A Host–Based Approach to Network Attack Chaining Analysis

Paul Ammann
ISE Department
George Mason University
pammann@gmu.edu

Joseph Pamula
Center for Secure Information Systems
George Mason University
jpamula@gmu.edu

Ronald Ritchey
Booz Allen & Hamilton
ritchey_ronald@bah.com

Julie Street
ISE Department
George Mason University
jstreet1@gmu.edu

Abstract

The typical means by which an attacker breaks into a network is through a chain of exploits, where each exploit in the chain lays the groundwork for subsequent exploits. Such a chain is called an attack path, and the set of all possible attack paths form an attack graph. Researchers have proposed a variety of methods to generate attack graphs. In this paper, we provide a novel alternative approach to network vulnerability analysis by utilizing a penetration tester's perspective of maximal level of penetration possible on a host. Our approach has the following benefits: it provides a more intuitive model in which an analyst can work, and its algorithmic complexity is polynomial in the size of the network, and so has the potential of scaling well to practical networks. The drawback is that we track only "good" attack paths, as opposed to all possible attack paths. Hence, an analyst may make suboptimal choices when repairing the network. Since attack graphs grow exponentially with the size of the network, we argue that suboptimal solutions are an unavoidable cost of scalability, and hence practical utility. A working prototype tool has been implemented to demonstrate the practicality of our approach.

1. Introduction

Even well–administered networks are vulnerable to some level of attack, since eliminating all susceptibility to attack is arguably equivalent to isolating the network, which is not an option in most enterprises. One of the challenging tasks that a security administrator faces is to analyze his or her organization's networks for susceptibility to attack and, as necessary, to modify the network so that it becomes sufficiently secure. Penetration testing is a typical mechanism for carrying out this analysis, but penetration testing is expensive, labor intensive, and often incomplete. Consequently, there is considerable interest in automating aspects of penetration testing. This paper offers a novel approach to automating part of the penetration tester's job.

Penetration testers routinely use attack graphs to help them understand a network's weaknesses. Roughly speaking, attack graph nodes represent network states, and attack graph edges represent the application of an exploit that transforms one network state into another, more compromised, network state. The terminal state of the attack graph represents a network state in which the attacker has achieved his or her goal. Typically these exploits take advantage of known vulnerabilities in various systems and services. Such vulnerabilities and their associated exploits are well documented in public sites such as `bugtraq` [3], and many commercial tools such as `Nessus` [11] and `Retina` [18] can identify known vulnerabilities in a given host or network.

At the research level, methods have been proposed to construct attack graphs based on data provided by commercial vulnerability scanning tools. Details of these methods, and their relation to the current work, are given in our "Related Work" section. For the purposes of the current paper, the salient observation is that attack graphs quickly become unmanageably large as network complexity grows past a few machines. Indeed, displays from tools built to support attack graph generation often show an extremely dense web of connections that provides little guidance to the analyst.

From a theoretical point of view, attack graphs become unmanageable because they are, by nature, exponential in the size of the network. To see why, consider a typical attack path: it represents a *minimal* set of exploits [1]; that is,

[1] A similar analysis follows from the equivalent formulation of an attack path as a minimal set of "facts", where each fact is some network condition

each exploit is necessary for that attack path as a whole to achieve the desired goal. There are potentially many such minimal sets of exploits. Given e exploits, there are 2^e possible subsets of exploits, and the number of minimal subsets is still $O(2^e)$. Unfortunately, as network size grows, so does the number of exploits, since each exploit is bound to a particular host. To see the reason for this, consider applying a buffer overflow attack against the `ssh` service. Attacking host $\mathcal{B}$ from host $\mathcal{A}$ with this exploit results in a different network state than attacking host $\mathcal{C}$ from host $\mathcal{A}$ with the same type of exploit, and so we must consider each attack separately. The bottom line is that adding another host inevitably increases the number of exploits e that must be considered. Hence *complete* attack graphs, by their very nature, do not scale well.

One way out of this computational quagmire is to go back to the penetration tester's perspective and ask what structure short of the entire attack graph would nonetheless be useful. The answer we propose in this paper is that penetration testers often think in terms of the maximal level of penetration possible with respect to a given host, and push the details of how to achieve this level to the background.

The contribution of this paper is an alternative to complete attack graphs. Our approach has the following benefits:

- The approach meshes well with the typical mental model of both the system administrator and the penetration tester, both of whom, we argue, find it natural to analyze network security in terms of the maximal compromise possible on each host.
- The approach can scale better than complete attack graphs to realistic size networks. This is because our host–centric model grows polynomially with the number of hosts, but the complete attack graph model grows exponentially.
- The approach through multiple *iterations* [2] will provide a system administrator with a secure network the same as a complete attack graph would, at the expense that the choices made may be suboptimal. Since each iteration is done in polynomial time, it can be computed in real–time and will give the system administrator an area that needs to be fixed.
- The approach can be used to provide near real–time early warning of potential attacks, to identify the network policy rules violations, and to conduct analysis on the potential impacts of giving different permissions or credentials to users (i.e., modelling insider attacks).

(e.g., `ssh` service is running on host $\mathcal{A}$) (initially) necessary for a given chain of exploits to be feasible.

[2]The term "iterate" refers to a one complete execution cycle of the algorithm.

The downside of our approach is that the analyst is not presented with complete information about possible damage, but instead is presented with worst case possible damage in each iteration. Consequently, the analyst may make suboptimal choices in terms of which vulnerabilities to repair. When complete attack graphs are used the analyst can determine the minimal changes required to fix the network at the expense of computing the large exponential graph. In using our approach, with every iteration run, analysts are quickly presented with an area that needs to be fixed at the expense that the choices made maybe suboptimal. This means that the choices made will provide the same secure network as attack graphs, however they may not be the minimal number of changes. Our position is that since the optimal choice of vulnerability repair is NP–complete anyway, suboptimal solutions are a foregone conclusion for realistic networks.

In our model, hosts have an ordered set of possible levels of access, ranging from `none` to `admin`. For example, we might record that from host $\mathcal{A}$, it is possible to obtain `admin` level of access on host $\mathcal{B}$ with a buffer overflow exploit of the `ssh` service running on host $\mathcal{B}$. At this point in the analysis, we would not be concerned with other possible attacks on $\mathcal{B}$ that result in `admin` (or lesser) level of access. If the `ssh` exploit from host $\mathcal{A}$ to host $\mathcal{B}$ becomes no longer beneficial, either because connectivity between hosts $\mathcal{A}$ and $\mathcal{B}$ was interrupted (perhaps by a firewall), or because the `ssh` service was removed from host $\mathcal{B}$, then the analysis of other exploits would begin again, with the goal of choosing one that leads to the (new) maximal level of penetration of host $\mathcal{B}$.

Our model also handles a transitive aspect of chained exploits. Suppose that there exists a `user` level access from host $\mathcal{A}$ to host $\mathcal{B}$, but there is `admin` level access from host $\mathcal{B}$ to host $\mathcal{C}$. From the penetration testing perspective, this is equivalent for $\mathcal{A}$ having `admin` level access to $\mathcal{C}$. Our algorithm presented in Section 2 will use this property to compute a transitive closure of our access graph.

To demonstrate the practicality of our approach, a working prototype tool has been implemented. The tool comprises all the algorithms presented in this paper. We have also put together a companion website for this paper which describes our tool's usage by using the example presented in the paper. The website address is: `http://ite.gmu.edu/~jpamula/tools/host-based.html`. It walks the reader through the example in Section 4 by showing the tool's screenshots at different stages of access graph construction and analysis.

The remainder of this paper is organized as follows: Section 2 presents the description of the algorithms, and also discusses the computational complexity of our host–based model approach to chained network attacks. In Section 3 we suggest a number of ways the analyst can use the results

of a stable host–based access graph (from Section 2) to help secure the network. In Section 4 we give an example that illustrates the ideas presented in this paper. Finally, Section 5 presents related work and Section 6 concludes this paper.

2. Model

We make an explicit assumption of *monotonicity* in our model. One way of thinking about monotonicity is that it means the attacker never has to backtrack. Although there are certain 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 the modeling the "port forward" exploit monotonically.

Our model is organized around *hosts*, rather than exploits or vulnerabilities. It constructs an *access graph* with a *node* for each host in the network. A *directed edge* from host h_1 to host h_2 in the access graph represents the access available on h_2 from h_1. Initial access is achieved directly between two hosts by a trust relationship based on network rules and configuration. Once complete, this initial set of edges represents the intended network access. Note that there might be multiple ways in which two hosts in the network can communicate. Instead of adding multiple edges in the graph, our model will only retain the *highest* access that can be achieved between hosts, since higher levels of access to the destination host typically mean that more powerful attacks can be accomplished. For example, a user with `user` access to a host cannot, in general, mount as powerful an attack on another host as can a user with `admin` access.

After the initial set of edges is complete, exploits are then introduced into the model. A host can improve its access to another host by using an exploit on the destination host. For example, h_1 may have a trust relationship with h_2 that allows `user` level access on h_2. However it may also be possible for h_1 to use a buffer overflow exploit on h_2 to gain `admin` level access. In this case the edge between h_1 and h_2 would be updated to reflect this new access. Hosts can also gain higher access indirectly through a series of exploits on numerous hosts. For example, h_1 may not be able to connect to h_3 directly. However, h_2 may have `admin` access on h_3 by using a remote root level exploit. Using this exploit, and the buffer overflow exploit on h_2, h_1 can successfully gain `admin` level access on h_3.

The model to construct such an access graph consists of two steps: *initialization* and calculation of *maximal access*. The goal of initialization is to establish the initial trust relationships between hosts in absence of applying any exploits. Initialization is accomplished using the algorithm `findInitialAccess`, shown in Figure 1 and described next. Here `access()` returns the possible access level gained between two hosts in trust relationship table T, or a current level of access for a given edge e_{ij} between two hosts i and j.

```
findInitialAccess(H, T, E) :                                          1
Input:  A set of host nodes, H                                        2
Input:  A set of trust relationships, T                               3
Input:  An access graph E, each edge e_ij ∈ E is initialized          4
        to none                                                       5
Output: Access graph of hosts H with maximal access                  6
        edges E                                                       7
                                                                      8
For each h_i ∈ H Do                                                   9
   For each h_j ∈ H Do                                                10
      For each (h_i, h_j) ∈ T Do                                      11
         If access((h_i, h_j) ∈ T) > access(e_ij) Then                12
            /* Update e_ij ∈ E */                                     13
            e_ij.access = access((h_i, h_j) ∈ T)                      14
            If access(e_ij) == admin Then                             15
               Stop, move to next host-pair                           16
```

Figure 1. `findInitialAccess` **algorithm.**

Algorithm `findInitialAccess` deserves a few comments. We include in the set of hosts, one outside host with no privileges and no exploits. This host represents the basic attacker. We also consider a limited, ordered set of access levels, namely, `none`, `connectivity`, `pass-through`, `user`, and `admin`. Our notion of access level could clearly be refined, but we take the position that it is an adequate start. Initially, access graph E has each edge $e_{ij} \in E$ initialized to access level `none`. In some cases there may be multiple trust relationships between hosts in a network. In these cases, `findInitialAccess` is designed to examine each relationship and retain the one that provides the *highest* level of access (lines $12-14$). To help reduce computational cost, lines $15-16$ cause the algorithm to cease checking for new access levels if the current level of access for a given edge is already at its highest.

The next step in our model is to calculate *maximal* obtainable level of access between all the hosts in the network using each host's known exploits. If sufficient connectivity on the appropriate ports exists between two hosts, an exploitable vulnerability exists on the destination host, and the source host has all of the prerequisites for the exploit, an edge can be added from source to destination. For book-

keeping purposes, edges are tagged with a route ID, source host, destination host, the means by which the edge was achieved (trust relationship, exploit name, etc.), the access level achieved, bugtraq ID, and a chain ID flag to determine if the edge was part of a chain of exploits. The chain ID flag will be empty if it is not part of a chain, otherwise the route ID of the last edge used in the chain is indicated. The algorithm `findMaximalAccess`, shown in Figure 2, calculates the maximal access each host has on each other host in presence of applying any exploits. Here `access()` returns the current level of access for a given edge e_{ij} (an edge between hosts i and j), or a possible higher level access gained through an exploit x_k launched from host i against the target host j.

```
findMaximalAccess(H, E, X, V) :                                  1
INPUT:  A set of host nodes, H                                   2
INPUT:  A set of access edges, E                                 3
INPUT:  A set of network exploits, X                             4
INPUT:  A set of vulnerabilities at each host, V                 5
OUTPUT: Access graph of hosts H with maximal access              6
        edges E                                                  7
                                                                 8
/* Direct Exploits */                                           9
For each h_i ∈ H Do                                              10
   For each h_j ∈ H Do                                           11
      If access(e_ij) == admin Then                              12
         Stop, move to next host-pair                            13
      For each v_k ∈ V exhibited by host h_j Do                  14
         For each x_w ∈ X against v_k Do                         15
            If all preconditions x_w are TRUE Then               16
               If access(x_w) > access(e_ij) Then                17
                  /* Update edge e_ij ∈ E */                     18
                  e_ij.access = access(x_w)                      19
               If access(e_ij) == admin Then                     20
                  Stop, move to next host-pair                   21
                                                                 22
/* Indirect Exploits */                                         23
For each h_k ∈ H Do                                              24
   For each h_i ∈ H Do                                           25
      If access(e_ik) > none Then                                26
         For each h_j ∈ H Do                                     27
            If access(e_ij) == admin Then                        28
               Stop, move to next host-pair                      29
            If access(e_kj) > access(e_ij) Then                  30
               /* Update edge e_ij ∈ E */                        31
               e_ij.access = access(e_kj)                        32
               e_ij.chainID = e_ik                               33
```

Figure 2. `findMaximalAccess` algorithm.

The algorithm `findMaximalAccess` examines both direct exploits and chained sets of exploits (indirect edges). Lines 9 through 21 attempt for each host to run the best (direct) available exploit against each of it's neighbors (e.g., hosts they can communicate with). In some cases there may be an exploit that can be run by a host against itself, these are often referred to as "self–elevation of privilege" attacks. For example, on several platforms including Solaris 2.5 through 2.6 the "dtappgather symlink" vulnerability (`bugtraq` 131) can be used to overwrite any file present on the filesystem, regardless of the owner of the file, because of improper ownership checking. Using this exploit a malicious user can alter files and permissions to gain greater access on the affected machine. Thus, the algorithm `findMaximalAccess` includes (h_i, h_i) as a valid host pair. An interesting aspect of the model is that successful exploits carried out on one host can satisfy the preconditions for a different exploit on some other host; resulting in an attack chain. Our model captures this through the use of *indirect* edges—a transitive aspect of an attack chain is considered. For example, if there exists host $\mathcal{A}$ with `user` level access to host $\mathcal{B}$, but there also exists `admin` level access from host $\mathcal{B}$ to host $\mathcal{C}$, then from the penetration testing perspective, this is equivalent to $\mathcal{A}$ having `admin` level access to $\mathcal{C}$. Our access graph would use an indirect edge e_{ac} with access level `admin` to capture the aforementioned scenario. The second half of `findMaximalAccess`, lines $23-33$, computes the transitive closure (*cf.* [4]) of our access graph. This is done for each host by attempting to improve their access to another host by using the other exploits that have been carried out. To improve the algorithm performance, line 26 is used to test whether–or–not access level of edge e_{ik} is higher than `none`. Since i and k values do not change as j changes, the test for e_{ik} edge is moved out of the inner–most loop. Note that the monotonicity assumption ensures that this process will converge, will not involve backtracking, and hence is computationally feasible. To help minimize computational cost, `findMaximalAccess` has similar break points to `findInitialAccess` where the algorithm will stop checking for new access levels if an edge already has the highest level of access.

The information to develop this type of access graph can be discovered directly with vulnerability scanning tools and resources such as the `bugtraq` database. It is helpful to sort the possible exploits by level of access gained. The first exploit for which all of the preconditions are satisfied is guaranteed to be the most powerful exploit available. Once an edge is added, the algorithm can then be modified to move on to the next pair of hosts, subsequently reducing the computational cost.

The computational cost of developing our access graph can be roughly analyzed as follows. `findInitialAccess` computes the intended access levels of the network using a set of trust relationships, T. In the algorithm, there are as many nodes, n, as there are hosts in the network, and each host pair is analyzed, hence making a quadratic number of edges, n^2. Therefore making the required number of computations Tn^2. Since `findMaximalAccess` uses exploits in its algorithm to determine the maximal access in a network, it has a slightly higher computational cost. The first part of

`findMaximalAccess` examines only the direct edges in a network, so in the worse case the number of computations required is XVn^2, where X is the total number of exploits for all the hosts, V is the total number of vulnerabilities present in the network, and n is the total number of hosts. The second half of `findMaximalAccess` (indirect exploits) examines chains of exploits by examining existing edges in combination with exploits iteratively until all possible paths have been examined. In the densest access graph, with all access levels higher than `none`, we can compute the transitive closure (*cf.* [4]) in time proportional to n^3. Therefore the total computational cost of `findMaximalAccess` is $XVn^2 + n^3$.

3. Analysis

Given a stable access graph, we suggest that an analyst can use this information in a variety of ways to help secure the network. First, we suggest that an analyst can identify a host with an unacceptable level of compromise and a solution to fix the problem. This involves, in some but not all cases, patching the vulnerabilities of the exploit that led immediately to that level of access. The exploit to patch is the one identified by the edge tags explained above. This is not true in all cases, which is the rationale for the approaches in the literature that compute entire attack graphs. In this paper, however, this is exactly the trade–off we make: we accept that the analyst may make a suboptimal choice of which vulnerability to patch, in return for having the computational complexity be polynomial instead of exponential. Although a complete attack would provide an analyst with the minimal number of changes required to secure the network, they are extremely expensive to build. Our approach will present the analyst with an area that needs to be fixed and since it is computed in polynomial time it provides real–time results. The approach cannot provide a comprehensive list of changes that are required in one iteration, however it can be accomplished through several iterations.

Once the access graph stabilizes, the analyst can then compute the effect a patch of a vulnerability has on the network access graph by first checking if there exists an *equivalent* exploit. An equivalent exploit is an exploit that will result in the same access on the target host and whose *all* preconditions are satisfied by the same set of hosts. If this is the case, the analysis is complete. If not, then we need to find the next most powerful, applicable exploit for edges that directly use that exploit. For each effected direct edge, we will attempt to find the next best access it can achieve by examining trust relationships and exploits. Then for every other edge whose chain depended on the now infeasible edge is re–examined, as it may no longer be feasible as well. In each case, the next most powerful exploit is chosen. This occurs recursively until all affected edges are examined and the graph stabilizes. Now the analyst can review the new network configuration to see if it is acceptable.

The access graph developed by an analyst can also be used as a means of providing near real–time early warning of potential attacks. Intrusion detection systems (IDS) are used to help the system administrator detect when an attack is being carried out in real–time in order to minimize the damage. Our approach can be computationally run in real–time to provide early warning of vulnerable areas to prevent exposure from ever occurring. Given a stable access graph, an analyst can determine risk areas and secure them to preempt an attack in near real–time.

```
potentialNewMaximalAccess(H, T, E, X, V, h_a) :                          1
INPUT:  A set of host nodes, H                                           2
INPUT:  A set of trust relationships, T                                  3
INPUT:  A set of access edges, E                                         4
INPUT:  A set of network exploits, X                                     5
INPUT:  A set of vulnerabilities at each host, V                         6
INPUT:  A new attacker node, h_a                                         7
OUTPUT: Access graph of hosts H with maximal access                      8
      edges E                                                            9
                                                                         10
/* Trust Relationships */                                               11
For each h_i ∈ H Do                                                      12
   For each (h_a, h_i) ∈ T Do                                            13
      If access((h_a, h_i) ∈ T) > access(e_ai) Then                      14
         /* Update e_ai */                                               15
         e_ai.access = access((h_a, h_i) ∈ T)                            16
         If access(e_ai) == admin Then                                   17
            Stop, move to next host-pair                                 18
                                                                         19
/* Direct Exploits */                                                   20
For each h_i ∈ H Do                                                      21
   If access(e_ai) == admin Then                                         22
      Stop, move to next host-pair                                       23
   For each v_j ∈ V exhibited by host h_i Do                             24
      For each x_k ∈ X against v_j Do                                    25
         If all preconditions x_k are TRUE Then                          26
            If access(x_k) > access(e_ai) Then                           27
               /* Update edge e_ai ∈ E */                                28
               e_ai.access = access(x_k)                                 29
            If access(e_ai) == admin Then                                30
               Stop, move to next host-pair                              31
                                                                         32
/* Indirect Exploits */                                                 33
For each h_k ∈ H Do                                                      34
   For each h_i ∈ H Do                                                   35
      If access(e_ik) > none Then                                        36
         For each h_j ∈ H Do                                             37
            If access(e_ij) == admin Then                                38
               Stop, move to next host-pair                              39
            If access(e_kj) > access(e_ij) Then                          40
               /* Update edge e_ij ∈ E */                                41
               e_ij.access = access(e_kj)                                42
               e_ij.chainID = e_ik                                       43
```

Figure 3. `potentialNewMaximalAccess` **algorithm.**

We also suggest that the access graph can be used to conduct analysis on the potential impacts of giving different permissions or credentials to users. For example, this type of analysis can be done to model *insider* attacks. To compute this type of analysis, an additional node is used for the special case attacker along with the attacker's credentials and permissions added to the set of trust relationships. Then the `potentialNewMaximalAccess` algorithm, shown in Figure 3, is used to determine what new access levels are gained.

`potentialNewMaximalAccess` is designed to leverage the existing access graph when adding additional hosts to the graph. The algorithm first focuses on adding any existing trust relationship edges the new host can achieve. This is accomplished on lines 11 through 18, where the new attack host attempts to leverage its access to other hosts in the network, in absence of applying any exploits. Next the algorithm computes and possibly updates any new additional relevant direct and indirect edges. This is accomplished on lines $20 - 43$. Once `potentialNewMaximalAccess` is finished computing a new access graph, the analyst can now determine what additional accesses a user gains by having a new specific set of permissions. This information can then be used to modify the network to minimize the undesired access.

The computational cost saving of running `potentialNewMaximalAccess` instead of `findInitialAccess` and `findMaximalAccess` is seen when computing new access edges from trust relationships and direct exploits. The new host will attempt to gain new access on the other hosts, n. In the worse case, the algorithm will examine all trust relationships, T, exploits, X, and vulnerabilities, V, for each host in the network making the cost $Tn + XVn$. The indirect edges are computed in the same fashion as in `findMaximalAccess`, thus making the total computational cost of `potentialNewMaximalAccess` $(Tn + XVn) + n^3$. Since the total computational cost is smaller, it makes sense for an analyst to use this algorithm rather than to recompute the entire access graph from scratch.

Another way our approach can be used to secure a network is by analyzing network policy rules. An analyst can use a stable access graph to determine policy violations in a network. Policy rules, such as limiting access to specific areas, is a typical way system administrators think about safeguarding their network. Since our approach meshes well with this typical mental model, such policy rules can be expressed as access edges. For example, a network administrator may not want any outside access to a particular internal database host. This can be expressed as a set of access edges (Any, Database, attacker, any, pass through), (Any, Database, attacker, any, user), (Any, Database, attacker, any, admin). An analyst, given a set of policy rules expressed as edges, P, and the computed set of maximal access edges, E, can determine the policy rules violations using $P \bigcap E$. An analyst can not only determine which policy rules are compromised, but also can determine the exploit(s) that lead to these violations by examining the edge's tags. The edge tags will identify any direct exploits that were used to compromise the policy rules, as well as, any indirect paths with a complete trail of how they were achieved. Through several quick iterations of fixing the problematic exploits, the analyst can properly prevent policy rules from being violated.

4. Example

To demonstrate how our approach works, we have created a small example network. In it, there are three target hosts. These are a publicly accessible web server, a publicly accessible file server, and a back–end database server. In addition, there is a host to represent the attacker located somewhere out on the Internet. The target hosts are protected by a firewall, which is limiting connectivity between the Internet, the DMZ, and the internal networks. The firewall rules are shown in Table 1. Finally, each of the hosts has certain vulnerabilities, which the attacker would like to be able to exploit. These are shown in Table 2 along with the corresponding access level gained when these vulnerabilities are exploited. Data in Table 2 is easily obtained from vulnerability scanners such as `Nessus` and `Retina`.

Source	Destination	Service	Action
All	Web	http	Allow
All	Web	ftp	Allow
All	File	ftp	Allow
Web	Database	Oracle	Allow
File	Database	ftp	Allow
All	All	All	Deny

Table 1. Firewall Rules

Host	Vulnerability	Bugtraq	Access
Web	Apache Chunked-Enc.	5033	admin
Web	Wu-Ftpd SockPrintf()	8668	admin
File	FTP Bounce	126	pass-thr.
Database	Oracle TNS Listener	4033	admin

Table 2. Host Vulnerabilities

The chief data structure in our model is an access graph. Consequently, initialization of the model proceeds by creating a node in the access graph for every host, and then adding directed edges to represent any initial trust relationships that may exist using `findInitialAccess`. In this

example our set of host nodes, H, is {Database (d), File (f), Web (w), Attacker (a)}. Trust relationships, T, are represented as (source, destination, access level) and determined based on the information gathered in network scanning tools. In this example, the trust relationships are: { (d, f, connectivity), (d, w, connectivity), (f, d, user), (f, w, user), (w, d, user), (w, f, user), (a, w, connectivity), (a, f, connectivity)}. Lines $9-10$ of `findInitialAccess` ensure that every host pair is examined and lines $11-16$ ensure that the appropriate edges are added. In this example, the first host pair to be examined is (d, d). Since (d, d) does not have a trust relationship in T, the algorithm moves to the next host–pair, namely (d, f). Since (d, f) has a trust relationship (line 11) that will give Database `connectivity` on File (this is higher than no access that is currently in our graph), an edge will be updated to reflect this on line 14. Using the defined format for edges, it will be marked as (D_F, Database, File, Trust, Connectivity, -, -). Since the highest access has not been achieved, the algorithm will examine the next trust relationship between Database and File. However since only one exists, the algorithm returns to line 10 to examine the next destination host, Web. The algorithm looks at the first trust relationship and on line 12 determines that Database can have `connectivity` on Web and it is higher than no access. The edge (D_W, Database, Web, Trust, Connectivity, -, -) will be updated on line 14. This process continues until there are no remaining destination hosts. Then the algorithm returns to line 9 to get another source host to conduct the same analysis. Once the algorithm is completed, the resulting graph for our example is shown in Figure 4. This graph shows how much access each host has to each other host prior to any exploitation attempts. Note, that in order to keep the graph readable and to emphasize our point, only the edges with `pass-through` access or higher are included in the diagrams. However, all edges are still used during the computations.

The next step in the analysis is to determine the best exploit a host can run against each of its neighbors, which is accomplished by `findMaximalAccess`. In this example, the algorithm will first start with Database attempting to attack itself. In this case, the source and destination hosts are the same. The only exploit available on the Database is (`bugtraq` 4033), which must be done as a remote attack and thus no edge can be added. With the exploits exhausted, the algorithm moves onto the next destination host, File. Since the edge between Database and File exists, and is not at `admin` level of access, the analysis proceeds in an attempt to improve the current access. File's only vulnerability is the FTP Bounce exploit (`bugtraq` 126). Although Database has `connectivity` access on File, it does not satisfy all the preconditions necessary run the exploit, so the edge is not updated. Having examined all the exploits available on File, the algorithm moves to the next destination

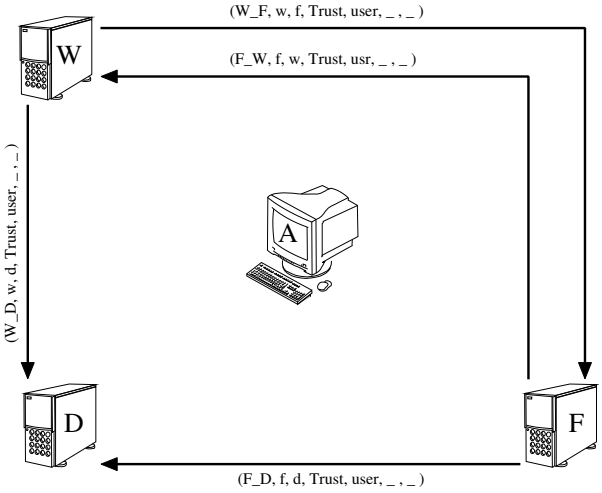

Figure 4. Access graph with intended access.

host, Web. Since the edge between Database and Web is not at `admin` level of access, the algorithm will attempt to use one of Web's two known exploits. Web's Apache service is vulnerable to a remote root exploit (`bugtraq` 5033). Database can communicate with this service on Web and by exploiting this vulnerability, Database gains `admin` level control of Web. Since `admin` is higher than its current access level of `connectivity`, we update the directed edge between Database and Web, labeled by a route ID (D_W), with the vulnerability that was used (Chunk), the access that was gained (admin), and the `bugtraq` ID 5033: (D_W, d, w, Chunk, admin, 5033, -). The chain flag remains empty because this edge is not part of a chained attack. Since the edge from Database to Web is at the highest level of access, there is no need to continue examining any more exploits and the algorithm breaks out of the *For* loop on line 13. This process continues until all the possible source and destination host pairs have been examined.

The second half of `findMaximalAccess`, lines $23-33$, calculates access achieved through a series of attacks. The algorithm works by examining existing edges that can be leveraged to achieve an attack on another host through an indirect edge. In essence, the algorithm computes the transitive closure of the access graph. The algorithm works by representing all possible host–pair combinations as an adjacency matrix. Systematically, each edge is examined and updated only if it is more beneficial to use an indirect attack path (through some other host(s)) rather than a direct edge to the target. More specifically, an edge e_{ij} (between hosts h_i and h_j) is updated only when it is more beneficial for host h_i to go through some other (intermediate) host h_k whose access level to host h_j is higher than the current level of access for the edge e_{ij} (i.e., line 30 of the algorithm). The

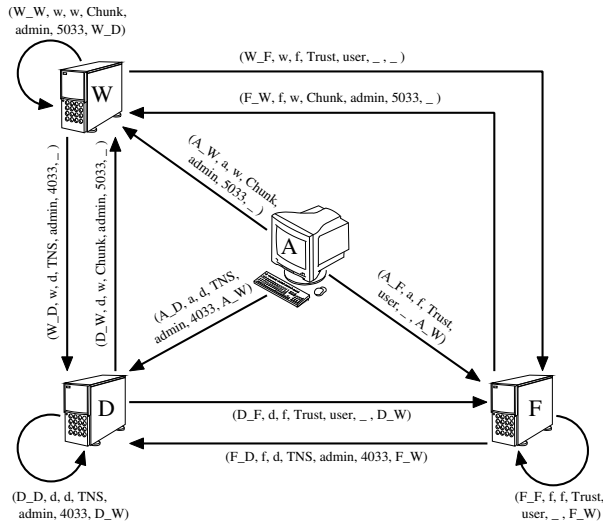

Figure 5. Maximal possible access due to all exploits.

edge label captures this by marking the `chainID` field of edge e_{ij} with host h_k. In this example, the algorithm first selects the host pair whose source being examined is also the destination host corresponding to edge e_{dd}. Since the current level of access for the edge e_{dd} is only `none`, the algorithm tries to improve it. The edge gets updated since there exists an indirect path to leverage the access by going through Web host. First, Database gains `admin` level access on Web through Chunk (`bugtraq` 5033) exploit. Then, TNS (`bugtraq` 4033) is used to gain `admin` level access from Web to Database. This is captured by labeling edge e_{dd} with (D_D, d, d, TNS, admin, 4033, D_W). Since this is a chained attack, the chain flag is not longer empty and the last edge used in the chain of exploits, namely D_W, is used as `chainID` flag. The algorithm then tries to leverage the access level between the next host pair (Database and File). Through available trust relationships, vulnerabilities and exploits, the algorithm uses Database's `admin` control of Web to improve access to File. Currently Database only has `connectivity` to File. However, by exploiting the trust relationship between Web and File, Database can leverage its level of access to File to that of `user` by using Web as an intermediate host. Lines $31 - 33$ cause the corresponding edge e_{df} to update its label as (D_F, d, f, Trust, user, -, D_W). Since this is also a chained attack, the chain flag is not longer empty and `chainID` is marked with the last edge used in the chain of exploits, namely D_W. Since access edge between Database and Web is already `admin`, the algorithm moves to the next host–pair. This process will continue until all host pairs have been examined and no edges can be updated. To optimize the algorithm, two checks are being done. First, line 26 checks whether–or–not the path through an intermediate host has an access level higher than `none`. If the answer is yes, the inner–most loop is executed since we are at least guaranteed `connectivity` through the intermediate host. If the answer is no, it makes no sense to continue since there does not exist an edge to the intermediate host whose access level is higher than `none`. Second, lines $28 - 29$ quit the inner–most loop if the currently examined edge is already at `admin` level of access. The resulting access graph in Figure 5 shows the best exploitation paths that can be created in the example network.

Once the graph stabilizes, the analyst can then examine the undesirable accesses and look for ways to remedy them. With so much access being achieved due to the Apache vulnerability (`bugtraq` 5033) on the Web server, this would be an obvious place to start remediation efforts. According to the `bugtraq` notice, upgrading to a newer version of the Apache software will resolve this problem. Assuming that this occurs, it will invalidate any direct edges that terminate on Web and use Apache Chunk vulnerability and any chained edges that are dependent on the now infeasible edges. The affected edges (represented as "dashed" lines) are shown in Figure 6.

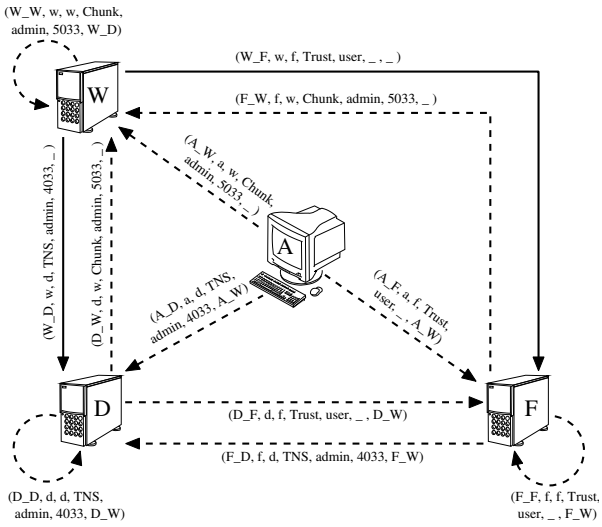

Figure 6. Effect of patching Apache Chunk vulnerability.

This example shows how to compute an access graph using our approach. Through multiple iterations, the network will become secure. Although our approach may not provide an analyst with a minimal set of changes, it does produce the same desired goal of a secure network. Since our approach can be used in real–time analysis because of its polynomial cost, it is desirable for system administrators

to incorporate this approach into their security arsenal.

4.1. Larger Network Example

To further demonstrate that our approach runs in reasonable time for *realistic* networks, we ran our tool on a network comprised of 6 subnets (separated by a border and internal firewalls) and 87 hosts. The experiment was conducted on an Intel Pentium 4 (2.0 GHz) with 512 MBytes of RAM running on Fedora Core 3 (Linux 2.6.9). In our model, there are as many nodes, as there are hosts in the network. Each host pair is analyzed, hence making $87^2 = 7569$ edges in the resulting access graph. Out of these edges, 2088 had level of access higher than `connectivity`. The open source graph visualization program, `graphviz` [8], was used to generate graphs for closer analysis and visualization. First, the system was initialized with the network's topology/configuration before `findInitialAccess` and `findMaximalAccess` algorithms were used. Our tool read this information from couple of files: set of host nodes—H, set of trust relationships between hosts—T, set of available network exploits—X, set of vulnerabilities present at each host in the network—V, and set of firewall rules—F. This took 1.527 seconds to complete. Then, `findInitialAccess` took 0.107 seconds to establish the initial trust relationships between hosts in absence of applying any exploits. `findMaximalAccess` took 1.571 seconds to calculate maximal accesses between all the hosts in the network using each host's known exploits. The `graphviz` tool was then used to generate access graph for visualization and analysis purposes.

By way of comparison, a model presented by Sheyner *et al* [20] took 5 seconds to execute for a network comprised of 3 hosts, but when their sample network was increased to 5 hosts, their tool took over 2 hours to construct a corresponding attack graph. Similar scalability problems are encountered by models which construct complete attack graphs [19, 16, 22], and by alert correlation techniques [13, 14].

5. Related Work

A variety of graph based approaches [2, 25, 16, 22, 23, 6] to modelling network vulnerabilities have been proposed. Swiler *et al* [16, 22] developed one of the initial graph–based formalisms for analyzing network vulnerabilities. The requires/provides model of Templeton and Levitt [23] has been used by many other researchers to model the role of exploit pre– and post–conditions in chaining exploits together. Ritchey and Ammann [19] proposed the use of model checkers to generate attack paths for known exploits. Ramakrishnan and Sekar [17] used a model checker to carry out a related analysis in single host systems with respect to unknown vulnerabilities.

In an extension of the network vulnerability analysis of Ritchey and Ammann cited above, Jha *et al* [10, 9] and Sheyner *et al* [20, 21] used model checking to analyze attack graphs on heterogeneous networks. They consider interconnected network of computers with known vulnerabilities that attackers can combined in order to attack one or more hosts. If an attack succeeds, the authors provide a mechanism to allow the analyst to understand *all* possible attack scenarios. Ammann *et al* [1] introduced a monotonicity assumption and used it to develop a polynomial algorithm to encode all of the edges in an attack tree without actually computing the tree itself. Noel *et al* [15] developed an elegant, though still exponential, algorithm to recursively back substitute exploits with preconditions; the result is a boolean expression describing the initial conditions that lead to a compromise. Also, there have been number of different series of techniques developed [7, 5, 13, 14] that integrate alert correlation methods in order to build possible attack scenarios.

As described above, researchers have proposed a variety of methods to generate attack graphs. The computational complexity for most of their methods quickly becomes exponential in the size of the network as network complexity grows past a few machines. By utilizing penetration tester's worst case possible damage perspective, our model's algorithmic complexity is reduced to polynomial in the size of the network, and so has the potential of scaling well to practical, more realistic networks.

To help feed information into our model, number of different research efforts and tools can be used. Analyst can get vulnerabilities and their associated exploits from well documented public sites such as `bugtraq`. Commercial tools such as `Nessus` and `Retina` can be used to identify known vulnerabilities in a given host or network. Since information from various tools can be in multiple formats and each may provide different information, Vigna *et al* [24] developed `NetMap` [12] program for integrating multiple tools, and for providing information in a more complete and concise format. The information retrieved by the tool is a nice complement to developing our model.

6. Conclusions

In this paper, we developed a host–centric approach to analyzing network vulnerabilities. Our approach is a complement to the attack graph approach, and has the benefit of being computationally feasible on large networks, albeit at the expense of not explicitly identifying every possible attack sequence. Instead, we argue that penetration testers look for specific degrees of compromise on a given host, and focusing on that level of compromise is a natural ap-

proach for a system analyst. Our model can be computed and analyzed in real–time (its algorithmic complexity is polynomial in the size of the network), it can be used to 1) provide near real–time early warning of potential attacks, 2) to identify the network policy rules violations and 3) to conduct analysis on the potential impacts of giving different permissions or credentials to users (modelling insider attacks), thus making it a viable solution for industry.

Acknowledgments

This work was supported in part by the National Science Foundation under grant CCR–0208848. The work of Joseph Pamula was partially supported by the Air Force Research Laboratory, Rome under the grant F30602–00–2–0512 and by the Army Research Office under the grants DAAD19–03–1–0257 and W911NF–05–1–0374.

References

[1] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable graph-based vulnerability analysis. In *Proceedings CCS 2002: 9th ACM Conference on Computer and Commuincations Security*, pages 217–224, Washington, DC, November 2002.

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

[3] Bugtraq. The security vulnerabilities mailing list. http://www.securityfocus.com.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company and The MIT Press, 1998.

[5] F. Cuppens and A. Miege. Alert correlation in a cooperative intrustion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P '02)*, May 2002.

[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] H. Debar and A. Wespi. Aggregation and correlation of intrusion detection alerts. In *Proceedings of Recent Advances in Intrusion Detection (RAID 2001)*, pages 85–103, 2000.

[8] Graphviz. Graph visualization software. http://www.graphviz.org.

[9] 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.

[10] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *Proceedings of the 2002 Computer Security Foundations Workshop*, pages 49–63, Nova Scotia, June 2002.

[11] Nessus. Open source vulnerability scanner project. http://www.nessus.org.

[12] NetMap. Network modeling, discovery, and analysis. http://www.cs.ucsb.edu/∼rsg/NetMap/index.html.

[13] P. Ning, Y. Cui, and D. Reeves. Constructing attack scenarios through correlation of intrustion alerts. In *Proceedings of the 9th ACM Conference on Computer & Communications Security*, pages 245–254, Washington D.C., November 2002.

[14] P. Ning, D. Xu, C. Healey, and R. S. Amant. Building attack scenarios through integration of complementary alert correlation methods. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, pages 97–111, February 2004.

[15] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *Proceedings ACSAC 2003: 19th Annual Computer Security Applications Conference*, pages 86–95, Las Vegas, December 2003.

[16] 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.

[17] 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.

[18] Retina. Network security scanner. http://www.eeye.com/html/products/Retina/.

[19] 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.

[20] 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)*, pages 254–265, Oakland, CA, May 2002.

[21] O. Sheyner and J. Wing. Tools for generating and analyzing attack graphs. In *To appear in Proceedings of International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science, 2005.

[22] 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.

[23] 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.

[24] G. Vigna, F. Valeur, J. Zhou, and R. Kremmerer. Composable tools for network discovery and security analysis. In *Proceedings ACSAC 2002: 18th Annual Computer Security Applications Conference*, Las Vegas, December 2002.

[25] 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.