larger and less secure networks whose attack graphs are overly complicated. In this paper, we propose a solution to automate the task of hardening a network against multi-step intrusions. Unlike existing approaches whose solutions require removing exploits, our solution is comprised of initially satisfied conditions only. Our solution is thus more enforceable, because the initial conditions can be independently disabled, whereas exploits are usually consequences of other exploits and hence cannot be disabled without removing the causes. More specifically, we first represent given critical resources as a logic proposition of initial conditions. We then simplify the proposition to make hardening options explicit. Among the options we finally choose solutions with the minimum cost. The key improvements over the preliminary version of this paper include a formal framework of the minimum network hardening problem, and an improved one-pass algorithm in deriving the logic proposition while avoiding logic loops.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Intrusion detection; Vulnerability analysis; Intrusion prevention

1. Introduction

Attackers typically employ multiple attacks to evade security measures and to gradually gain privileges and approach the final goal. Such a multi-step network intrusion can often infiltrate even a seemingly well guarded network. Isolated vulnerabilities reported by vulnerability scanners, such as Nessus [3], may not seem to be a serious threat until they are cleverly combined by attackers. The completeness of a penetration testing usually heavily depends on techniques of the red team, and is prone to human errors.

Existing approaches build *attack graphs* to represent attack paths, i.e., the possible sequences of vulnerabilities that attackers may exploit during a multi-step intrusion. However, while attack graphs reveal the threats, but they do not directly provide a solution to harden the network against them. Removing vulnerabilities usually incurs different costs, and in practice it is usually infeasible to remove all identified vulnerabilities. A critical but unanswered question in defending against multi-step intrusions is thus: *which of the vulnerabilities should be removed, such that none of the attack paths leading to given critical resources can be realized, where such removal incurs the least cost?* Finding an answer to this question manually is error-prone and tedious, and becomes infeasible for larger and less secure networks whose attack graphs are too complicated.

One recent effort aims to compute a minimal set of vulnerabilities as the solution to harden the network [15,6]. However, such a solution is not directly enforceable, because some of the vulnerabilities are consequences of exploiting other vulnerabilities, and the consequences cannot be removed without first removing the causes. For example, the solution may require an FTP-related

* Corresponding author. Tel.: +1 703 993 3931.
E-mail address: lwang3@gmu.edu (L. Wang).

vulnerability to be removed. The vulnerability depends on the existence of the vulnerable FTP service on the destination host and the FTP access privilege for source hosts, and the latter may further depend on other vulnerabilities on the source hosts. Clearly, there are multiple choices with different costs in removing this single vulnerability. This shows that a minimal set of vulnerabilities is not necessarily a minimal solution, considering the vulnerabilities they may implicitly depend on.

In this paper, we propose a different method that takes into account the dependency relationships among vulnerabilities in deriving hardening solutions. More specifically, we view each vulnerability as a Boolean variable, and we derive a logic proposition to represent the negation of given critical resources in terms of initially satisfied security-related conditions (or *initial conditions* for short). This proposition is thus the necessary and sufficient condition for protecting the critical resources. To make hardening options explicit, we transform this logic proposition into its disjunctive normal form (DNF). Each disjunction in the DNF provides a different option in hardening the network. We then choose options with the minimum costs based on given assumptions on the cost of initial conditions.

Our solution removes the previously mentioned limitation of existing approaches, because the hardening options require disabling initial conditions only. Each initial condition can be independently disabled because they do not depend on other vulnerabilities or conditions. For example, instead of requiring the removal of an FTP vulnerability, our solution may required disabling the vulnerable FTP service or denying FTP accesses to certain hosts, which are both readily enforceable. In the simplification of the logic proposition, we can identify seemingly relevant initial conditions whose removal does not really help to protect the critical resources. Such insights are important in keeping the cost of network hardening minimal, but they are also impossible to obtain in previous approaches.

The preliminary version of this paper has outlined the basic idea and method [9]. The key improvements in the current paper are as follows. First, we formalize the notation of attack graph and clearly define the minimum-cost network hardening problem. Second, instead of depending on an extra forward search to remove cycles in attack graphs, we propose a different algorithm that searches the attack graph and removes cycles all in one-pass. This approach removes the difficulty of the previous method in dealing with cycles that cannot be easily removed in the forward search. As a side benefit, it also improves the performance by approximately 50% through saving the preprocessing step of forward search.

The rest of this paper is organized as follows. The next section reviews related work. Section 3 provides a formal framework of the attack graph and an example to motivate our study. Section 4 states the problem of network hardening and derives a solution based on graph searching. Section 5 provides a case study to illustrate the proposed method. Finally, Section 6 concludes the paper.

2. Related work

A number of tools are available for scanning network vulnerabilities, such as Nessus [3], but most of them can only report isolated vulnerabilities. On the research front, attack graphs are constructed by analyzing the inter-dependency between vulnerabilities and security conditions that have been identified in the target network [4,18,11,2,10,13,16,14,1,15,5]. Such analysis can be either forward starting from the initial state [11,16] or backward from the goal state [13,15]. Model checking was first used to analyze whether the given goal state is reachable from the initial state [13,12] but later used to enumerate all possible sequences of attacks between the two states [15,6].

The explicit attack sequences produced by a model checker face a serious scalability issue, because the number of such sequences is exponential in the number of vulnerabilities multiplied by the number of hosts. To avoid such combinatorial explosion, a more compact representation of attack graphs was proposed in [1]. The *monotonicity assumption* underlies this representation, i.e., an attacker never relinquishes any obtained capability. This newer representation can thus keep exactly one vertex for each exploit or security condition, leading to an attack graph of polynomial size (in the total number of vulnerabilities and security conditions). In this paper, we shall assume such a compact representation of the attack graph.

Algorithms exist to find the set of exploits from which the goal conditions are reachable [1]. This eliminates some irrelevant exploits from further consideration because they do not contribute to reaching the goal condition. However, as we show in Section 5, this result may still include many irrelevant exploits, even though the goal condition is reachable from them. The reason lies in that the reachability is only a necessary but not sufficient condition for an exploit to actually contribute to reaching the goal condition. On the other hand, our solution is necessary and sufficient for a goal condition to be satisfied.

Closest to our work, the *minimal critical attack set* is a minimal set of exploits in an attack graph whose removal prevents attackers from reaching any of the goal states [15,6,1]. The minimal critical attack set thus provides solutions to harden the network. However, their method ignores the critical fact that consequences cannot be removed without removing the causes. The exploits in their solutions usually depend on other exploits that also need to be disabled. The solution is thus not directly enforceable. Moreover, after taking into account those implied exploits the solution is no longer minimum. Our method fixes this problem by including only initial conditions in the solution. The initial conditions can be independently disabled, leading to a readily deployable solution.

Attack graphs have been used for correlating intrusion alerts into attack scenarios [8,17]. Such alert correlation methods are parallel to our work, because they aim to employ the knowledge encoded in attack graphs for detecting and taking actions against actual intrusions, whereas our work aims to harden the network before any intrusion may happen. The relationship between those methods and our work is analogous to that between IDSs and vulnerability scanners, although IDSs and vulnerability scanners work on the alert and vulnerability level, whereas the alert correlation methods and our methods work at a higher level, i.e., the attack scenarios. Moreover, previous methods of alert correlation via attack graphs do not consider the minimal-cost hardening problem, as we do here.

3. Preliminaries

This section first reviews the concepts of attack graphs. Then it shows an example to motivate further studies.

3.1. Attack graph

Attack graphs represent prior knowledge about vulnerabilities, their dependencies, and network connectivity. There are two different representations possible for an attack graph. First, an attack graph can explicitly enumerate all possible sequences of vulnerabilities an attacker can follow, i.e., all possible *attack paths* [15,6]. However, such graphs face a combinatorial explosion in the number of attack paths. Second, with a monotonicity assumption stating an attacker never relinquishes an obtained capability, an attack graph can record the dependency relationships among vulnerabilities and keep attack paths implicitly without losing any information [1]. The resulting attack graph has no duplicate vertices and hence has a polynomial size in the number of vulnerabilities multiplied by the number of connected pairs of hosts. We shall assume this latter notion of attack graphs.

An attack graph can be represented as a directed graph with two type of vertices, *exploits* and *security conditions* (or simply *conditions* when no confusion is possible). Most generally, we denote an exploit as a predicate $v(h_s,h_m,h_d)$. This indicates an exploitation of the vulnerability v on the destination host h_d, initiated from the source host h_s, through an intermediate host h_m. Similarly, we write $v(h_s, h_d)$ or $v(h)$, respectively, for exploits involving two hosts (no intermediate host) or one (local) host.

A security condition is a predicate $c(h_s,h_d)$ that indicates a satisfied security-related condition c involving the source host h_s and the destination host h_d (when an exploit involves a single host, we simply write $c(h)$). Examples of security conditions include the existence of a vulnerability or the connectivity between two hosts.

There are two types of directed edges that inter-connect exploits with conditions (but no edges directly between exploits or directly between conditions). First, an edge can point from a condition to an exploit. Such an edge denotes the *require* relation, which means the exploit cannot be executed unless the condition is satisfied. Second, an edge pointing from an exploit to a condition denotes the *imply* relation, which means executing the exploit will satisfy the condition. For example, an exploit usually requires the existence of the vulnerability on the destination host and the connectivity between the two hosts. We formally characterize attack graphs in Definition 1.

Definition 1. Given a set of exploits E, a set of conditions C, a *require* relation $R_r \subseteq C \times E$, and an *imply* relation $R_i \subseteq E \times C$, an **attack graph** G is the directed graph $G(E \cup C, R_r \cup R_i)$ ($E \cup C$ is the vertex set and $R_r \cup R_i$ the edge set).

One important aspect of attack graphs is that the require relation is always conjunctive, whereas the imply relation is always disjunctive. More specifically, an exploit cannot be realized until *all* of its required conditions have been satisfied, whereas a condition is satisfied if *any* of the realized exploits implies the condition. Exceptions to the above requirements do exist. First, an exploit with multiple variations may require different sets of conditions, whence the require relation for this exploit is disjunctive (between these sets of conditions). This case can be handled by having a separate vertex for each variation of the exploit such that the require relation for each variation is still strictly conjunctive.

On the other hand, a collection of exploits may jointly imply a condition whereas none of them alone can do so, whence the imply relation becomes conjunctive for this condition. This case can be handled by inserting dummy conditions and exploits to capture the conjunctive relationship. For example, suppose both e_1 and e_2 are required to make a condition c satisfied. We insert two dummy conditions c_1 and c_2 and a dummy exploit e_3 into the attack graph. The edges are inserted such that e_1 and e_2 imply c_1 and c_2, respectively, and c_1 and c_2 are required by e_3, which in turn implies c. Now the conjunctive relationship that both e_1 and e_2 are required for c to be satisfied is encoded in the fact that e_3 requires both c_1 and c_2. As will be discussed later, the result of our methods includes only those conditions that are not implied by any exploit, and hence introducing dummy conditions and exploits does not affect the effectiveness of our methods.

3.2. A motivating example

Fig. 1 depicts an example of attack graphs, which is similar to those in [15,1,6]. Some modeling simplifications have been made, such as combining transport-layer ftp connectivity, physical-layer connectivity, and the existence of the ftp daemon into a single ftp condition [14,6]. In the figure, exploits appear as ovals, and conditions as plain text (with the goal condition shaded). The details of the attack scenario (for example, network topology, services, and operating systems) are omitted here and can be found elsewhere [15,1].

Fig. 1 is a relatively simple scenario with three hosts and four vulnerabilities. However, because multiple interleaved attack paths can lead to the goal condition, an optimal solution to harden the network is still not apparent from the attack graph itself, and finding such a solution by hand

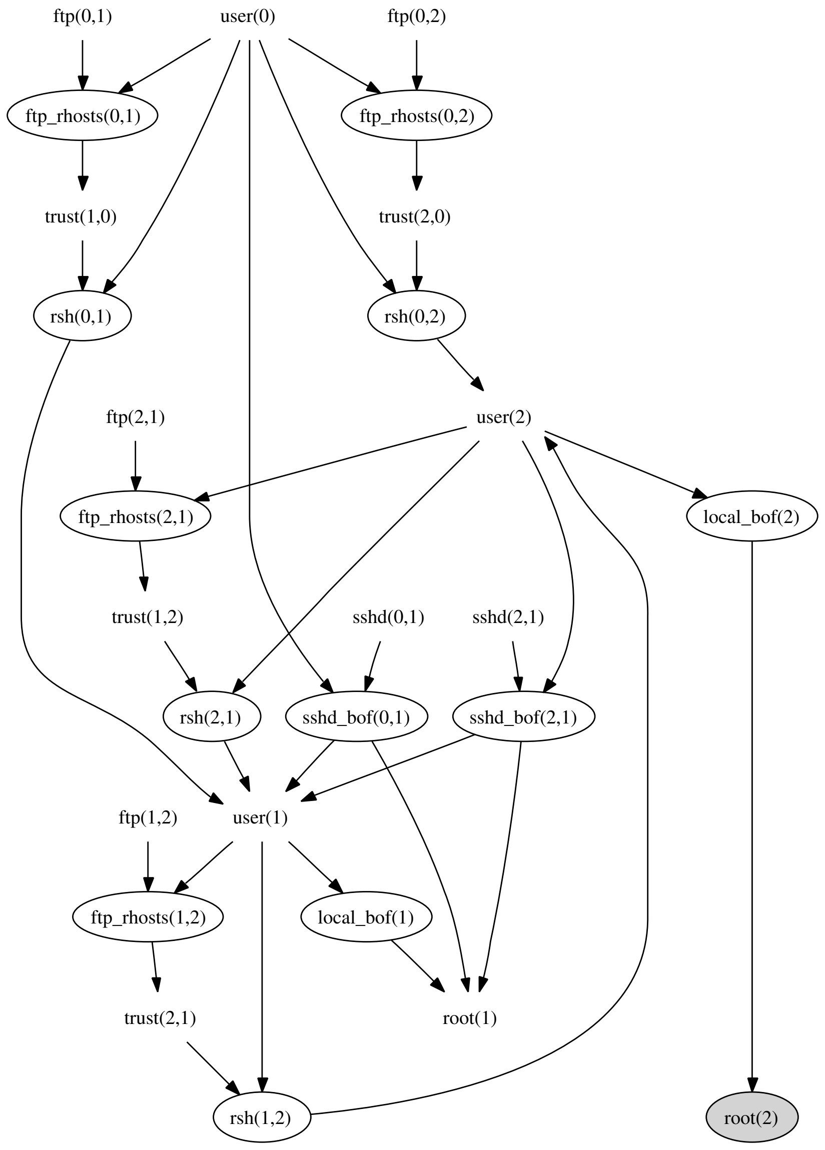

Fig. 1. An example attack graph.

may not be trivial, either. As an example of attack paths, the attacker can first establish a trust relationship from his machine (host 0) to host 2 (the condition *trust*(2, 0)) via the ftp. rhosts vulnerability on host 2 (the exploit *ftp_rhosts*(0, 2)), then gain user privilege on host 2 (the condition *user*(2)) with an rsh login (the exploit *rsh*(0, 2)), and finally achieve the goal condition *root*(2) using a local buffer overflow attack on host 2 (the exploit *local_bof*(2)). The following are some of the valid attack paths that can be generated using existing algorithms [1].

- *ftp_rhosts*(0, 2), *rsh*(0, 2), *local_bof*(2)
- *ftp_rhosts*(0, 1), *rsh*(0, 1), *ftp_rhosts*(1, 2), *rsh*(1, 2), *local_bof*(2)
- *sshd_bof*(0, 2), *ftp_rhosts*(1, 2), *rsh*(1, 2), *local_bof*(2)

Intuitively, to prevent the goal condition from being satisfied, a solution to network hardening must break all the attack paths leading to the goal. This intuition was captured by the concept of *critical set*, that is, a set of exploits (and corresponding conditions) whose removal from the attack graph will invalidate all attack paths [15,6]. It has also been shown that finding critical sets with the minimum cardinality is NP-hard, whereas finding a minimal critical set (that is, a critical set with no proper subset being a critical set) is polynomial. Based on the above attack paths, there are many minimal critical sets, such as {*rsh*(0, 2), *rsh*(1, 2)}, {*ftp_rhosts*(0, 2)}, *rsh*(1, 2)}, {*ftp_rhosts*(1, 2), *rsh*(0, 2)}, and so on. If any of those sets of exploits could be completely removed, all the attack paths will become invalid, and hence the goal condition is safe.

Unfortunately, the above solution ignores the following important fact. Not all exploits are under the direct control of administrators. An exploit can only be removed by disabling its required conditions, but not all conditions can be disabled at will. Intuitively, a consequence cannot be removed without removing its causes. Some conditions are implied by other exploits. Such *intermediate* conditions cannot be independently disabled without removing those exploits that imply them. Only those *initial* conditions that are not implied by any exploit can be disabled independently of other exploits or conditions. Hence, it is important to distinguish between those two kinds of conditions, as formally stated in Definition 2.

Definition 2. In an attack graph $G(E \cup C, R_r \cup R_i)$, **initial** conditions refer to the subset of conditions $C_i = \{c \mid$ there does not exist $e \in E$ such that $(e,c) \in R_i\}$, whereas **intermediate** conditions refer to the subset $C - C_i$.

In Fig. 1, each of the rsh exploits requires intermediate conditions only, and hence cannot be removed without first removing other exploits. For example, the exploit $rsh(1,2)$ cannot be independently removed, because the two conditions it requires, $trust(2,1)$ and $user(1)$, are both intermediate conditions and cannot be independently disabled. As long as an attacker can satisfy those two conditions through other exploits (for example, $ftp\_rhosts(1,2)$ and $sshd\_bof(2,1)$), the realization of the exploit $rsh(1,2)$ is unavoidable. In practice, although one can stop the rsh service on host 2 to remove this exploit, this action adversely reduces the availability of usable service to normal users. Hence, any of the above minimal critical sets, such as $\{rsh(0,2), rsh(1,2)\}$, is theoretically a sound solution, but practically not enforceable.

4. Deriving solutions to harden the network

We first state the network hardening problem in Section 4.1, and we then provide the solution in Section 4.2.

4.1. The network hardening problem

The discussions in Section 3.2 show that a network-hardening solution based on exploits or intermediate conditions cannot be easily enforced. This observation motivates us to pose the following question: *which of the initial conditions must be disabled, if the goal conditions are never to be satisfied?* Our solution will thus be actually possible to implement, because it only includes initial conditions, which can be independently disabled. To more formally state the above problem, it is convenient to interpret an attack graph as a simple logic program as follows. Each exploit or condition in the attack graph is interpreted as a logic variable. The interdependency between exploits and conditions now becomes logic propositions involving the two connectives *AND* and *OR*, with *AND* between the conditions required by each exploit and *OR* between the exploits implying each condition.

As the interpretation of the logic program, all the variables are Boolean. A true initial condition means the condition is satisfied and a false one means it has been disabled for hardening the network. A true exploit means it has been realized because all of its required conditions are satisfied. A true intermediate condition means it has been satisfied by at least one realized exploit implying the condition. With this logic program, the *network hardening* problem is simply to find the value assignments to the initial conditions such that a given set of goal conditions are all false. Those are more formally stated in Definition 3 and illustrated in Example 4.1.

Definition 3. Given an attack graph $G(E \cup C, R_r \cup R_i)$ with a given set of goal conditions $C_g \subseteq C$, let $P(G)$ denote a logic program comprised of the following clauses.

- $e \leftarrow c_1 \wedge c_2 \wedge \cdots c_n$, where $e \in E$ and each c_i is in $R_r(e)$
- $c \leftarrow e_1 \wedge e_2 \vee \cdots e_m$, where $c \in C$ and each e_i is in $R_i(c)$

The **network hardening** problem is to satisfy the goal $\neg c_1 \wedge \neg c_2 \wedge \cdots \neg c_l$ where each c_i is in C_g.

Example 4.1. For the attack graph G shown in Fig. 1, the following are examples of the clauses in $P(G)$

- $ftp\_rhosts(0,1) \leftarrow ftp(0,1) \wedge user(0)$
- $rsh(0,1) \leftarrow trust(1,0) \wedge user(0)$
- $user(1) \leftarrow rsh(0,1) \vee rsh(2,1) \vee sshd\_bof(0,1) \vee sshd\_bof(2,1)$
- $root(2) \leftarrow local\_bof(2)$

To harden the network, we need only to find a value assignment to the initial conditions such that the goal clause $\neg root(2)$ is satisfied.

4.2. A solution based on graph searching

By Definition 3, we can certainly depend on logic programming techniques to find a solution to the problem. However, considering the simplicity of the logic program, we shall instead resolve to a simpler solution based on graph searches. Roughly speaking, we start from the goal conditions to traverse the attack graph backwards by following the directed edges in the reverse direction. During the traversal we make logical inferences. At the end of the graph traversal, a logic proposition of the initial conditions is derived as the necessary and sufficient condition for satisfying the goal. Fig. 2 shows a procedure *Network_Hardening* that more precisely describes this process.

In Fig. 2, the first four lines of procedure *Network_Hardening* initialize the result L, a queue Q used for searching the attack graph, and a *predecessor set* $Pre(x)$ for each vertex x (an exploit or a condition) that includes all the vertices reachable from x. The procedure then searches the attack graph backwards as follows. For each condition c it leaves (that is, dequeued from Q), it substitutes c in the result L with a logically equivalent proposition, that is, the conjunction of those exploits that imply condition c (line 6 through line 10). It expands the search in a breadth-first manner (lines 11–12). It adds the vertices reachable from the current vertex to the predecessor set

```
Procedure Network_Hardening
Input: An attack graph $G(E \cup C, R_r \cup R_i)$ and the goal conditions $C_g \subseteq C$
Output: A solution $L$ to the goal $\bigwedge_{c \in C_g} \neg c$
Method:
    1. Let $L = \bigwedge_{c \in C_g} \neg c$
           //The initial goal
    2. Let $Q$ be a queue with the conditions in $C_g$ enqueued
           A queue used for searching in the graph
    3. For each $e \in E$ and $c \in C$
    4.     Let $Pre(e) = \{e\}$ and $Pre(c) = \{c\}$
               //The predecessor list
    5. While $Q$ is not empty, do
    6.         For each condition $c$ dequeued from $Q$, let $S_e = \{e_1, e_2, \ldots, e_n\}$ be the exploits pointing to $c$ in $G$
    7.             Let $T = (e_1 \vee e_2 \vee \ldots e_n)$
                   //Temporary variable
    8.             For each $e_i \in S_e \cap Pre(c)$
    9.                 Replace $e_i$ with $FALSE$ in $T$
    10.            Replace $c$ with $T$ in $L$
    11.            For each $e_i \in S_e - Pre(c)$
    12.                Enqueue $e_i$ in $Q$
    13.                Let $Pre(e_i) = Pre(e_i) \cup Pre(c)$
    14.        For each exploit $e$ dequeued from $Q$, let $S_c = \{c_1, c_2, \ldots, c_m\}$ be the conditions pointing to $e$ in $G$
    15.            Let $T = (c_1 \wedge c_2 \wedge \ldots c_n)$
    16.            For each $c_i \in S_c \cap Pre(e)$
    17.                Replace $c_i$ with $FALSE$ in $T$
    18.            Replace $e$ with $T$ in $L$
    19.            For each $c_i \in S_c - Pre(e)$
    20.                Enqueue $c_i$ in $Q$
    21.                Let $Pre(c_i) = Pre(c_i) \cup Pre(e)$
    21. Return $L$
```

Fig. 2. A procedure for deriving solutions to the network hardening problem.

of that vertex (line 13). The procedure avoids running into cycles by only expanding the search towards those vertices not reachable from the current vertex (line 11), and it also avoids introducing unsatisfiable logic loops into the final result (line 9). The procedure handles exploits in a similar way (line 14 through line 21).

To illustrate how the procedure *Network_Hardening* works, we revisit the attack graph and goal condition in Fig. 1. Because the procedure will only search among the vertices from which the goal condition is reachable, we can safely remove from further consideration the exploit *local_bof*(1) and the condition *root*(1), together with corresponding edges. The condition *user*(0), which denotes the attacker's privilege on his/her own machine, can also be removed because it is beyond the control of administrators. The simplified version of the attack graph is shown in Fig. 3.

We first consider how the search in the procedure *Network_Hardening* traverses this attack graph, and we shall show how the result L is updated shortly. Although the search will actually advance breadth-first, we shall describe it in a depth-first manner for clarity. The dotted lines in Fig. 3 illustrate how the procedure searches the attack graph. For clarity we have also divided the search into following steps, which correspond to the labels of the dotted lines in Fig. 3.

A: The search starts from the goal condition *root*(2) and advances to *user*(2).
B: It branches there and the first branch stops at *ftp*(0, 2).
C: The other further branches at *rsh*(1, 2), and the first branch reaches *user*(1).
D: The second branch also goes to *user*(1).
E: Further branches at *user*(1), and the first upward branch stops at *ftp*(0, 1).
F: The second upward branch stops at *sshd*(0, 1).
G: One downward branch goes to *sshd_bof*(2, 1) and then stops at *user*(2), because *user*(2) is in the predecessor set of *sshd_bof*(2, 1).
H: The other downward branch goes to *rsh*(2, 1) and further branches there.
I: The first branch stops at *user*(2), because *user*(2) is in the predecessor set of *rsh*(2, 1).
J: Similarly, the second branch also stops at *user*(2).

The result L is initially $\neg root(2)$ and is subsequently updated as in Fig. 4 (some straightforward steps and parentheses are omitted for simplicity). The condition *user*(1) actually appears twice in the proposition, required by *rsh*(1, 2) in line 3 and by *ftp_rhosts*(1, 2). The second appearance should be included in line 4 but we have omitted it for simplicity since $user(1) \wedge user(1)$ is logically equivalent to *user*(1). Notice, however, such simplification is not always possible (for example, in the case of $x \wedge y \vee x \wedge z$, both copies of x must be kept), and it is not part of the procedure. Indeed, the procedure differs from normal breadth-first search (BFS) because it may need to search through a

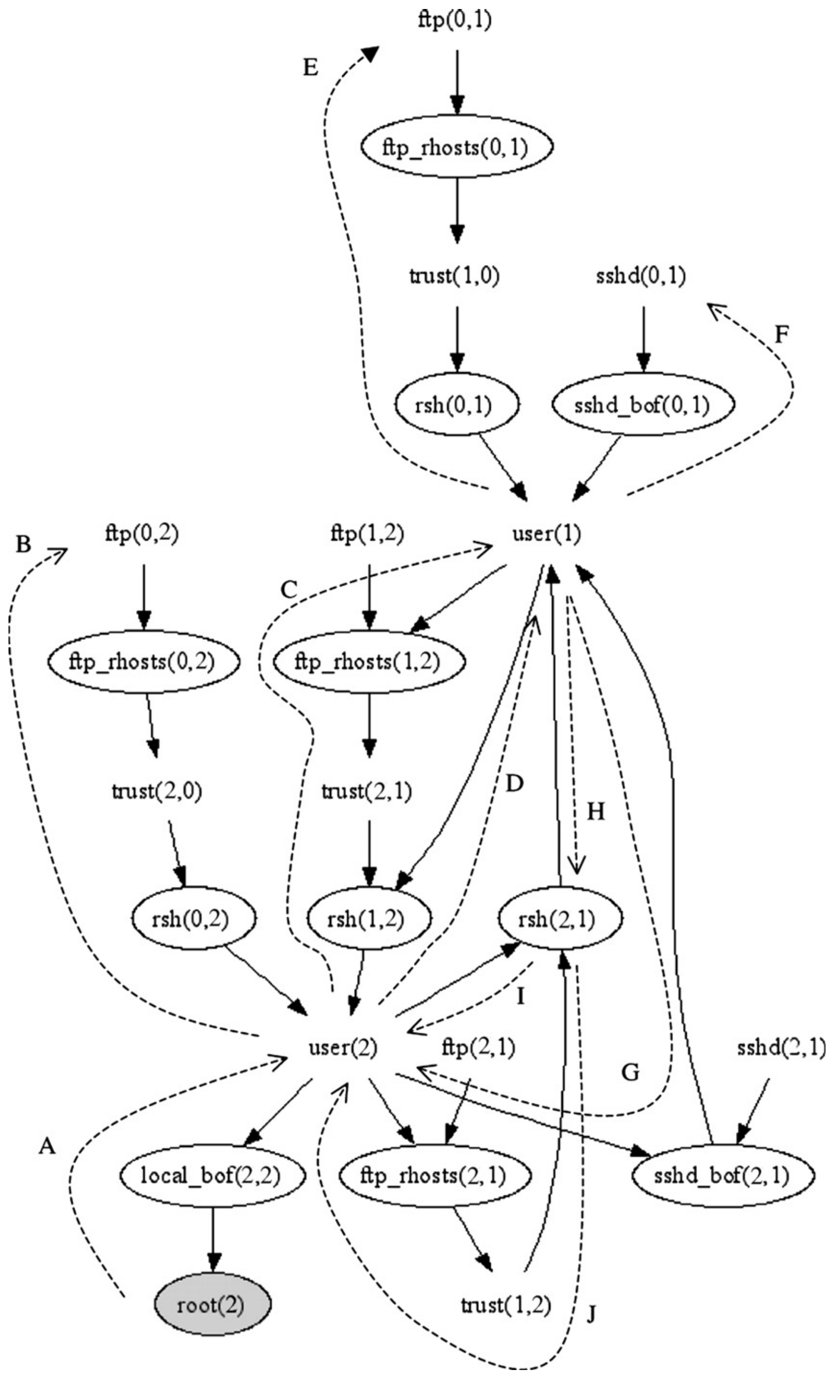

Fig. 3. Illustration of the procedure *Network_Hardening*.

1. $L = \neg root(2)$
2. $\quad = \neg(rsh(0,2) \vee rsh(1,2))$
3. $\quad = \neg(ftp\_rhosts(0,2) \vee trust(2,1) \wedge user(1))$
4. $\quad = \neg(ftp(0,2) \vee ftp(1,2) \wedge (rsh(0,1) \vee sshd\_bof(0,1) \vee rsh(2,1) \vee sshd\_bof(2,1)))$
5. $\quad = \neg(ftp(0,2) \vee ftp(1,2) \wedge (ftp(0,1) \vee sshd(0,1) \vee trust(1,2) \wedge FALSE \vee sshd(2,1) \wedge FALSE))$
6. $\quad = \neg(ftp(0,2) \vee ftp(1,2) \wedge (ftp(0,1) \vee sshd(0,1) \vee ftp(2,1) \wedge FALSE \wedge FALSE \vee sshd(2,1) \wedge FALSE))$
7. $\quad = \neg(ftp(0,2) \vee ftp(1,2) \wedge (ftp(0,1) \vee sshd(0,1)))$

Fig. 4. An example of applying the procedure *Network_Hardening*.

vertex multiple times (for example, x in the case of $x \wedge y \vee x \wedge z$) whereas a BFS visits each vertex exactly once.

In Fig. 4, the *FALSE* values are results of the two cycles in the attack graph (from *user*(1) to *user*(2), through *sshd_bof*(2,1) and through *rsh*(2,1), respectively). For example, when the search leaves *rsh*(2,1) and reaches *user*(2), it finds that *user*(2) is in the predecessor set of *rsh*(2,1). Hence, instead of replacing *rsh*(2,1) with $user(2) \wedge trust(2,1)$, line 16 and 17 in Fig. 2 replace *rsh*(2,1) with $trust(1,2) \wedge FALSE$. Similar argument explains the other FALSE values in line 5 and line 6. Although we remove the effect of those *FALSE* values in line 7 to simplify the result, note that this is not part of the procedure.

Proposition 1. *The procedure* Network_Hardening *shown in Fig. 4 returns the necessary and sufficient condition of the given goal condition.*

Proof (Sketch). The correctness of the procedure can be proved by induction on the number of involved exploits. When no exploit is involved, the goal condition must be an initial condition itself, for which the result trivially holds. For the inductive case, suppose the result holds when n exploits are involved. For the case of $n+1$, considering that the procedure advances in a breadth-first manner, the partial result must hold after n exploits have been processed and the conditions they require have been enqueued (lines 14–21). There must be at least one such condition that is implied by the $(n+1)$th exploit. We only need to show the procedure correctly expands such a condition into its necessary and sufficient condition through the $(n+1)$th exploit.

After the condition is dequeued (line 6), it is replaced by the disjunction of all the exploits that may imply it (including the $(n+1)$th exploit), which is the necessary and sufficient condition for the condition to be satisfied (lines 7 and 10). However, if an exploit is in the predecessor list of the condition, then realizing the exploit requires the condition to be satisfied in the first place, and hence the exploit should be regarded as unrealizable (an exploit is unrealizable if any of its required conditions is so) and be replaced by *FALSE* (line 9). Next each of the exploits not in the predecessor list of the condition is enqueued (line 12) and dequeued (line 14) for processing.

Each dequeued exploit is then replaced by its necessary and sufficient condition, that is the conjunction of all the conditions they require (lines 15 and 18). However, if a condition is in the predecessor list of the exploit, then the exploit can imply the condition only if the condition is satisfied in the first place. This logic loop implies that this occurrence of the condition in the result should be replaced by *FALSE*. Notice that only this specific occurrence of the condition is set as *FALSE*, but other occurrences of the same condition in the result are not affected. This is because the condition may still be implied by other exploits and hence may still be satisfiable. From the above discussion, the procedure never runs into a loop, and it always terminates with the correct result.

The way we handle cycles in attack graphs is different from that in the preliminary version of this paper, where cycles are removed through an extra forward search [9]. For example, in Fig. 3, the forward search will detect a cycle when it reaches *user*(1) for the second time from *rsh*(2,1) (through *user*(2) and *user*(1)). The cycle will be removed by deleting the last edge pointing from *rsh*(2,1) to *user*(1). We do not require such an extra step but avoid cycles in the backward search itself. This not only simplifies the process, but also enables our method to handle the following case. If a cycle is detected when the search reaches an exploit instead of a condition for the second time, then deleting the last edge will cause an incorrect conclusion. In Fig. 5, deleting the edge from c_4 to e_1 will yield the erroneous result that e_1 only requires c_1 and c_2 (our solution will yield the correct result $L = \neg(c_1 \wedge c_2 \wedge FALSE) = TRUE$, that is, the goal condition is already safe).

4.3. Choosing minimum-cost solutions

By Definition 3, a logic proposition L returned by the *Network_Hardening* is the necessary and sufficient condition for hardening the network such that none of the goal conditions can be satisfied. However, such a proposition usually implies multiple non-comparable options. Moreover, such options are not always so apparent from the proposition itself. Therefore, we go even further to simplify the proposition and choose optimal solutions with respect to given cost metrics.

We first convert the proposition L returned by the Procedure *Network_Hardening* to its disjunctive normal form (DNF). Each disjunction in the DNF thus represents a particular sufficient option in hardening the network. Because each disjunction in the DNS is the conjunction of negated initial conditions, those initial conditions must be disabled. This is enforceable, because all the initial conditions are independent of others and can be readily disabled. For example, from the last line in Fig. 4 we have that $L = \neg(ftp(0,2) \vee ftp(1,2) \wedge (ftp(0,1) \vee sshd(0,1)))$. We can convert L to its DNF as follows. First, $L \equiv \neg((ftp(0,2) \vee ftp(1,2)) \wedge (ftp(0,2) \vee ftp(0,1) \vee sshd(0,1)))$

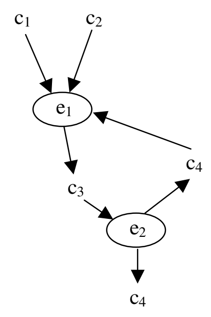

Fig. 5. An example of attack graph with cycle.

holds by the tautology $A \vee B \wedge C \leftrightarrow (A \vee B) \wedge (A \vee C)$ [7]. Then by applying De Morgan's law, we have $L \equiv \neg ftp(0,2) \wedge \neg ftp(1,2) \vee \neg ftp(0,2) \wedge \neg ftp(0,1) \wedge \neg sshd(0,1)$. From this DNF, we clearly know the two options in hardening the network: one is to disable both $ftp(0,2)$ and $ftp(1,2)$, the other is to disable the three conditions $ftp(0,2)$, $ftp(0,1)$, and $sshd(0,1)$.

However, although any of the disjunctions in the DNF of L is a sufficient option for hardening the network, the cost of those options may be different. First, the set of initial conditions involved in one option may be a proper super set of those involved in another option. The cost incurred by the latter option is clearly no greater than that by the former, and hence the former option can be removed from further consideration. Fig. 6 shows an example of attack graph with two initial conditions. The Procedure *Network_Hardening* will return $L = \neg((c_1 \vee c_2) \wedge c_1 \wedge c_2)$, and the DNF is $L \equiv \neg c_1 \wedge \neg c_2 \vee \neg c_1 \vee \neg c_2$. Clearly, among the three options $\neg c_1 \wedge \neg c_2$, $\neg c_1$, and $\neg c_2$, the first incurs no less cost than the second or the third and hence should be removed from consideration.

The above example also shows that theoretically the DNF of L may have an exponential size in the number of initial conditions (after the above reduction, this number of options will be bound by the number of incomparable subsets of n initial conditions, which is known as the binomial coefficient $\binom{n}{\lfloor n/2 \rfloor}$ by Sperner's Theorem). This implies that the Procedure *Network_Hardening* has an unavoidable exponential worst-case complexity, because its result is exponential. Indeed, the procedure may visit a vertex many times (bound by the in-degree of the vertex). However, based on our experiences we believe in practice the running time is usually acceptable. This is because the attack graph of a well-protected network is usually small and sparse (the in-degree of each vertex is small) since most easy-to-remove vulnerabilities have already been disabled through existing security measures such as firewalls. Attack graph analysis is less useful for unsecured networks, in which the analysis conclusion is simply "harden everything." It is more useful in situations where some basic network hardening has already been done, and careful analysis is needed for evaluating risk versus hardening cost in the context of multistep attacks.

After the above reduction, the options left will only involve pairwise incomparable subsets of initial conditions. The options that incur the minimum cost can be easily chosen, if the cost of disabling each initial condition has been assigned by administrators. In such a case, the cost of an option is simply equal to the summation of the cost of all the initial conditions involved by the option. Although it is usually difficult to assign precise cost to each condition, the conditions can always be partially ordered based on their costs. Consequently, the options can also be partially ordered based on the cost of conditions. An option with a cost no greater than any other options can thus be chosen based on the partial order.

For example, consider the two options we have derived, that is either to disable both $ftp(0,2)$ and $ftp(1,2)$, or to disable the three conditions $ftp(0,2)$, $ftp(0,1)$, and $sshd(0,1)$. The condition $ftp(0,2)$ must be disabled in either case, and hence it can be ignored in considering relative costs. Since the condition $sshd(0,1)$ can be disabled by patching the buffer overflow vulnerability in the sshd service, the cost may be relatively low. On the other hand, the conditions involving the ftp service incurs more costs, because the ftp service is properly functioning, and is simply used by the attacker in a clever way. Moreover, disabling $ftp(0,2)$ may mean stopping the ftp service on host 2 to all external hosts, which may incur a higher cost than stopping the ftp service between two internal hosts 1 and 2 (they may still communicate files via other services). Based on those assumptions, the first option has a lower cost than that of the second and thus should be chosen as the solution.

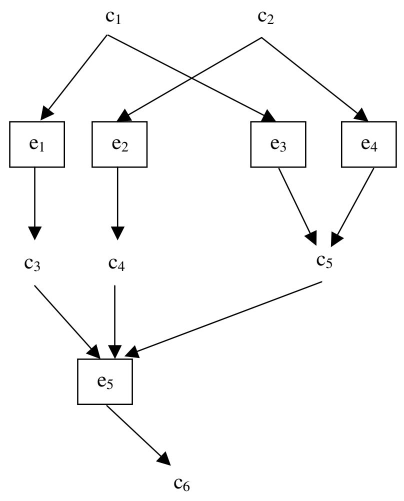

Fig. 6. An example of attack graph with exponential number of hardening options.

5. A case study

In this section, we study a relatively more realistic example with enhanced security measures as well as attacks at multiple network layers. The example discussed in previous sections has focused on the Procedure *Network_Hardening*, whereas this second example will better justify our techniques in choosing the minimum-cost network hardening option as discussed in Sections 4.2 and 4.3. Moreover, this example more clearly reveals the limitations of existing approaches, such as the minimum critical set approach [15,6] and the set of all reachable exploits [1].

Fig. 7 shows the network configuration for our second example. The link-layer connectivity between the three hosts is provided by an Ethernet switch. At the transport layer, security has been enhanced by removing unused services, replacing FTP and telnet with secure shell, and adding tcpwrapper protection on RPC services.

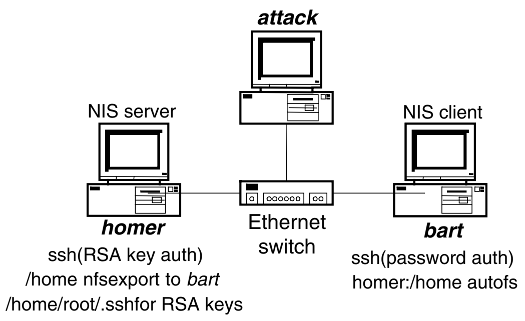

Fig. 7. Network configuration for second example.

Application-layer trust relationships further restrict NFS and NIS domain access. In spite of those security measures, exploits still exist and are listed in Table 1, whereas security conditions are listed in Table 2. Fig. 8 then shows the attack graph for this example. The lowercase Greek letters are used as short names for the corresponding initial conditions. Notice that we allow some of the conditions to have multiple copies to keep the attack graph legible. Also, the conditions are used as labels of edges between exploits for simplicity.

The Procedure *Network_Hardening* searches the attack graph in a way similar to the previous example. The search starts from the goal conditions and branches at the exploit *ssh_login_pk_su*(*bart*, *homer*). The right branch ends at the condition η. The middle branch reaches the exploit *ssh_ login_pw*(*attack*, *bart*) and consequently ends at the conditions α, β, χ. The left branch further branches at the exploit *create _nfs_home_ssh_pk_su*(*bart*, *homer*). The middle two branches reach the conditions ϕ and γ. The right branch reaches *ssh_login_pw*(*attack*, *bart*) for the second time and advances as usual. The left branch reaches the condition *pgm_glibc_bof*(*bart*) and further branches there. The right branch reaches conditions δ and ε, and also the exploit *ssh_login_pw*(*attack*, *bart*) for the third time. The left branch reaches α, β, and χ via the exploit *scp_upload_pw*(*attack*, *bart*).

The result returned by the procedure is thus $T = \neg(\eta \wedge (\phi \wedge \gamma) \wedge (\alpha \wedge \beta \wedge \chi) \wedge ((\alpha \wedge \beta \wedge \chi) \vee (\alpha \wedge \beta \wedge \chi \wedge \delta \wedge \varepsilon)))$ (some parentheses are omitted for simplicity). This seemingly complex result, however, has a simple DNF $\neg\alpha \vee \neg\beta \vee \neg\chi \vee \neg\phi \vee \neg\gamma \vee \neg\eta$. The two initial conditions δ and ε do not appear in the DNF. They have dropped out in this fashion: $(\alpha \wedge \beta \wedge \chi) \vee (\alpha \wedge \beta \wedge \chi \wedge \delta \wedge \varepsilon) \equiv \alpha \wedge \beta \wedge \chi$ due to the tautology $A \vee (A \wedge B) \leftrightarrow A$. Intuitively, the condition *pgm_glibc_bof*(*bart*) can be implied by any of the two exploits, *scp_upload_pw*(*attack*, *bart*) and *scp_download_pw*(*bart*, *attack*). The latter requires two more conditions, δ and ε, than the former does. Therefore, disabling δ and ε does not help at all.

It is worth noting that the above important observation would be difficult if at all possible with the exploit-based approaches [15,6,1]. First, the goal conditions are reach-

Table 1
Exploits for the Second Example

Exploit	Description
arp_spoof	Spoof (impersonate) machine identity via ARP poison attack
ypcat_passwd	Dump encrypted NIS password file
crack_yp_passwd	Crack encrypted user password(s)
scp_upload_pw	Secure shell copy, upload direction, using password authentication
scp_download_pw	Secure shell copy, download direction, using password authentication
ssh_login_pw	Secure shell login using password authentication
rh62_glibc_bof	Red Hat 6.2 buffer overflow in glibc library
create_nfs_home_ssh_pk_su	Exploit NFS home share to create secure shell key pair used for superuser authentication
ssh_login_pk_su	Secure shell login using public key authentication

Table 2
Security Conditions for the Second Example

Condition	Description
link_arp	Attacker shares link-level connectivity with victim (both on same LAN)
trans_yp	Transport layer connectivity to NIS server
trans_ssh_pw	Transport layer connectivity to secure shell server that supports password authentication
trans_ssh_pk	Transport layer connectivity to secure shell server that supports public key authentication
trans_nfs	Transport layer connectivity to NFS server
app_nfs_home_su	Application “connection” representing sharing superuser’s home directory
app_yp_domain	Application “connection” representing NIS domain membership
app_yp_passwd	Application “connection” representing acquisition of encrypted NIS password database
app_pwauth	Application “connection” representing acquisition of unencrypted user password
app_ssh_pk_su	Application “connection” representing acquisition/creation of key pair for superuser authentication
pgm_glibc_bof	Program used to exploit glibc library buffer overflow vulnerability
execute	Execute access obtained
superuser	Superuser privilege obtained

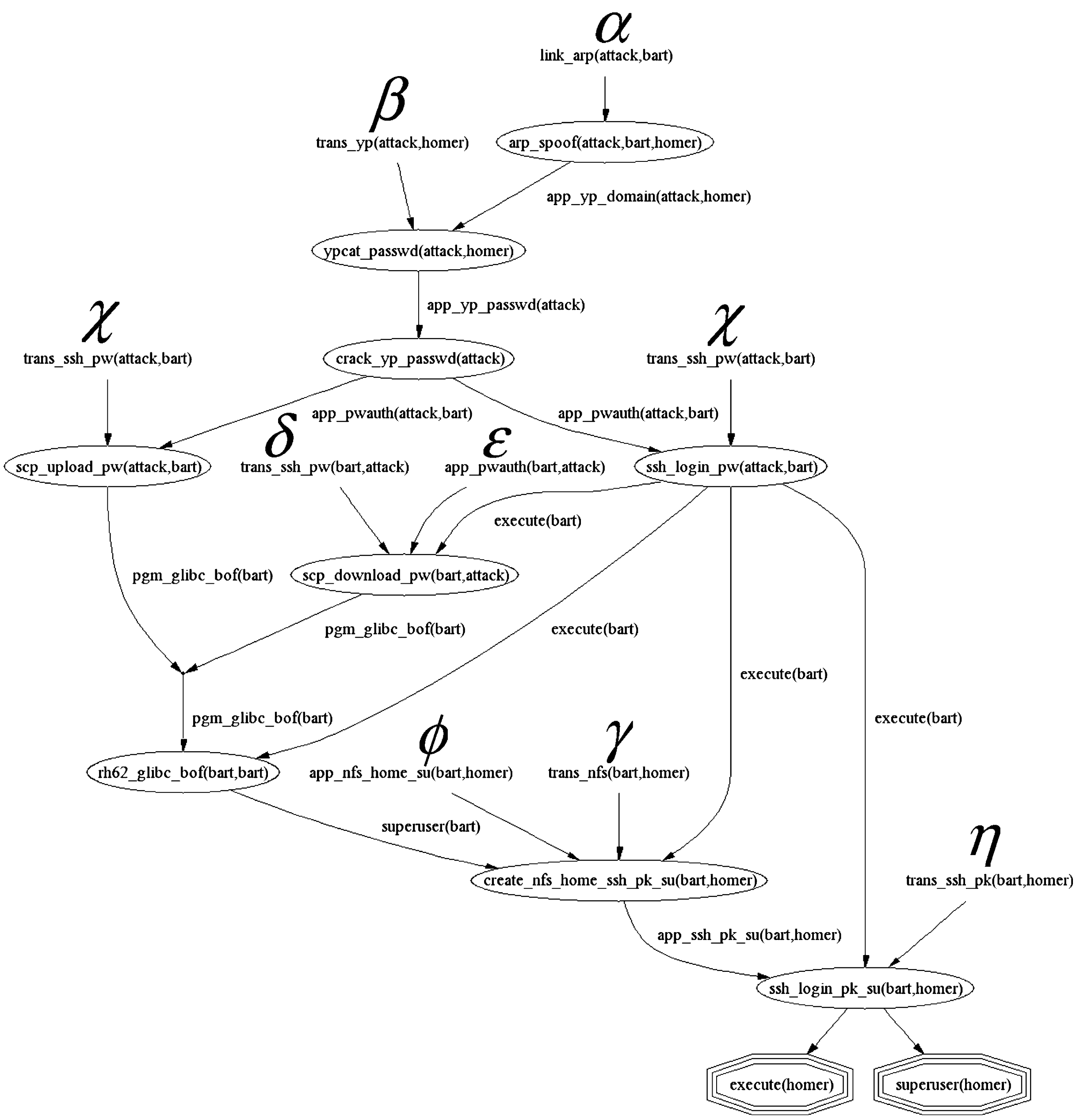
α
link_arp(attack,bart)
β
trans_yp(attack,homer)
arp_spoof(attack,bart,homer)
app_yp_domain(attack,homer)
ypcat_passwd(attack,homer)
app_yp_passwd(attack)
χ
trans_ssh_pw(attack,bart)
crack_yp_passwd(attack)
χ
trans_ssh_pw(attack,bart)
app_pwauth(attack,bart)
app_pwauth(attack,bart)
δ
ε
scp_upload_pw(attack,bart)
trans_ssh_pw(bart,attack)
app_pwauth(bart,attack)
ssh_login_pw(attack,bart)
execute(bart)
pgm_glibc_bof(bart)
scp_download_pw(bart,attack)
pgm_glibc_bof(bart)
execute(bart)
execute(bart)
execute(bart)
pgm_glibc_bof(bart)
φ
γ
rh62_glibc_bof(bart,bart)
app_nfs_home_su(bart,homer)
trans_nfs(bart,homer)
superuser(bart)
η
create_nfs_home_ssh_pk_su(bart,homer)
trans_ssh_pk(bart,homer)
app_ssh_pk_su(bart,homer)
ssh_login_pk_su(bart,homer)
execute(homer)
superuser(homer)

Fig. 8. Attack graph for the second example.

able from both δ and ε, and hence a reachability-based algorithm will not detect their irrelevancy to network hardening. If such conditions are not initial conditions but the ends of another chain of exploits, then whatever initial conditions the chain lead to will be erroneously disabled, causing unnecessary loss of availability. Second, any singleton set of exploits except *scp_upload_pw*(*attack*, *bart*) and *scp_download_pw*(*bart*, *attack*) comprises a minimum critical set in this case, and the set {*scp_upload_pw*(*attack*, *bart*), *scp_download_pw*(*bart*, *attack*)} is a minimal critical set. However, such a result is not as clear as our result, considering the fact that some of the exploits cannot be directly removed, such as *crack_yp_passwd*(*attack*) and *rh*62_*glibc_bof*(*bart*, *bart*).

The six network hardening options we now have are:

(1) $\neg$*link_arp*(*attack*, *bart*), or
(2) $\neg$*trans_yp*(*attack*, *homer*), or
(3) $\neg$*trans_ssh_pw*(*attack*, *bart*), or
(4) $\neg$*app_nfs_home_su*(*bart*, *homer*), or
(5) $\neg$*trans_nfs*(*bart*, *homer*), or
(6) $\neg$*trans_ssh_pk*(*bart*, *homer*)

Those options can be partially ordered with following assumptions about the cost of each initial condition. For the first option, *link_arp* refers to mapping IP addresses to MAC addresses through ARP (address resolution protocol). Therefore, the first option can be enforced by hard-coding

IP/MAC address and switching port relationships throughout the network, which clearly incurs management overhead costs. For the third option, the condition *trans_ssh_pw*(*attack*, *bart*) can be disabled by modifying the sshd configuration of the host *bart* to use public-key authentication only (disabling password authentication). This enhances security and is thus a low-cost option (which is actually already in place for the host *homer*).

For the fourth option, *app_nfs_home_su*(*bart*, *homer*) is an application-layer association that represents the sharing of the superuser home directory. Although such configuration may ease administration, it is indeed a poor practice from a security viewpoint (it allows the overwriting of a secure shell authentication key pair in this particular case). Therefore, the fourth option incurs some administration cost and can be enforced by removing the directory share. On the other hand, the remaining three options (the second, fifth, and sixth option) would make critical network services unavailable and thus incur high costs in terms of availability. From the above discussion, we conclude with a partial order of the six options based on their relative costs: $3 < 1, 4 < 2, 5, 6$. That is, the third option is the minimum-cost solution to network hardening.

6. Conclusion

In this paper, we have proposed methods to compute network hardening options for protecting given critical resources. We have also discussed how to choose hardening options with the minimum costs. Unlike previous approaches, the network hardening solutions we provide are in terms of adjustable network configuration elements rather than exploits. Such solutions take into account the often complex relationships among exploits and configuration elements. In this way, our solution is readily enforceable, and it keeps the cost of network hardening minimal by leaving out those seemingly irrelevant exploits whose removal actually have no effect on protecting the critical resources. Unlike the previous version of our method, the new algorithm we have proposed can derive solutions with one-pass of search in the attack graph while avoiding logic loops. This not only saves an unnecessary preprocessing step, but also addresses the difficulty in handling cycles that cannot be easily removed by a forward search. The current algorithm builds the logic proposition then simplifies it. In our future work, we shall pursue a solution that integrates the two steps into one single algorithm such that redundant clauses in the proposition can be avoided.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments. This material is based upon work supported by HSARPA under the contract FA8750-05-C-0212 administered by the Air Force Research Laboratory, by Army Research Office under grants DAAD19-03-1-0257 and W911NF-05-1-0374, by Federal Aviation Administration under the contract DTFAWA-04-P-00278/0001, and by the National Science Foundation under grants IIS-0242237 and IIS-0430402. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] P. Ammann, D. Wijesekera, S. Kaushik, Scalable, graph-based network vulnerability analysis. In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02), pp. 217–224, 2002.

[2] M. Dacier, Towards quantitative evaluation of computer security. Ph.D. Thesis, Institut National Polytechnique de Toulouse, 1994.

[3] R. Deraison, Nessus scanner, 1999. Available at <http://www.nessus.org./>.

[4] D. Farmer, E.H. Spafford, The COPS security checker system. In USENIX Summer, pp. 165–170, 1990.

[5] S. Jajodia, S. Noel, B. O'Berry, Topological analysis of network attack vulnerability, in: V. Kumar, J. Srivastava, A. Lazarevic (Eds.), Managing Cyber Threats: Issues, Approaches and Challenges, Kluwer Academic Publisher, 2003.

[6] S. Jha, O. Sheyner, J.M. Wing, Two formal analysis of attack graph. In Proceedings of the 15th Computer Security Foundation Workshop (CSFW'02), 2002.

[7] E. Mendelson, Introduction to Mathematical Logic, fourth ed., Chapman & Hall, 1997.

[8] S. Noel, S. Jajodia, Correlating intrusion events and building attack scenarios through attack graph distance. In Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), 2004.

[9] S. Noel, S. Jajodia, B. O'Berry, M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03), 2003.

[10] R. Ortalo, Y. Deswarte, M. Kaaniche, Experimenting with quantitative evaluation tools for monitoring operational security, IEEE Trans. Software Eng. 25 (5) (1999) 633–650.

[11] C. Phillips, L. Swiler, A graph-based system for network-vulnerability analysis. In Proceedings of the New Security Paradigms Workshop (NSPW'98), 1998.

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

[13] R. Ritchey, P. Ammann, Using model checking to analyze network vulnerabilities. In Proceedings of the 2000 IEEE Symposium on Research on Security and Privacy (S&P'00), pp. 156–165, 2000.

[14] R. Ritchey, B. O'Berry, S. Noel. Representing TCP/IP connectivity for topological analysis of network security. In Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02), p. 25, 2002.

[15] O. Sheyner, J. Haines, S. Jha, R. Lippmann, J.M. Wing, Automated generation and analysis of attack graphs. In Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02), pp. 273–284, 2002.

[16] L. Swiler, C. Phillips, D. Ellis, S. Chakerian. Computer attack graph generation tool. In Proceedings of the DARPA Information Survivability Conference & Exposition II (DISCEX'01), 2001.

[17] L. Wang, A. Liu, S. Jajodia. An efficient and unified approach to correlating, hypothesizing, and predicting intrusion alerts. In Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005), pp. 247–266, 2005.

[18] D. Zerkle, K. Levitt, Netkuang – a multi-host configuration vulnerability checker. In Proceedings of the 6th USENIX Unix Security Symposium (USENIX'96), 1996.

Lingyu Wang is an assistant professor in the Concordia Institute for Information Systems Engineering at Concordia University, Montreal, Quebec, Canada. He received his Ph.D. degree in Information Technology from George Mason University. His research interests include database security, data privacy, vulnerability analysis and intrusion detection. He holds a M.E. from Shanghai Jiao Tong University and a B.E. from Shen Yang Institute of Aeronautic Engineering in China.

Steven Noel is Associate Director and Senior Research Scientist at the George Mason University Center for Secure Information Systems. His research interests include cyber attack modeling, intrusion detection data mining, and visualization for information security. He received his Ph.D. in Computer Science from the University of Louisiana at Lafayette in 2000. His dissertation work was in data mining and visualization for information retrieval. He also earned an M.S. in Computer Science from the University of Louisiana at Lafayette (1998) and a B.S. in Electro-Optics from the University of Houston–Clear Lake (1989). From 1990 to 1998, Dr. Noel was a research scientist at the Naval Surface Warfare Center in Dahlgren, Virginia, where he worked in image/video compression, wavelets, neural networks, genetic algorithms, radar signal processing, missile guidance, and astronomy. He has published numerous conference papers, journal articles, and technical reports. He is a member of IEEE, SPIE, and INNS, as well as the Alpha Chi, Phi Kappa Phi, and Upsilon Pi Epsilon honor societies.

Sushil Jajodia is BDM International Professor of Information Technology and the director of Center for Secure Information Systems at the George Mason University, Fairfax, Virginia. His research interests include information security, temporal databases, and replicated databases. He has authored five books, edited twenty-four books, and published more than 300 technical papers in the refereed journals and conference proceedings. He is the founding editor-in-chief of the Journal of Computer Security and on the editorial boards of ACM Transactions on Information and Systems Security, IEE Proceedings on Information Security, International Journal of Cooperative Information Systems, and International Journal of Information and Computer Security.