Practical Attack Graph Generation for Network Defense*

Kyle Ingols, Richard Lippmann, Keith Piwowarski
MIT Lincoln Laboratory
244 Wood Street
Lexington, Massachusetts 02420-9108
Email: {kwi, rpl, piwowk}@ll.mit.edu

Abstract

Attack graphs are a valuable tool to network defenders, illustrating paths an attacker can use to gain access to a targeted network. Defenders can then focus their efforts on patching the vulnerabilities and configuration errors that allow the attackers the greatest amount of access. We have created a new type of attack graph, the multiple-prerequisite graph*, that scales nearly linearly as the size of a typical network increases. We have built a prototype system using this graph type. The prototype uses readily available source data to automatically compute network reachability, classify vulnerabilities, build the graph, and recommend actions to improve network security. We have tested the prototype on an operational network with over 250 hosts, where it helped to discover a previously unknown configuration error. It has processed complex simulated networks with over 50,000 hosts in under four minutes.*

1 Introduction

Defending large enterprise networks is very difficult. A defender must be able to locate all paths into the network and prevent attackers from using them; an attacker needs to find only one unprotected path. A network defender has the advantage of intimate knowledge of the network, such as the ways traffic may move through it, the services running on it, and the vulnerabilities in those services. A defender can use that knowledge to improve situational awareness.

Attack graphs are one way to leverage that data. There are many different papers on attack graphs and many representations, but the core idea remains the same: an attack graph shows the ways an attacker can compromise a network or host. Defenders can then use the attack graph to identify critical bottlenecks and work to secure those bottleneck hosts and services first.

Previous work on attack graphs has suffered from two substantial problems [19]. First, a large amount of source data must be assembled to accurately build an attack graph, and many past papers assume the data is both preprocessed and extensive. While this assumption is valid for a theoretical graph design, it is unsuitable for implementation. Our system assumes the data is provided in common formats, and performs all of the necessary preparatory work. Second, past research has had difficulty scaling to large, enterprise-size networks with tens of thousands of hosts. We have focused heavily on scalability concerns and have developed a system capable of handling very large networks in reasonable amounts of time using commodity hardware.

Our system, called NetSPA, is able to import data from common sources, including the Nessus [4] vulnerability scanner, Sidewinder and Checkpoint firewalls, the CVE [2] dictionary, and the NVD [20] vulnerability database. It automatically computes *reachability*, or the ability for a given host to connect to open ports on all hosts in the network. It rapidly generates an attack graph showing how an attacker can maximally compromise the targeted network. The tool can model an attacker able to start from any IP address, maximally exploiting any special "holes" in the perimeter firewalls. Our system builds *multiple-prerequisite graphs*, or MP graphs, which are faster to build and have greater expressive power than our previous work's *predictive graphs* [17, 18]. MP graphs are able to model portable credentials, such as passwords, which an attacker can use from anywhere to compromise a target. MP graph construction is substantially faster than all other published attack graph results of which we are aware [19, 25].

We have implemented a working prototype in Perl and C++ that is capable of automatically computing reachability from readily available source data, generating an MP graph, analyzing it, and producing the graph and recommended actions as output. We have explored our system's scalability by evaluating simulated networks with up to 50,000 hosts,

*This work is sponsored by the United States Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)
0-7695-2716-7/06 $20.00 © 2006 IEEE

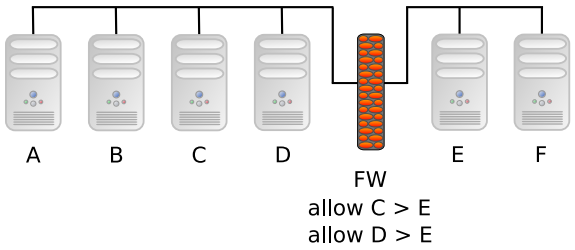

Figure 1. Simple Example Network

and verified its value by conducting a field test on an operational network with over 250 hosts.

For the remainder of the paper, the simple network shown in Figure 1 will be used to explain the data required, computations performed, and results obtained. The attacker begins on host A, and does not use arbitrary source IP addresses. Every other host in the network has a single open port, with a single remotely-exploitable vulnerability. The firewall permits hosts C and D to communicate with host E and drops all other traffic.

The paper is organized as follows: Section 2 explores the source data required for graph generation. Vulnerability evaluation is touched upon in Section 3. Reachability generation is discussed in Section 4. Section 5 explains the structure of the multiple-prerequisite attack graph and how it is built. Section 6 proposes techniques for using the resulting graph, by simplifying it for viewing and by automatically analyzing it to produce recommended defensive actions. Tests on real and simulated networks are covered in Section 7. Related work is reviewed in Section 8. Section 9 concludes the paper.

2 Data Used by the NetSPA tool

NetSPA's network model supposes that an individual *host* possesses one or more *interfaces* which have listening addresses. These interfaces have zero or more open *ports*, accepting connections from other hosts. A host and its interfaces may have *rules* that dictate how network traffic may flow to, and through, the host. A port has zero or more *vulnerability instances*, particular flaws or configuration choices which may be exploitable by an attacker. Each interface on a host is connected to a *link*, representing some combination of hubs and switches connecting a set of interfaces together. An attacker is able to obtain one of four *access levels* on a host: "root" or administrator access, "user" or guest access, "DoS" or denial-of-service, or "other," a confidentiality and/or integrity loss. The combination of a host and an access level is an attacker *state*. A state may provide the attacker zero or more *credentials*; vulnerability instances may require zero or more of them. An attacker obtains a host's *reachability* if "root" or "user" access is achieved. Reachability and credentials serve as *prerequisites* to exploitation of a vulnerability instance. This model is admittedly simplistic, but we can populate it using available data and it is realistic enough for our needs.

Our use of the term "vulnerability" is somewhat unconventional. In NetSPA, a vulnerability is any way an attacker could gain access to a system. Examples include software flaws, trust relationships, and server misconfigurations. A feature, such as remote login with the use of a private key, is a vulnerability from the point of view of an attacker.

NetSPA requires a large amount of data to populate its model, but system administrators often collect this data as a matter of course. The core pieces are network topology, vulnerability information, and credentials. NetSPA itself runs offline using the provided data, minimizing the risk of an attacker obtaining the source data or resultant graph.

Network topology is obtained from both the user and the Nessus vulnerability scanner [4]. The user must provide a "map" of the network, enumerating the links and indicating which Nessus scans belong to which link, and which multihomed hosts' interfaces are on which links. The map is fairly small and static, and should remain a tractable task as larger networks are considered. Firewall rulesets are also a component of the network model; the user provides the original rulesets from the firewall directly to NetSPA, which converts the rulesets to an internal format for use. Nessus provides information on individual interfaces, ports, and vulnerability instances.

Our concept of "credential" is any information used as access control: a password or a private key, for example. It may be possible to automatically determine where credentials are and what they protect – [28] has done so for SSH, for example – but any data import will be specific to an application or platform, hampering its use. We do not currently have any readily available, automatable sources of credential data, nor have we written importers that could automatically import credential-related data.

For the sample network in Figure 1, two Nessus scans are required: one on the left side of the firewall, targeting hosts B, C, and D, and one on the right, targeting E and F. If scans are not carried out inside each subnet, the firewall could prevent Nessus from finding and reporting every host, port, and vulnerability.

Notably absent are data regarding non-cyber attacks, such as social engineering attacks and physical attacks on a datacenter's perimeter. While these threats are real and relevant, we have focused our efforts on data that can be readily obtained in an automated fashion. Tying the generation of the attack graph to data that can be obtained quickly ensures that any new vulnerabilities are discovered, evaluated, and reported as early as possible, minimizing the time the network is exposed to attack.

We additionally utilize non-network-specific data which

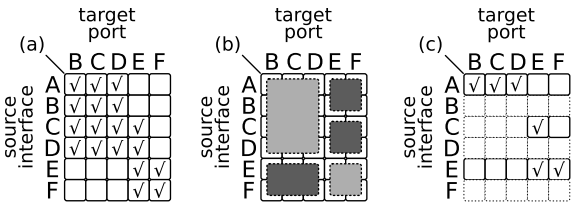

Figure 2. The (a.) Reachability Matrix, (b.) Reachability Groups, and (c.) Collapsed Reachability Matrix for the Simple Example Network

needs to be imported only once. These data, discussed in Section 3, allow NetSPA to determine the impact of the vulnerability instances reported by Nessus.

NetSPA must derive a few core pieces of knowledge to build an attack graph. For a given host, the tool must know which ports the host can reach. For each instance of a vulnerability, the tool must know what is required to exploit it and what is gained by exploiting it.

3 Vulnerability Evaluation

In addition to network-specific data, the system requires additional knowledge about vulnerabilities. Nessus can identify the hosts, interfaces, and ports on a network, pinpointing where vulnerabilities are. However, Nessus does not clearly articulate a vulnerability's prerequisites or what an attacker gains by exploiting it. We define a straightforward representation to model vulnerability prerequisites and postconditions.

In our model, a vulnerability has *locality*, indicating whether it is remotely exploitable. It provides an *effect*, which is one of the four access levels an attacker can obtain in our model: root, user, DoS, or other. When known, any credentials required for exploitation are also considered.

The vulnerability model is simple because available data constrains the fidelity. Vulnerability databases such as NVD [20] and Bugtraq [1] describe vulnerabilities' impacts in detail. Unfortunately, much of the available data is intended for human consumption and is sometimes incorrect or out of date [9]. NetSPA uses a simple logistic regression classifier, trained on a hand-evaluated sample set, to automatically classify vulnerabilities. Details on the classifier are available in [18].

Our attacker model is likewise simple: the attacker knows about all vulnerabilities and will successfully exploit all reachable vulnerabilities to their fullest effect. A worst-case attacker model prevents false negatives and requires no additional assumptions about the potential threat.

4 Computing Reachability

Computing reachability is a complex, time-consuming task, but an attack graph system applicable to real networks must obtain and use reachability information. Reachability computation uses available information on the network topology, filtering devices, and hosts to find paths between source hosts and target ports. The rulesets of all filtering devices on the network must be imported and modeled.

A straightforward method to compute reachability is to try to reach every known target IP address and port from every host in the network. Such an approach would generate a *reachability matrix*, where a row represents a source interface on a host, a column represents a target port on a destination interface, and each cell indicates whether or not the source can reach the target. A reachability matrix for the sample network of Figure 1 is shown in Figure 2a. This is correct, but it scales poorly in terms of both space and time.

We have made three improvements to the straightforward approach. We collapse sections of the matrix into *reachability groups* [18], saving large amounts of both time and memory. Filtering rulesets are collapsed into Binary Decision Diagrams (BDDs) [10], allowing the reachability system to traverse a set of filtering rules in constant time. We also hypothesize a "generic attacker" by selecting a link on which the attacker will begin and allowing the attacker to use the most advantageous source IPs.

Reachability groups identify redundancies in the reachability matrix and collapse submatrices into single subrows before computing the contents, saving both time and space. First, intra-subnet reachability which is not influenced by any filtering devices can be collapsed into a single subrow, because every source interface within the subnet will have the same reachability to all ports within that same subnet. The two intra-subnet reachability groups for the sample network are shown as light grey boxes in Figure 2b.

Second, inter-subnet reachability can be collapsed by identifying sets of interfaces within a subnet which are treated identically by the filtering devices on the network. If the source IP addresses of a set of interfaces on the same subnet match in the same set of filtering rules, the interfaces are grouped together and reachability is computed for only one of them. Grouping interfaces is expensive, but far less expensive than actually recomputing reachability. The three inter-subnet reachability groups for the sample network are shown as dark grey boxes in Figure 2b.

Reachability groups drastically reduce the cost of computing the reachability matrix. The collapsed reachability matrix for the sample network, shown in Figure 2c, is 60% smaller. The reduction is often much larger in sizeable networks as a larger number of interfaces collapse together into a handful of reachability groups.

Reachability groups reduce the number of cells which

must be computed. BDDs reduce the cost of computing an individual cell. Our implementation uses the ideas of the FIREMAN firewall modeler [33] to collapse filtering rulesets into a BDD, permitting constant-time traversal of a ruleset. Like FIREMAN, the NetSPA prototype uses the BuDDy library [16] to manipulate BDDs.

It is important to consider an attacker coming from an arbitrary source location. Production firewall rulesets are often large, and may contain mistakes allowing unintended traffic [32]. NetSPA is able to exercise all of these rules by discovering the source IP addresses of greatest advantage to an attacker. The tool collects a representative IP address from every IP singleton, subnet, and range used in rules in the network, and uses each of those addresses in turn to compute reachability for the attacker's starting location. This can uncover interesting flaws and vulnerabilities in the network configuration. The sample network in Figure 1 and graphs in Figure 3 do not use this feature.

5 Attack Graphs

It is easy to determine attack paths for the sample network of Figure 1 by hand. The attacker from host A can directly compromise hosts B, C and D. From C or D, the attacker can traverse the firewall and compromise host E. From E, the attacker can compromise host F, completing the process. The paths are *monotonic*, assuming an attacker will never need to relinquish a state. The assumption of monotonicity greatly simplifies the task of modeling attacker actions and has been made in several other papers [7, 15, 25].

Part of a full graph [8] for the sample network is shown in Figure 3a. Nodes correspond to states, and edges to vulnerability instances. Full graphs add a node to the graph if no ancestor node has the same state as the new node and was reached via the same vulnerability as the new node. To conserve space, the children of the B and C nodes at the top level are omitted from the figure.

Full graphs illustrate every order in which the attacker can compromise the hosts in the network – first B, then C, or vice versa. However, they scale as $O(n!)$ and quickly become too large to compute as the network size increases.

A predictive graph [17, 18] is shown in Figure 3b. Nodes and edges have the same meanings as in the full graph. Predictive graphs add a node to the graph if no ancestor of the current node used the same vulnerability to obtain the same state as the new node. They avoid much of the redundant structure of the full graph and correctly predict the impact of removing any of the vulnerability instances in the network. Predictive graphs are much faster to build than full graphs, and we have used them on large, real networks, but they still include redundant structure in some cases. For example, the subtrees beneath C and D in Figure 3b are redundant, a phenomenon called a *firewall explosion* in [18].

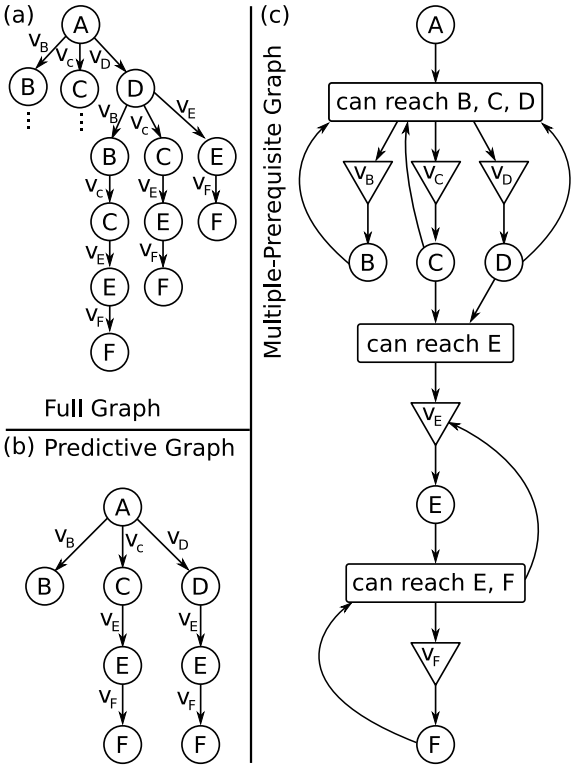

Figure 3. The (a.) Full Graph, (b.) Predictive Graph, and (c.) Multiple-Prerequisite Graph for the Example Network

Predictive graphs are also unable to model credentials.

A multiple-prerequisite (MP) graph for the simple network is shown in Figure 3c. The MP graph has contentless edges and three node types, discussed in Section 5.1. It explicitly represents the prerequisites of an attack. In this example, the only prerequisite is reachability.

The MP graph's cycles embed the information contained in full and predictive graphs without the redundant structure. If no credentials are used, we can build a full graph from the MP graph by exploring the MP graph in a depth-first manner, stopping the exploration when we reach a vulnerability instance already used on the path from the root to the current node. Similarly, we can build a predictive graph from the MP graph by exploring the MP graph in a breadth-first manner.

During graph construction, both predictive and full graphs must attempt actions which are not shown on the graph. The bottom node F of the full graph of Figure 3a, for example, must explore (and then prune) every possible vulnerability that could be reached from host F. The MP graph avoids this problem by evaluating a prerequisite, such as the ability to reach hosts E and F, only once. Full and predictive graphs must evaluate a prerequisite once for each state that

provides the prerequisite.

The MP graph also shows all hosts which can be compromised from any host the attacker has compromised. Host F, for example, is capable of compromising host E. The MP graph shows this (via backedges), but the other graph types do not. We can take advantage of this property to generate "all sources, all targets" MP graphs, showing every malicious action that could take place from any host or attacker starting location to any host in the network. Because it contains every potential attack path in the network, such a graph could be useful to tools attempting to correlate IDS alerts with known attack paths.

5.1 MP Attack Graph Structure

The maximum number of nodes in an MP graph is linearly related to the source data. There is at most one node for each vulnerability instance, state, reachability group, and credential. The maximum number of reachability groups is proportional to the number of interfaces, but is generally much smaller.

The MP graph uses the following three node types:

State nodes represent an attacker's level of access on a particular host. Outbound edges from state nodes point to the prerequisites they are able to provide to an attacker. In Figure 3c, state nodes are circles.

Prerequisite nodes represent either a reachability group or a credential. Outbound edges from prerequisite nodes point to the vulnerability instances that require the prerequisite for successful exploitation. In Figure 3c, prerequisite nodes are rectangles.

Vulnerability instance nodes represent a particular vulnerability on a specific port. Outbound edges from vulnerability instance nodes point to the single state that the attacker can reach by exploiting the vulnerability. In Figure 3c, vulnerability instance nodes are triangles.

These three node types in turn define the sole ordering of paths in the graph: a *state* provides *prerequisites*, which allow exploitation of *vulnerability instances*, which provide more *states* to the attacker.

5.2 Data Structures

This section and Section 5.3 cover the process of constructing MP attack graphs. We first examine the necessary data and how it may be stored in memory. Section 5.3 will then use these data to efficiently construct the MP graph.

The data we store and abbreviations for each are shown in Table 1. Most of the input data can be stored in arrays, and other dynamic structures can be immediately allocated as arrays because their maximum size is tractable and known. We use the notation "X2Y" to indicate a data structure where the key is of type X and the value or values are of type Y. For example, P2N is a mapping from a prerequisite to the unique node in the attack graph representing it. The only data structures we dynamically resize are T2R and R2T, which represent the collapsed reachability matrix discussed in Section 4. Their worst-case size is $O(TI)$, but the actual size of $O(TR)$ is much smaller in practice. A reachability group is only formed and placed in R if the attacker has gained access to it.

Symbol	Name
C	Credential
H	Host
I	Interface
L	Link
N	Node in the MP graph
P	Prerequisite (a reachability group or a credential)
R	Reachability group
S	State (host and access level)
T	Target port
V	Vulnerability instance

Table 1. Data Types Used During Graph Generation

The graph's nodes and edges also need to be represented. The root nodes, representing the attacker's starting locations, are kept in an array because their number is known at the outset. (Root nodes are states, so they are also noted in the S2N structure.) Pointers to all other nodes are kept in V2N, P2N, and S2N, as well as in edge pointers between nodes. All nodes maintain parent and child adjacency lists in balanced search trees, making it easy to traverse the graph and determine if a node has a specific parent or child.

5.3 Graph Construction

The graph is built using a breadth-first technique. No node is explored more than once, and a node only appears on the graph if the attacker can successfully obtain it. The pseudocode for the process is shown in Figure 4. With the exception of line 4, all of the lines in the pseudocode are straightforward. We will discuss line 4 in detail, based on the type of node being considered.

- if `CurNode` is a *state*, then `DestSet` includes all credentials from S2C for that state. If the state's access level is user or root, indicating the attacker has access to the system, then we add all reachability groups from I2R for every interface on the state's host.
- if `CurNode` is a *prerequisite* that is a *reachability group*, then `DestSet` is initially every vulnerability instance that the reachability group can reach. We use R2T to determine all reachable ports, and then use

```
1  BFSQueue starts with the root node(s),
           representing the attacker's
           starting STATE(s)
2  while( BFSQueue is nonempty )
3   CurNode = BFSQueue.dequeue()
4   DestSet = all nodes that can be
             reached from CurNode
5   foreach node DestNode in DestSet
6    add an edge from CurNode to DestNode
7    if DestNode is brand-new,
8     BFSQueue.enqueue( DestNode)
```

Figure 4. Pseudocode for Main Loop

```
1  VulnInst = the vulnerability instance
             we're evaluating
2  DestPort = V2T(VulnInst)
3  if T2R(DestPort) is empty
4   return Failure // there is no known
           // reachability to the port
5  foreach Cred in V2C(VulnInst)
6   if P2N(Cred) is empty
7    return Failure // there is an
       //  unavailable credential
```

Figure 5. Pseudocode for Vulnerability Prerequisite Verification

T2V for each port to determine the vulnerability instances. If there is not yet a node for a given vulnerability (V2N(VulnInst) is empty), we must further verify that all the vulnerability's prerequisites are satisfied before adding them to `DestSet`. Figure 5 contains the pseudocode for the verification step.

- if `CurNode` is a *prerequisite* that is a *credential*, then `DestSet` is initially every vulnerability in C2V for the given credential. If there is not yet a node for a given vulnerability, ie. V2N(VulnInst) is empty, we must perform the same verification step as in the previous case.
- if `CurNode` is a *vulnerability instance*, then `DestSet` is the single state in V2S for the given vulnerability instance.

The most intricate step is the addition of vulnerability instance nodes. The attacker can successfully exploit a vulnerability if it can be reached from *at least one* host the attacker has access to, and if the attacker has obtained *all* credentials required by the vulnerability. The pseudocode in Figure 5 checks to ensure the needed prerequisites are present. If they are, we draw edges to the new vulnerability instance node from all of the reachability groups able to reach it and from all of the credentials used to satisfy it.

An upper bound for the graph's computational complexity can be obtained by observing that the maximum number of nodes is fixed. Assume a network with V vulnerability instances, T ports, C credentials, I interfaces, and R reachability groups. For simplicity, assume $T < V$. The most expensive operation is the transition from reachability group prerequisites to vulnerability instances, costing $O(V+VC)$ time to check every reachable vulnerability and determine if the credentials it requires are present. Over all possible reachability groups, computation is $O(VR+VRC)$. In typical networks where C is small and $R \ll I$, performance is nearly linear in the overall network size. The expected worst-case performance is $O(\max(V,T)RC)$.

6 Automated Graph Analysis

Attack graphs for all but the smallest networks are too large for hand evaluation. We have considered two approaches to this problem: automatic graph simplification and automatic recommendation generation. The former aims to reduce the size of the graph by collapsing similar nodes together. The latter treats the attack graph as an intermediate structure, not a final product, and extracts useful information from the graph for presentation to the user.

6.1 Graph Simplification

Although an MP graph is much smaller than the corresponding full graph, it is still large, and the number of cycles makes it difficult to lay out coherently. We have developed a simple algorithm to "collapse" many graph nodes together, simplifying the visual presentation.

First, state nodes are combined when the prerequisites that can be used to reach them, the prerequisites provided by them, and the access levels match. Vulnerability instance nodes are then combined when the prerequisites necessary to exploit them and the *collapsed* state nodes they provide match. The resulting simplified graph shows the relationships between prerequisites and the quantity of compromise they enable. In Figure 3c, the states C and D would collapse, followed by the vulnerability instances V_C and V_D.

The simplified MP graph can be efficiently derived from the original MP graph. Each pass visits the N nodes and E edges of the graph a constant number of times, and we sort hashes of the matching criteria to discover matches. The resulting complexity is $O(E + N \lg N)$. Other forms of graph simplification may also prove valuable.

6.2 Recommendation Algorithms

Even visually simplified attack graphs can be large and unwieldy. The core information from the graph should be

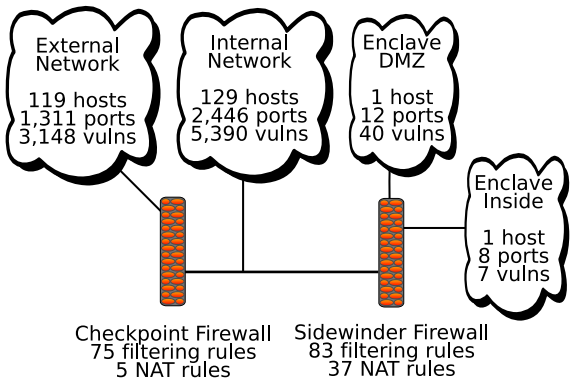

Figure 6. Field Test Network

extracted by the tool and presented to the user in a more immediately useful form.

Often an attacker must compromise a directly-accessible host through a filtering device in order to attack a group of hosts behind the filtering device. Attack graphs can be used to identify these bottlenecks and produce a list of the critical vulnerabilities which allow the attacker to compromise the bottleneck hosts. Defenders can then patch these vulnerabilities first to protect all of the hosts beyond the bottleneck.

We form recommendations by computing, for each individual prerequisite in the graph, which vulnerability instances need to be removed in order to prevent the attacker from reaching the prerequisite, and which states the attacker cannot reach with the prerequisite absent. We accomplish this by rebuilding the MP graph for each potential recommendation, noting which vulnerability instances are actually necessary to reach the selected prerequisite and which states are no longer achievable. Some prerequisites may yield identical recommendations. We discard duplicates.

We weight recommendations based on the number of hosts denied the attacker. A user could supply per-host "asset values" or weights to prioritize steps that protect critical servers. Other weighting metrics, such as the ratio of protected hosts to required patches, may be preferable.

7 Test Results

We have applied NetSPA in one field test deployment and successfully discovered a misconfigured firewall. We have also verified our scaling assumptions by testing against simulated networks.

7.1 Field Test Results

We have tested our prototype on a small operational network, shown in Figure 6. The network has 252 hosts, 3,777 ports, and 8,585 vulnerability instances. No credentials were modeled. The prototype used Nessus scans of

```
To protect 116 hosts, patch
the following 4 vulnerabilities:
  on Host server01.example.gov @ 10.90.0.2,
   on Port 25/tcp:
    CAN-2003-0161: The prescan() function in the ad ...
    CVE-2002-0906: Buffer overflow in Sendmail befo ...
    CAN-2002-1337: Buffer overflow in Sendmail 5.79 ...
   on Port 53/tcp:
    nessus11318: The remote BIND 9 server, accordin ...
```

Figure 8. Recommendation Excerpt for Field Test Network

the four links shown and copies of the rulesets of the two firewalls. The field test results were computed on a laptop with a Pentium-M 1.6Hz processor and 1GB of main memory, running a 2.6 Linux kernel. We have used anonymized hostnames and IP addresses. During normal network operations, a computer from the external network should be able to reach `server01.example.gov` on the internal network only via SMTP.

NetSPA's Perl frontend converted the source data to NetSPA's internal binary format. The firewall rulesets and Nessus scans were automatically read and interpreted. All of the vulnerabilities were read and classified as in [17, 18]. The entire import stage required 24 seconds.

Once converted, the network is read into the C++ stage of the prototype. This stage computes reachability, generates the MP attack graph, computes automated recommendations, and creates the simplified MP attack graph. It writes the two graphs to disk in the DOT language [3] and the recommendations as text. When the attacker is hypothesized on the "external network" segment of the network, the entire time for load, computation, and write was 0.5 seconds.

The resulting MP graph contains 8,901 nodes and 23,315 edges. A total of 12 filtered and four unfiltered reachability groups were formed. In order to evaluate the sixteen potential recommendations, the MP graph was rebuilt *sixteen times* in the 0.5 second runtime.

The simplified MP attack graph, presented in Figure 7, is still too complicated to read and interpret despite an over 99% reduction in size. The graph contains 80 nodes and 190 edges. The attacker's starting location is in the upper right. States are represented by black nodes, prerequisites by dark grey nodes, and vulnerability instances by light gray nodes.

The list of eleven automatically generated recommendations is far more helpful. The recommendation that protects the most hosts is shown in Figure 8.

We expected reachability to the SMTP server on `server01.example.gov` from the external network, but not to the DNS server. We used the NetSPA prototype to automatically generate a complete list of every port an attacker could reach and how an attacker could get there. We discovered a misconfigured firewall rule that allowed exter-

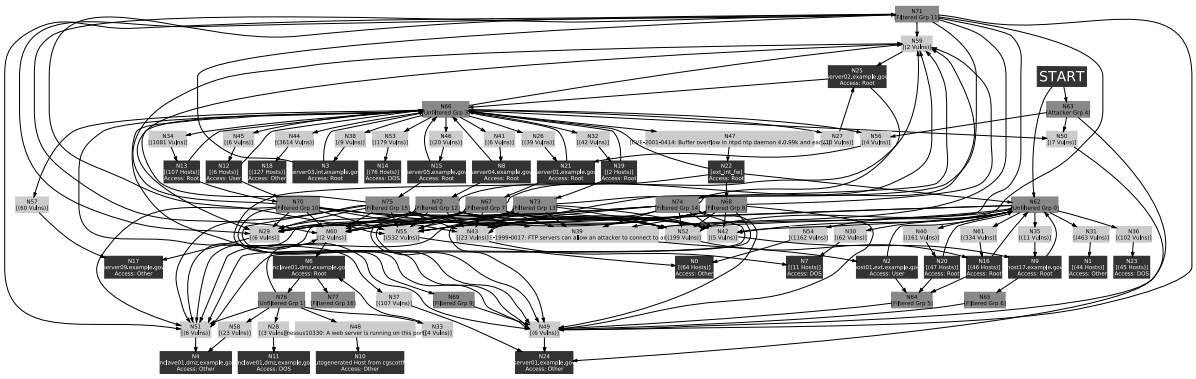

Figure 7. Simplified MP Attack Graph for Field Test Network

nal access to all hosts on the inside network via port 53. The rule was corrected following our analysis. The misconfiguration only permitted access from a few IP addresses that are not normally used. A Nessus scan from the external network to the internal network would not have discovered this, because the scanner would not have used one of the source IP addresses able to cross the firewall.

We also recomputed the results for all possible starting locations at once – placing an attacker with the ability to spoof any source IP on all four links, and also allowing all hosts to initiate attacks. No recommendations are generated in this case. Loading the data, computing reachability and the MP graph, and saving the results consumed 0.54 seconds. The resulting MP graph shows all possible compromises in the network – from all sources, to all targets.

The field trial's results fuel optimism in the prototype and its utility. We are working to conduct additional field trials on larger, more complicated networks.

7.2 Simulation Test Results

We used an automated network generator to explore the scalability of the NetSPA prototype. The generator created a network of three sites. Each site had a fairly generic network structure, with its own border firewall, DMZ, internal administrative LAN, and multiple other internal subnets. Our test setup used 400 filtering rules on each border firewall, six hosts on the DMZ and administrative LANs, and 80 other internal subnets in each site. Each host had 30 open ports. Half had ten remote-to-other vulnerabilities, and the other half had one remote-to-root vulnerability and nine remote-to-other. The available attack path to each site from the outside compromised one DMZ host, then one administrative LAN host, and finally all of the vulnerable hosts on the inside. NetSPA collapsed each site's 80 internal subnets into single unfiltered reachability groups.

Figure 9 shows results with this configuration, varying the number of hosts on the 80 internal subnets of each site. These tests were performed on a Windows Server 2003 machine with dual 3.2GHz Xeon processors and 2GB of main memory. The prototype is single-threaded and never required swap space. The Y axis is the elapsed time, in seconds, and the X axis is the total number of hosts in the network. The plot shows that scaling is linear, as expected, and handles a network with over 50,000 hosts and over 1.5 million ports in under four minutes. NetSPA was able to compute an "all sources, all targets" graph for the 50,000 host network in under twelve minutes.

The effects of different network configurations (via simulation) are examined further in [18]. The results in [18] are based on the use of predictive graphs, however; we expect equal or better performance with MP graphs.

8 Related Work

Some of the earliest work on attack graphs was done by hand. Schneier's *attack trees* [29] were designed to show how multiple attack vectors could compromise a single target. The approach is worthwhile when brainstorming a set of potential attacks and there is a single goal or target. Schneier's example is opening a safe.

Ritchey and Ammann [26] used model checking techniques to find a counterexample to an asserted security policy. Although model checking is more powerful and does not require a monotonicity assumption, it scales very poorly for this application, as noted in [15] and elsewhere.

Others explored the use of full attack graphs [8, 30], as shown in Figure 3a. Full graphs grow combinatorially and cannot be used for large networks.

Ammann [7] developed an algorithm which scales as roughly $O(n^6)$ [19], but is capable of finding all exploits which can be used to reach a specified goal. Jajodia et al. [15] adopt the algorithm and use Nessus scans to identify some vulnerability locations and reachability. The paper proposes the use of Nessus to discern reachability by scanning from every subnet to every other subnet. This approach

may introduce false negatives by neglecting reachability unavailable to machine running the Nessus scanner.

In later work, Ammann et al. [6] presented an $O(n^3)$ algorithm that quickly determines the worst-case attack paths to all compromisable hosts, and argues that such a report is more useful to an analyst or penetration tester than a traditional attack graph. The method may also be applicable to MP graphs in $O(n^3)$ time.

An approach due to Ou et al., called MulVAL [25], uses a monotonic, logic-based approach. MulVAL requires reachability information and can produce counterexamples for a given security policy. The results shown in [25] imply a runtime between $O(n^2)$ and $O(n^3)$.

Other research has focused on the source data required to build attack graphs. Ritchey et al. [27] propose a framework for modeling reachability. NetSPA uses a simpler reachability model that is decoupled from the underlying host's software and vulnerabilities. Templeton and Levitt's [31] prerequisite/postcondition model for attack components and Cuppens and Ortalo's LAMBDA language [14], for example, provide detailed models of vulnerability and attacker action. We are not aware of any readily available vulnerability database populated with the level of detail required by these approaches, nor any similarly detailed scanner. Such tools would be very useful.

Another research focus is tools to explore attack graphs and utilize the information they contain. Noel et al. [24] propose a symbolic equation simplifier to produce recommendations from the graphs of [15]. They also simplified graphs by collapsing related nodes [22] and by transforming the results into an adjacency matrix [23]. NetSPA's simplified MP graphs serve a similar purpose to [22], but improved visualization remains a concern.

Attack graphs may be used to form scenarios and filter IDS alerts. If a series of alerts matches a path in the attack graph, the series is more likely to be genuine. Papers such as [11, 13, 21] explore this use, but no practical application has yet been constructed. Ning identifies the method in [21] as NP-complete. NetSPA does not address this application.

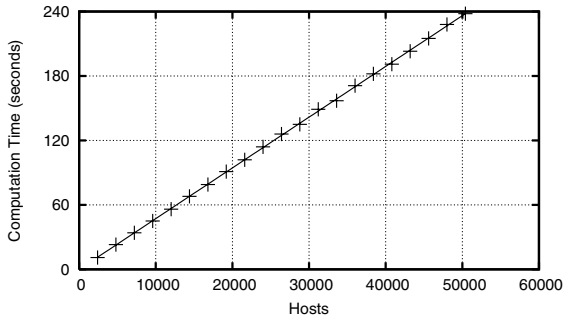

Figure 9. NetSPA Prototype Performance on Simulated Networks

Skybox View [5] is a commercial tool that performs attack graph analysis. The company's patent [12] describes their algorithm, asserts it is $O(n^3)$, and suggests $O(n^2)$ is possible. Based on the patent, we believe that Skybox may build a variant of a *host-compromised graph* [17, 18], and may report only the shortest attack paths to a target.

9 Conclusion and Future Work

Attack graphs are a useful tool in the arsenal of network defenders. Vulnerability scanners such as Nessus report large numbers of vulnerabilities, prioritizing them based on severity in isolation. The amount of work necessary to patch every identified vulnerability is often overwhelming. Attack graphs are able to coalesce a large amount of source data into a useful form, focusing defenders' efforts where it is most needed.

Attack graphs also enable the safe evaluation of what-if scenarios. Defenders can hypothesize new zero-day vulnerabilities on critical services, evaluate the impact of changing filtering rulesets, and determine the effect of adding a new, unpatched computer to various locations in a network. Alterations to the network's defensive posture can be evaluated before they are implemented.

The NetSPA system is able to build a new graph type, the multiple-prerequisite graph, at a very high rate of speed, enabling defenders to quickly evaluate their network's security. The NetSPA prototype can be applied to real operational networks in a straightforward manner, using data that network operators routinely collect. It produces valuable recommendations in seconds, helping defenders filter through thousands of vulnerability reports to find the few vulnerabilities that matter most.

The NetSPA prototype is useful in its current form. It can automatically import readily available source data, quickly compute network reachability, generate the MP graph, and produce useful recommendations. However, additional work remains. The MP graph is capable of supporting credentials as prerequisites to successful attack. However, we have no readily available source of credential data and have conducted no field trials utilizing it. "Client-side" attacks, in which an attacker uses a malicious server to compromise a vulnerable client, are not modeled. The recommendation algorithm does not evaluate the effect of protecting multiple prerequisites. New approaches to graph visualization and simplification may make better use of the graph as well. Future work on obtaining better source data and producing better recommendations and analyses promises to further improve the system's utility.

Acknowledgments

We would like to thank Seth Webster and Doug Stetson for reviewing much of the NetSPA prototype's design, Chris Scott, Kendra Kratkiewicz, Rob Cunningham and Mike Artz for their contributions to previous versions of NetSPA, Carrie Gates for feedback on the paper, and unnamed system administrators who helped us perform the field trial.

References

[1] Bugtraq vulnerability database. `http://www.securityfocus.com/archive/`.

[2] Common vulnerabilities and exposures dictionary. `http://cve.mitre.org`.

[3] Graphviz - graph visualization software. `http://www.graphviz.org`.

[4] Nessus security scanner. `http://www.nessus.org`.

[5] Skybox security, inc. `http://www.skyboxsecurity.com`.

[6] P. Ammann, J. Pamula, R. Ritchey, and J. Street. A host-based approach to network attack chaining analysis. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 72–84. IEEE Computer Society, 2005.

[7] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM Press, 2002.

[8] M. Artz. NETspa, a network security planning architecture. Master's thesis, Massachusetts Institute of Technology, 2002.

[9] D. Bilar. *Quantitative Risk Analysis of Computer Networks*. PhD thesis, Dartmouth College, 2003.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Trans. Comput.*, volume 35, pages 677–691. IEEE Computer Society, 1986.

[11] S. Cheung, U. Lindqvist, et al. Modeling multistep cyber attacks for scenario recognition. In *Proceedings of the Third DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 284–292, 2003.

[12] G. Cohen et al. System and method for risk detection and analysis in a computer network. United States Patent 6,952,779, October 2005.

[13] F. Cuppens. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Washington, DC, 2002.

[14] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 197–216, 2000.

[15] S. Jajodia, S. Noel, and B. O'Berry. *Topological Analysis of Network Attack Vulnerability*, chapter 5. Kluwer Academic Publisher, 2003.

[16] J. Lind-Nielsen et al. BuDDy, a binary decision diagram library. `http://buddy.sourceforge.net/`.

[17] R. P. Lippmann et al. Validating and restoring defense in depth using attack graphs. In *Proceedings of MILCOM 2006*, Washington, DC.

[18] R. P. Lippmann et al. Evaluating and strengthening enterprise network security using attack graphs. Technical report, MIT Lincoln Laboratory, Lexington, MA, 2005. ESC-TR-2005-064.

[19] R. P. Lippmann and K. W. Ingols. An annotated review of past papers on attack graphs. Technical report, MIT Lincoln Laboratory, Lexington, MA, 2005. ESC-TR-2005-054.

[20] P. Meil, T. Grance, et al. NVD national vulnerability database. `http://nvd.nist.gov`.

[21] P. Ning and D. Xu. Learning attack strategies from intrusion alerts. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 200–209, New York, NY, 2003. ACM Press.

[22] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118, New York, NY, USA, 2004. ACM Press.

[23] S. Noel and S. Jajodia. Understanding complex network attack graphs through clustered adjacency matrices. In *Proceedings of the 21st Annual Computer Security Conference (ACSAC)*, pages 160–169, 2005.

[24] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, pages 86–95. IEEE Computer Society, 2003.

[25] X. Ou, S. Govindavajhala, and A. Appel. MulVAL: A logic-based network security analyzer. In *Proceedings of the 14th USENIX Security Symposium*, pages 113–128, 2005.

[26] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 156–165, 2000.

[27] R. Ritchey, B. O'Berry, and S. Noel. Representing TCP/IP connectivity for topological analysis of network security. In *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, NV, 2002.

[28] S. Schechter, J. Jung, W. Stockwell, and C. McLain. Inoculating SSH against address harvesting. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, San Diego, CA, 2006.

[29] B. Schneier. Attack trees. *Dr. Dobb's Journal*, 1999.

[30] L. P. Swiler et al. Computer-attack graph generation tool. In *Proceedings DARPA Information Survivability Conference and Exposition (DISCEX II)*, pages 307–321, Los Alamitos, CA.

[31] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms*, New York, NY, 2001. ACM Press.

[32] A. Wool. A quantitative study of firewall configuration errors. In *IEEE Computer*, pages 62–67, 2004.

[33] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall modeling and ANalysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.