Computer-Attack Graph Generation Tool

Laura P. Swiler, Cynthia Phillips, David Ellis, and Stefan Chakerian
Sandia National Laboratories, Albuquerque, NM 87185
Email: lpswile@sandia.gov, caphill@sandia.gov; dellis@sandia.gov;schake@sandia.gov

Abstract

This paper presents a tool for assessment of security attributes and vulnerabilities in computer networks. The tool generates attack graphs[10]. Each node in the attack graph represents a possible attack state. Edges represent a change of state caused by a single action taken by the attacker or unwitting assistant, and are weighted by some metric (such as attacker effort or time to succeed). Generation of the attack graph requires algorithms that match information about attack requirements (specified in attack templates) to information about the network configuration and assumed attacker capabilities (attacker profile). The set of near-optimal shortest paths indicates the most exploitable components of the system configuration.

This paper presents the status of the tool and discusses implementation issues, especially focusing on the data input needs and methods for eliminating redundant paths and nodes in the graph.[1]

1. Introduction

Over the past decade, people have become more aware of the need for information security as viruses, attempted intrusions, and actual attacks have become more frequent and widespread. Industry and government are vulnerable to attack through their reliance on information technologies. Academia and computer companies have responded to the security problem by developing network-level software (e.g. stateful routers) and application-level software (e.g. intrusion detection, firewalls) to identify threats and vulnerabilities and protect information assets. The current approach to information assurance is often driven by checklists and compliance standards, and not by an overall understanding of the requirements of the system and the components chosen to implement those requirements within a required risk level. An example of a risk metric is the evaluation assurance level described in the Common Criteria [2].

Our goal is to create a design and assessment tool for qualitative and quantitative assessment of vulnerabilities such as those that arise as a result of integrating diverse sets of COTS components in a network. Our tool will go beyond the current generation of vulnerability-assessment tools to examine a network-security state from a system-level perspective (note: here we are referring to system at an enterprise level, such as the network of a business enterprise).

The attack graph tool is based on a rigorous methodology, which takes requirements necessary for a system state change and "matches" these requirements to information about the network configuration and the potential attacker to create an *attack graph*. Each node in the graph represents a possible system state during the execution of an attack on network components. System state includes level of penetration by the attacker (such as successful access of a web page or acquisition of root privileges) and configuration/state changes (such as modification of access control or placement of Trojan horses) achieved on specific physical machine(s). After post-processing, nodes contain the summation of the security metrics thus far (e.g. minimum time to achieve this state). Edges are directed arcs that connect one system state to the next. Edges represent a change of state caused by a single step. Templates list the required conditions for state transitions.

The graph is generated by matching the current (node) state against a library of templates, choosing only the templates that apply to the current state. Paths in the graph represent the dynamics of attack

[1] This research was funded in part by DARPA's Information Assurance Science and Engineering Tools (IASET) program, Contract number 062980217-3. This work was supported in part by the United States Department of Energy under contract DE-AC04-94AL85000.

0-7695-1212-7/01 $10.00 © 2001 IEEE

accomplishment, exposing the causality and relationships between parts of an entire system. Any path in the graph represents a unique attack path, though it could be a series of steps combined from many known attacks.

The quantitative metric assigned to each edge represents the cost of making the state transition. Currently we assume that the quantitative metrics are linear and cumulative in some sense. A path from a start node to a goal node has a cost equal to the sum of the costs of the edges in the path. The graph can be used as a basis for system comparison. We assume the set of nearly shortest paths represent the most likely or lowest cost paths for the metric of interest.

A major innovation of our method over other vulnerability-analysis methods is that it considers the physical network topology in conjunction with the set of attacks in a dynamic fashion. The method is flexible, allowing analysis of attacks from both outside and inside the network. It can analyze risks to a specific network asset, or examine the universe of possible consequences following a successful attack. Network configuration/monitoring/scanning tools take a snapshot of the configuration and vulnerabilities of the enterprise components at one time, but the attack graph method allows for changing configurations and the evolution of the attacker's penetration and the degradation of defenses over time. The dynamic aspect of the graph gives unique insight into the behavior of the enterprise. We envision that our tool will work in conjunction with network tools to define an initial system state. Scanners are extremely powerful now, but they do not verify that all conditions for a complete attack are met or identify linked attacks potentially more harmful than individual attacks. Though they can suggest fixes for local potential problems, they don't consider the network as a whole, proposing a global set of cost-effective defenses designed to protect the network's most critical resources. Thus, our graph-based tool goes beyond currently available scanning tools.

A high level conceptual diagram of the attack graph tool is presented in Figure 1. The graph generator matches information about the enterprise network configuration to generic exploit information, generating paths that are feasible given the current state of the system. The resulting attack graph, greatly simplified in this picture, shows an attacker starting as a normal user on Workstation 1 (WS1) and using various exploits to access a file server. Some of the paths involve the attacker becoming root on WS1, and others involve the attacker gaining normal privilege levels on two PCs in the network.

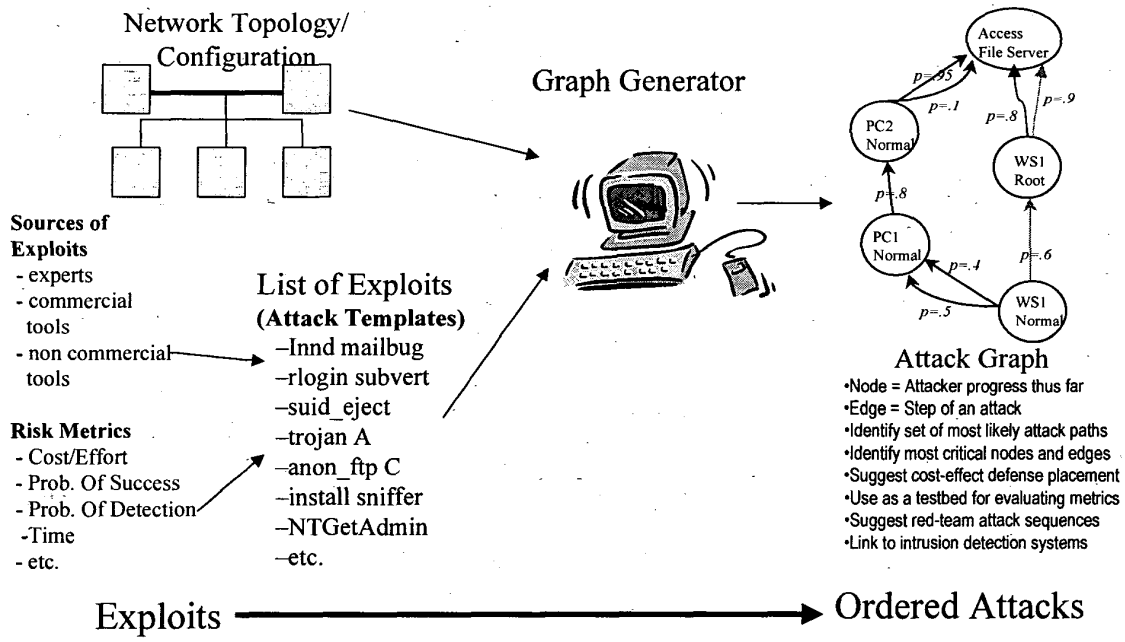

Figure 1. Conceptual Overview of Attack Graph Tool

2. Attack Graph Methodology

2.1 Background and Prior Work

Ideally, a tool that assesses security in network design and actual mission execution should be able to model the dynamic aspects of the network, current and induced security configurations of components, multiple levels of attacker ability, multiple simultaneous events or multiple attacks, user access controls, and time-dependent, ordered sequences of attacks. We originally tried to apply *Probabilistic Risk Assessment* (PRA) techniques to the problem of network vulnerabilities. PRA techniques such as fault-tree and event-tree analysis provide systematic methods for examining how individual faults can either propagate into or be exploited to cause unwanted effects on systems. These methods, however, have limited effectiveness in the analysis of computer networks because they cannot model multiple attacker attempts, time dependencies, or access controls. In addition, fault trees don't model cycles (such as an attacker starting at one machine, hopping to two others, returning to the original host, and starting in another direction at a higher privilege level). Methods such as influence diagrams and event trees suffer from the same limitations as fault trees.

This is not the first system to represent attacks graphically. For example, Meadows [5] uses a graph representation to model stages of attacks, particularly attacks on cryptographic protocols. These visual representations resemble attack templates, but nodes in her graphs represent stages of the attack (a to-do list for an attacker) at a much higher level than our system. Edges represent temporal dependencies. There is no tie-in to particular user level, machine, configuration, etc, and there are no weights. Meadows describes previous work that also breaks attacks into atomic steps.

Moskowitz and Kang [6] use a graph to represent *insecurity flow*. Edges represent penetration of a security barrier such as a firewall. Each edge is weighted with the probability of successfully breaching the defense. They want to compute the probability that *any* path exists from the source to the sink (i.e. any hole exists in the system). They give an exponential-time algorithm to determine a set of edge-disjoint paths that correspond to "reasonable" attacks.

Our system is closely related to that of Dacier et al. [4] although these systems were developed independently. Dacier, et al. propose using a "privilege graph" to represent complex attacks with a single edge. The privilege graph does not explicitly represent attacker capabilities and is based mainly on the acquisition of "privileges" of the user (e.g., the ability to read, write, and modify certain files) whereas the attack graph encapsulates a much richer definition of "state" of a node including changes made by the attacker to the configuration, capabilities acquired by the attacker thus far, etc. Dacier et. al. transform the privilege graph into a Markov model and determine the estimated mean time and effort to target by enumerating all exponentially-many searches in the privilege graph. The Markov model represents all possible probing sequences of a non-omniscient attacker. Ortalo et al. [8,9], present experimental results using this model, based on a privilege graph constructed from 13 major UNIX vulnerabilities. They conclude that Mean Effort to Failure (METF) is more valuable as a security metric than the single shortest path or raw number of paths to target. However, they were not always able to compute METF, even for fairly small graphs.

We can compute all near-optimal shortest paths much more efficiently than the enumeration required to compute METF. We believe that the set of near-optimal shortest paths provides a good measure of overall system security. We expect the set of paths to evolve appropriately with underlying changes in the system, but not to be unduly volatile. In addition, by modeling at a finer level, we can potentially discover new attacks, have more confidence in our cost metrics for common operations, and provide more informative output for system administrators with limited security experience. Our method is more comprehensive since it can model time dependencies and multi-prong attacks. Our edge costs are more customized to a particular network and attacker.

We first described this attack graph concept in [10]. We had done no implementation at that time, nor had we tackled any data issues.

2.2 Attack Graph Concepts

The attack graph is automatically generated given three types of input: attack templates, a configuration file, and an attacker profile. *Attack templates* represent a generic attack step including necessary and acquired state attributes (e.g., capabilities and/or vulnerabilities). The *configuration file* contains initial architectural information about the specific system to be analyzed including the topology of the network and

detailed configurations of particular network elements such as workstations, servers, or routers. The configuration files also include categorical labels for all known exploitable configurations (e.g. blank administrator password) and are used as triggers for state changes. The *attacker profile* contains categorical information about the assumed attacker's capabilities used for matching the templates, for example, the possession of an automated toolkit or a sniffer. The attack graph is a customization of the generic attack templates to the attacker profile and the network specified in the configuration file. Though attack templates represent pieces of known attack paths or hypothesized methods of moving from one state to another, their combinations can lead to descriptions of new attack paths. That is, any path in the graph represents an aggregate attack, though it could be constructed from many smaller attack steps.

More specifically, each node contains a set of *overwrites* to the initial configuration. Overwrites represent additional vulnerabilities or changes to system attributes that are obtained as attack steps occur. Thus the state of the system represented at a node is exactly the initial configuration *except for* the changes explicitly written in the node. Each node overwrite has a set of machines and vulnerabilities interpreted as "*all* of the vulnerabilities are present on *one* of the machines." (Note: for the purposes of this discussion, we are using the term vulnerability but more generally this means any system attribute. For a further discussion of attributes and vulnerabilities, see Section 3.2.1.) For example, Figure 2 shows a portion of an attack graph, with nodes depicted by circles[2]. In Figure 2, the state of the left node is: either machine M_1 or M_2 has vulnerability v_1, and either M_1 or M_2 has vulnerability v_2, and either M_3 or M_4 has vulnerability v_3. If the first two overwrites were combined to $M_1,M_2{:}v_1,v_2$, it would mean either M_1 or M_2 has both v_1 and v_2. This would rule out M_1 having v_1 and M_2 having v_2 for example. Vulnerabilities present in the initial configuration are not included in node overwrites: they are in the configuration files. In this example, initial vulnerabilities include v_5 on M_1 and v_3 on M_5.

[2] Note: In a real attack graph, the edges would be labeled by a weight, described later in this section; and a description of the attack (e.g. "sniff password")

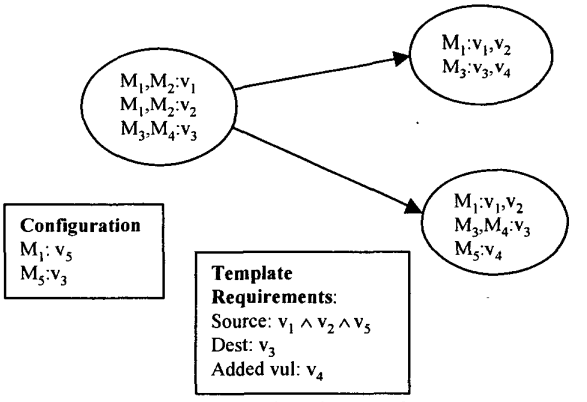

Figure 2. Example of Template Firing

To increase matching speed and space efficiency, the tool distinguishes between *bit* vulnerabilities and *ranked* vulnerabilities. Bit vulnerabilities answer yes/no questions such as "Does the machine have a telnet daemon running?" Ranked vulnerabilities have a state from a totally-ordered set, such as a hierarchy of user privileges. The hierarchy can depend on *fundamental classification* information (currently hardware and operating system type). In particular, there are ranked vulnerabilities associated with operating system and hardware version numbers. Tests for ranked vulnerabilities can involve more complex comparisons: "Is the operating system version greater than 2.3 and less than 5.6b?"

An attack template consists of the conditions or requirements necessary for a state transition, the edge weight (an additive, quantitative metric being evaluated), and the additional capabilities and new exploitable configurations present after a successful transition. Requirements are sets of tests as described above for bit and ranked vulnerabilities, combined with arbitrary logic (e.g. Test1 AND (Test2 OR (NOT Test3))). Each test is tagged as either a source or a destination requirement. Destination machines acquire the new vulnerabilities. Source machines support the attack step and must satisfy requirements, but don't acquire new vulnerabilities. Templates may have no source requirements and it is acceptable for a machine to serve as both source and destination.

When a current node in the graph matches all of the requirements of the template, the edge transition is triggered, and one or more new nodes are created in the attack graph and/or new edges are created to existing nodes. The new node contains vulnerabilities acquired from the attacker action, and possibly a narrowing of the flexibility represented in the originating state. For example, in Figure 2, only

machine M_1 satisfies the source requirements, but only if it is the single machine that has vulnerabilities v_1 and v_2 in the first two overwrites. The new states after the application of the template reflect this requirement. Machines M_3 and M_5 satisfy the destination requirements. If M_3 is the destination, M_4 cannot be the machine with vulnerability v_3, but if M_5 is the destination, the final overwrite in the original state isn't affected. Therefore, this template application produces two new nodes: the top one with destination M_3 and the bottom one with destination M_5.

Each edge has a weight representing one (user-defined) system-security metric, such as success probability, average time to succeed, or a cost/effort level for an attacker. This weight is a function of configuration and attacker profile. Paths which have low weight, meaning they are low cost for the attacker, take a short time to succeed, etc., are considered the most exploitable paths.

Since edge weights may only be estimates, we consider the set of all ε-optimal paths, defined as all paths with length at most (1+ε) times the shortest path length, where ε is any non-negative number. Epsilon must be large enough to account for uncertainty in individual edge metrics and uncertainty in the actual path the attacker will choose. For example, if edge weights can be wrong by a factor of 2 on average, than ε should be at least one so that the ε-optimal paths are all of those within (1+ε = 1+1 = 2) times the length of a shortest path. The ε-optimal paths represent the set of paths that the attacker might realistically consider: an attacker will not necessarily choose the shortest path for reasons of incomplete knowledge about a network, personal preferences for using certain attacks or toolkits, etc. Also, the weights may be inaccurate and thus the shortest path may not represent the best path for the metric. However, it is likely that an attacker would choose a path in the ε-optimal set. Thus, we designate the set of "ε-optimal paths" as high-risk attack paths. In addition, we can allow multiple weights on each edge. Multiple edge weights allow us to represent conflicting criteria (e.g. the attacker wishes to minimize both effort and probability of detection). In this case, we can use a single weight that is a function of both criteria, or optimize one metric (usually cost/effort) subject to a threshold constraint on the other metric (risk-aversion/safety).

We use Naor and Brutlag's algorithm [7] to compute ε-optimal paths and to compute the number of ε-optimal paths each edge participates in. The nodes and edges of the attack graph that appear most frequently in the set of ε-optimal paths are identified as critical nodes and edges, and can be used in subsequent analysis to suggest defense placement. Computing the edges involved in at least one ε-optimal path is dominated by the complexity of Dijkstra's algorithm. Theoretically, this has complexity $O(e + v \log v)$, where e is the number of edges and v is the number of nodes. However, our simpler implementation based on standard heaps has a complexity of $O(e \log v)$. Computing the number of ε-optimal paths for each node has pseudopolynomial-time complexity, since we have to compute the number of paths of each length (up to (1 + ε) times the shortest path). Actually, we compute the deviation from optimal, up to the maximum deviation of ε times the shortest path. Therefore the complexity of that piece is $O(v \bullet \varepsilon \bullet SP)$ where v is the number of edges, ε is the error parameter, and SP is the length of the shortest path.

This system can answer "what-if" questions regarding security effects of configuration changes such as topology changes (adding barriers to create security environments) or installation of intrusion-detection systems. It can indicate which attacks are possible only from highly skilled attackers, and which can be achieved with lower skill levels. A business owner might decide it is acceptable to allow a relatively high probability of network penetration by a "national-scale" effort, but will tolerate only a small probability of attack from an "average" attacker. Government sites, which may be attacked with much higher frequency, may need an extraordinarily low probability of attacker success in order to expect few penetrations, and they may be more willing to pay the cost for the level of security required. The attack graph tool can be used at various phases of a security analysis, from system design, to implementation of security lock-outs and warning systems, to analysis of detection and response options, and security policies.

3. Implementation issues

We developed a simple prototype tool to generate attack graphs. The prototype demonstrates the principles of the attack-graph methodology. However, we have only tested this early tool on extremely small cases, and it currently only generates graphs forward from a start node (vs. backward from a goal node). Our current research addresses many of the issues that must be solved for the tool to become robust, scalable, and able to generate and analyze the extremely large attack graphs that will be created for enterprise-level networks. Some of these issues include scalability of

the graph, decomposition and aggregation of nodes, extensibility of the tool, and visualization techniques required so that the viewer can quickly and easily comprehend the results of a large analysis. In this section, we discuss issues associated with actually generating the graph and issues associated with inputting large amounts of data about the network. First, we present the current architecture of our tool.

3.1 Functional Architecture

The functional architecture of the attack graph tool is presented in Figure 3. Information from configuration management tools and vulnerability scanning tools populates the configuration files. Additional configuration information may be gathered manually to obtain data that a scanner cannot deduce, for example, some types of access controls or corporate security practices. The monitoring tools also provide information on vulnerabilities to the template library developer. The graph generator matches information in the network configuration against the library of templates and the attacker profile, choosing only the templates that apply to the current state. The graph is generated by iteratively performing this matching operation until no more template matches apply. The user can specify a stop/goal node. The generator doesn't apply templates to such nodes. Then, the shortest path analysis code is applied to identify the most critical paths in the attack graph from the start node to a goal node. Finally, a GUI with graph drawing capabilities is used to present the analysis.

In our current prototype, we have implemented all of the boxes in Figure 3 with the exception of the attacker profile. Currently, we can put any required attacker profile information in the configuration files with the machine data, since the graph generator treats attacker requirements in the same manner as it treats machine requirements. Also, we have not fully implemented the importing of data from commercial tools (see next Section 3.2 below). The graph generator, analysis, and visualization codes are implemented in C++ on Unix/Linux workstations/PCs. Because we use a file interface, the scanners and configuration tools can run on any platform.

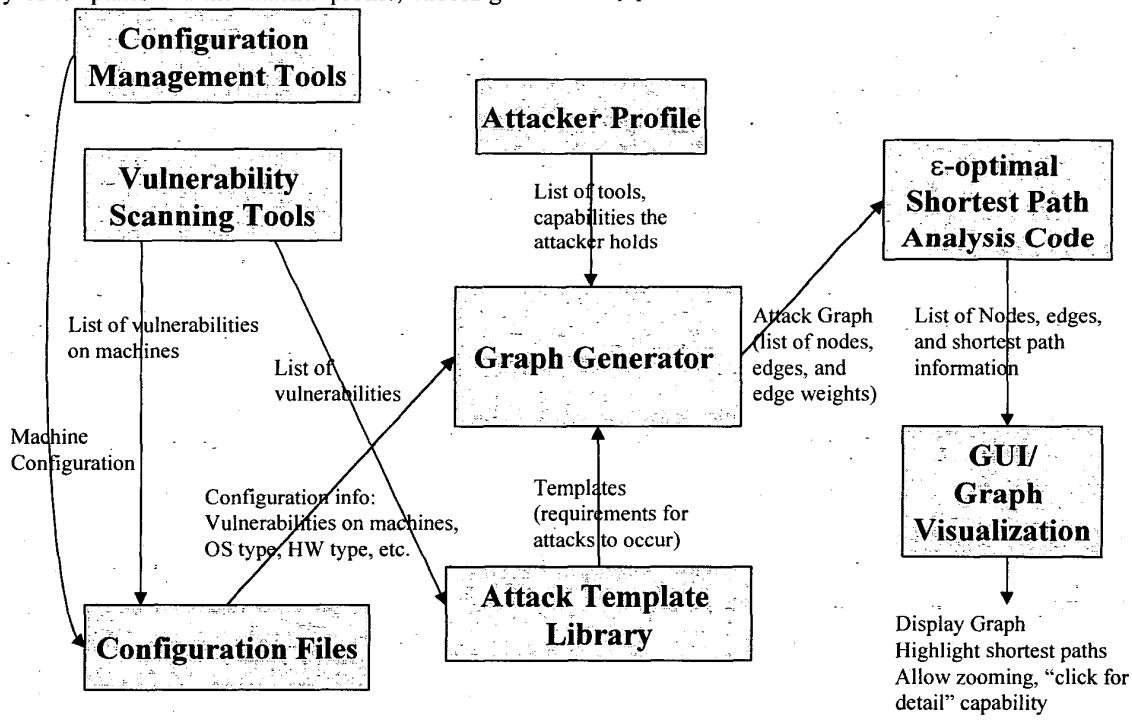

Figure 3. Functional Architecture of the Attack Graph

3.2 Data Issues

For the tool to be useful in real applications, it should link with commercially-available network monitoring/scanning and configuration tools to obtain information about the network. This section explains how we link data from multiple sources, including commercial tools and security experts, and create the configuration files and attack-template files. A more detailed view of the data input and translation piece is outlined in Figure 4 below.

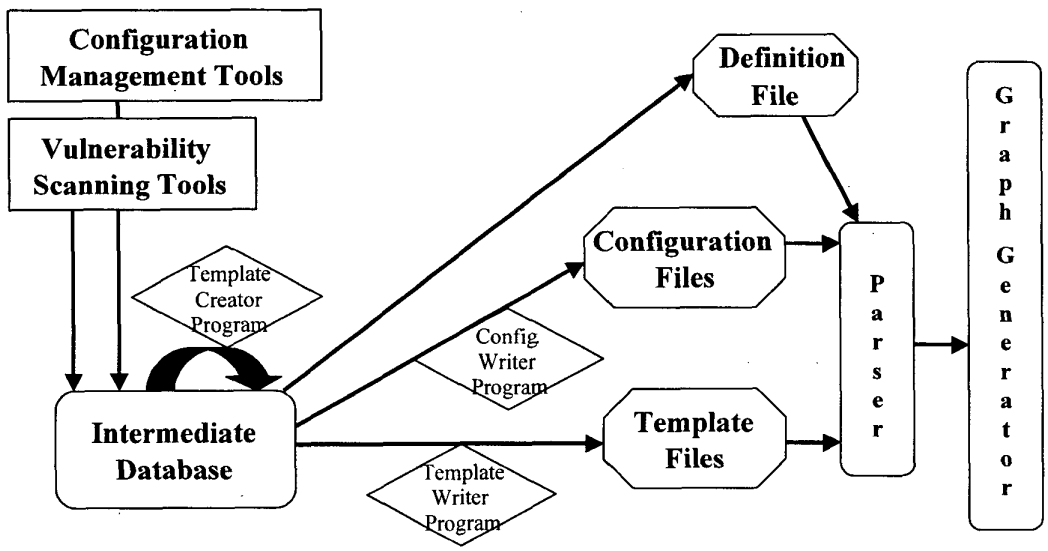

Figure 4. Overview of Information Flow to Graph Generator

3.2.1 Intermediate Database

Since commercial tools primarily use databases to store results, and since the attack-graph tool needs a large amount of network and vulnerability data, we have created an "intermediate" database. The database is a repository of computer security/attack information. This intermediate database contains a representation of the attack template model and copies of relevant fields from vulnerability and configuration management tools. Programs use this database to create the templates and configuration files. It currently is a relational database implemented in Microsoft Access, but it could more generally be any SQL database as data elements increase.

We populate this database with information from commercial tools by mapping data to standardized labels for each vulnerability, security attribute, or other state descriptor. For example, the services table in the ISS System Scanner calls port 20 udp or tcp "ftp-data." This could map to "ftp-dataport" in the intermediate database. Another scanning tool may have a different term which would also map to "ftp-dataport." The standard naming scheme ensures consistency of terms when the templates are matched against configuration information in the graph generation.

A state change is a modification of any attribute of the enterprise system. An attack template has a specific form to ensure all the necessary attributes to conduct a state change are present. Each incremental state change builds a chain of events ("attack steps") which lead to or define a vulnerability. For clarity and simplicity of exposition, our description of the graph generator uses the generic term "vulnerability" to refer to a set of attribute changes that ultimately allow the exploitation of the system. This could involve the compromise of the privacy, integrity, or availability of data and/or computer resources.

The intermediate database structure models the steps necessary to make an attribute change. Our database organization is based on the security reference monitor[1]. At the highest level there are five attribute groupings: subjects, permissions, actions, objects, and methods. We describe a state transition by the following: "Subject" has "permission" to "act" upon "object" via "method." For example, "Normal user" has "permission" to "modify" a "registry key" via "network access." Translating this to an attack template, there are two requirements: the attacker must

be a normal user and he must have "network access." The modified attribute is the new registry key value. The modified registry key may be necessary for a state change later in the attack sequence.

When data collection methods mature and the tools become more robust, we intend to expand these categories. For example "permission" could become part of the attack-template requirements involving tests of access controls on files and a more refined check on attacker privilege.

The database is also a repository for specific configuration data of the particular devices on the network under analysis. Each table in the database contains some information relating to a system "state" (the active processes, exploitable or interesting data objects, current network status, etc.) At a high level, network configuration includes:

- Classification properties, such as operating system types, network connectivity types (e.g., Ethernet, ATM, etc.)
- Dynamic subjects, such as processes running: user shell, operating system daemons such as ftp, etc.
- Data objects, which are acted upon by subjects.
- Subject-to-object processing paths.

Classification properties include type of hardware, type and version of operating system, display devices, storage devices, etc. These properties tend to classify the subjects and the objects of interest into categories. For example, Windows NT has a defined privilege structure for subjects and access control list for objects. Objects include files, directories, registries, etc. At the most detailed level, objects have attributes which define security characteristics. For example, files have access permissions, processes have privilege levels necessary for execution, etc.

Figure 5 shows how the attack templates and configuration files map to tables in the intermediate database. These tables contain attributes relating to the areas defined by the security reference monitor. The attack templates may be at a fairly high level, but will be decomposed to a level of the tables containing "subject", "permission", "action", "object", and/or "method" information. The configuration files may be at a more detailed level, but the properties of specific devices will be mapped up to terms of the security reference monitor. This "meeting in the middle" allows the templates to match the configuration information at the same level of decomposition detail.

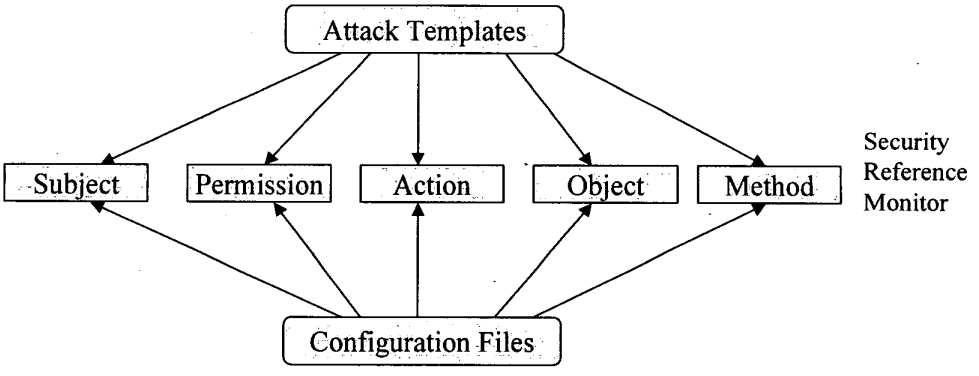

Figure 5. Mapping of Templates and Configuration Files to Tables in the Intermediate Database

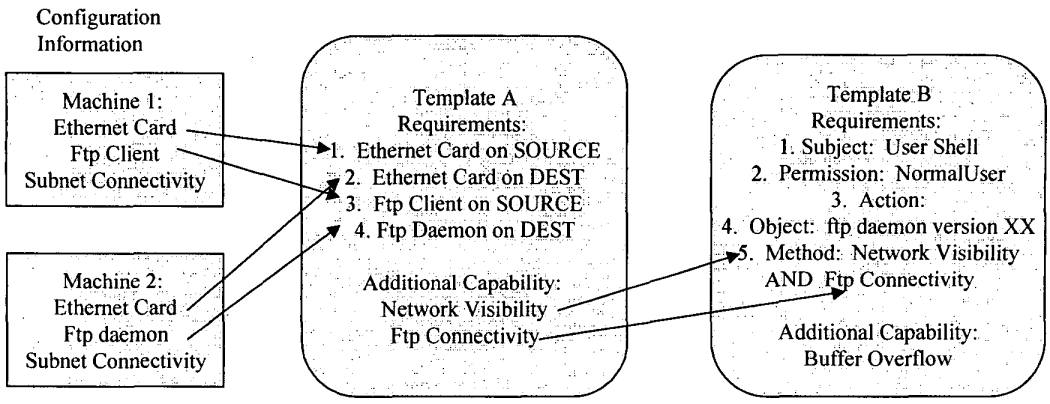

Figure 6. Example of Helper Template Requirements

With a little work on the part of the attack template library developer, a user can define templates at whatever level of abstraction he desires. Figure 6 shows an example where the configuration data is at a lower level of abstraction than the templates. For example, Template A specifies how an attacker can obtain information about a network. Many templates will require network visibility and/or ftp connectivity as one of the necessary conditions for the template to trigger. The helper or definitional template A defines the concept of network visibility. This "helper" checks the source and destination machine configurations to ensure that they have Ethernet cards and subnet connectivity and if so, turns on the "network visbility" attribute, thus allowing the graph generator to recognize this higher-level concept. Figure 6 shows how helper Template A checks the requirements for network visibility and ftp connectivity as a precursor for a template that uses these conditions (Template B). Template B is defined at the security reference monitor level, with requirements on source, permission, object, etc. It represents the conditions under which an attacker can use ftp and network connectivity to create a buffer overflow on the destination machine.

3.2.2 Data Defaults/Unknowns

A criticism of a data-intensive tool such as the attack graph generator is that one will never be able to gather all of the required input data. We counter this with the ability to leave configuration information "unknown" and resolve it with a two-level default system. Each attribute has a default value. For example, the access control default may be read and write. Whenever the graph generator encounters an unknown value in the configuration or attacker profile, it uses the default value for the attribute it is testing. An attribute default can also be "unknown". In this case, the test falls through to a global yes/no default. This default determines the outcome of the test rather than the value of the attribute match and is therefore suitable for both individual attributes and classification properties. If one of the unknown values is along a critical path, we can flag that edge so the user can gather more information about this value. He can then rerun the tool to generate a graph that more closely models true system vulnerabilities.

3.2.3 Template Creator Program

We have written a Visual C++ program to help the user populate the database with attack information hiding details of the database representation, while ensuring consistency and completeness. We have implemented this with a Graphical User Interface (GUI) that prompts the user for information related to a template using a series of menu-driven forms. For example, there is a "Template Addition" form where the user inputs the Template Name, the Edge Weight, and the requirements necessary for the template to "fire." When the user gets to the form for adding requirements, there will be a drop-down menu listing all of the attributes, services, files, etc. that are in the intermediate database. The user then only has to select the requirements by highlighting them. If particular requirements depend upon classification properties such as operating system or if the requirements are not yet in the database, then the program prompts the template developer for the required information as necessary.

3.2.4 Template Writer Program

Building and maintaining the template and configuration files by hand would be a tedious, time-

consuming and error-prone task which could seriously limit the utility of the tool. Thus, we have developed a "Template Writer" program to facilitate template development. Our template writer program, written in Visual C++, uses a series of customized database calls to read the tables in the intermediate database and write that information to template files. We plan to create a similar program to automatically write configuration files from data in the intermediate database.

3.2.5 Parser

The template creator helps with the editing and consistent creation of the template files and machine configuration files. However, the graph generator needs appropriately-filled data structures prior to creating the graph. For this, we use a parser as an intermediate step between the textual files and the graph generation code. The parser fills in the data structures, converts string variables for ranked-vulnerability hierarchies to appropriate values, fills the vulnerability lists and builds parse trees representing template requirements for the graph generation process. It is the glue between the machine configuration files and attack templates, and graph generator structures. Note that the ranked-vulnerability levels (such as five levels of user) are specified in another file that is input to the parser, the definition file in Figure 4. In addition to templates from the template writer program, the parser can take information from other sources, provided they use our template language. Examples include configuration files generated from various sources and "massaged" into the appropriate configurations, and templates created from other sources. Users have the option of using a simple text editor to create customized templates and configurations. Also, a user may write scripts to gather information of interest to them, and use this information to populate configuration files.

3.3 Graph Generation Issues

There are many difficulties associated with graph generation. One of the most critical is eliminating redundant paths and nodes. A related issue is scalability: ensuring that we can generate and analyze very large graphs.

3.3.1 Redundant path elimination

In our current implementation, a node contains information about all of the changes to system state that have been acquired since the "baseline" or initial system configuration. For example, in Figure 7, we show a simple graph representing the acquisition of vulnerabilities (more generally, vulnerability refers to a state attribute) during an attack on a single machine, M1. Vulnerabilities are represented by letters, and the two letters on each edge represent the required vulnerability to make the transition and the resulting attribute or capability gained when the transition is made. For example, the graph starts with the presence of Vulnerability A on machine 1. Then, there are three paths that require A and add vulnerabilities C, B, or D. From the node where vulnerability C has been added, we can still add B or D, etc. The graph shows all of the paths that lead to the placement of vulnerabilities B, C, and D on machine 1.

As demonstrated in Figure 7, there are often many paths to a node that involve acquiring multiple, independent vulnerabilities. In this example, the path involving acquiring the attributes in the order C-B-D is essentially the same as the path B-C-D or D-B-C: the same attributes are acquired in the same way, only in a different order. The elimination of redundant paths in the example in Figure 7 produces the attack graph depicted below in Figure 8.

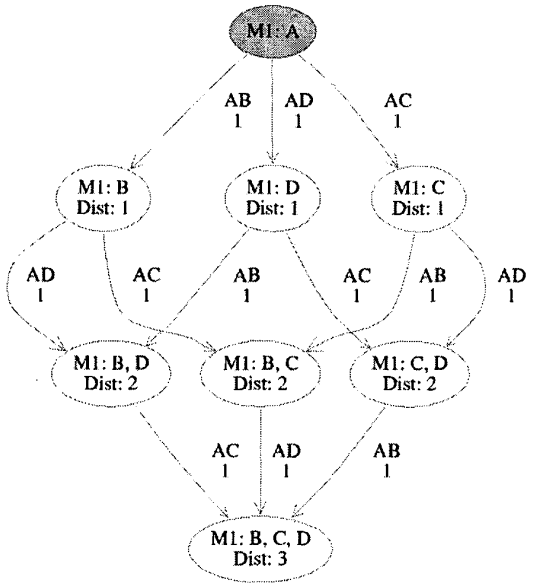

Figure 7. Graph Generation

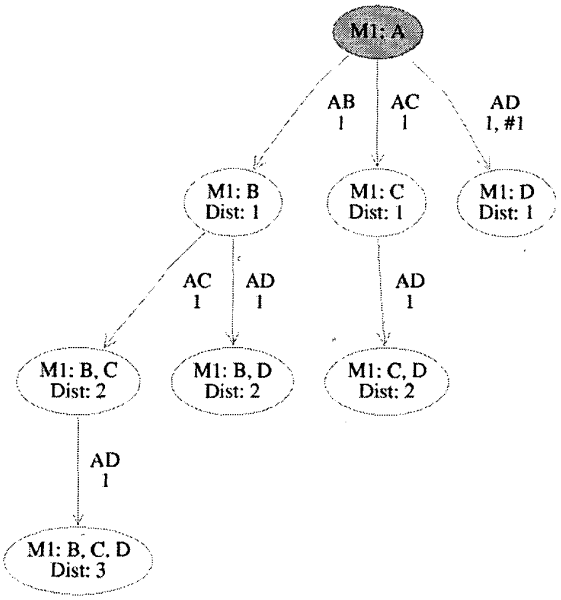

Figure 8. Graph Generation with Path Redundancy Elimination

We have implemented the elimination of such "ordered paths", that is, paths that involve the acquisition of the same vulnerabilities using the same set of templates but in different order, by enforcing a partial ordering for vulnerability acquisition. If one needs both vulnerabilities A and B for an attack, the system will generate an attack path where the attacker acquires A first and then B. This is not the same as acquiring B and then A, but clearly the system should not and cannot generate all *n*! paths to a node with *n* vulnerabilities. If the cost of acquiring A then B differs from the cost of acquiring B then A, we generate both paths: they are not redundant. However, each such non-redundant pair implies separate templates so graph complexity reflects the size and complexity of the attack template library, as a user would expect.

We enforce the partial order by giving each vulnerability a *rank*. A vulnerability precedes another (and must be acquired first) if it has a lower rank. Two vulnerabilities with the same rank are incomparable, and therefore we generate both orders. We compute the ranks as follows. Suppose vulnerability j is an *ancestor* of k. That is, vulnerability j can satisfy a requirement in a template that adds vulnerability k. The partial order must respect this ancestor relationship (it must be possible for a path to acquire vulnerability j before vulnerability k). We create an ancestor graph that has a node for each vulnerability and an edge from node j to node k if vulnerability j is an ancestor of vulnerability k. Nodes in the same strongly connected component share a cycle (each can be used to get the others) and therefore all have the same rank. We collapse these components to single nodes, compute a topological sort of the resulting acyclic graph, and give each node its rank in this topological sort (where nodes in collapsed strongly-connected components inherit the component's rank). We can compute the strongly-connected components and topological sorts in linear time.[3]

3.3.2 Redundant Node Elimination

We are currently designing a system to prohibit the generation of nodes that involve subsets of independent vulnerabilities. In the above example, with n independent attack steps, there are 2^n nodes that would be generated without this suppression, which is mathematically intractable. Furthermore, we can convey the essential information much more succinctly. Therefore, the forward (exploratory) generation will only generate "interesting" paths that lead to new vulnerabilities on each step. For example, Figure 9 shows the example of Figure 7 with node-redundancy elimination and an additional template that requires vulnerabilities B, C, and D to produce vulnerability E. Set {B,C,D} is interesting since it can produce vulnerability E. Because of the ordering restrictions described in section 3.2.1, we generate only the subsets

of {B,C,D} that obey the ordering constraints. Because node overwrites reflect only changes to the initial configuration, "interesting" subsets will depend upon the initial configuration in general. For example, in Figure 9, if machine M1 has vulnerability C initially, then none of the nodes showing vulnerability C would be generated, but nodes with sets {B,D} and {B,D,E} would be. Testing for interesting sets becomes more challenging when there are many ways to generate a vulnerability, many ways to generate its ancestor vulnerabilities and their ancestors and so on. It's possible that obeying ordering constraints will lead to graphs that show the generation of vulnerabilities in nonintuitive orderings. For example, two vulnerabilities directly used to get a third may not be acquired consecutively because the attack will ultimately need a vulnerability that has a rank between them.

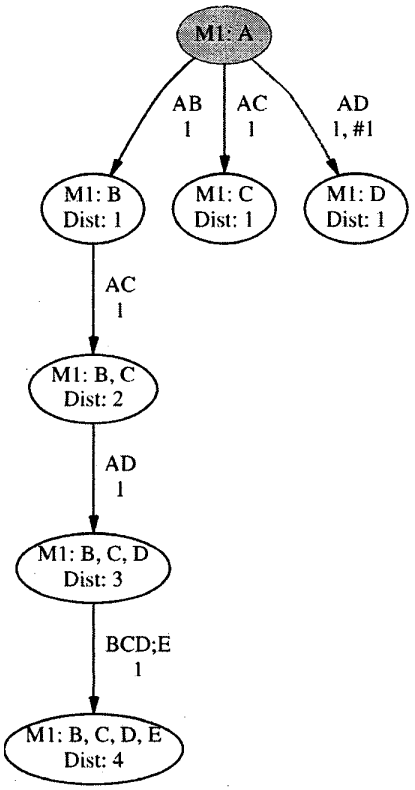

Figure 9. Graph Generation with Node Redundancy Elimination

After a forward (exploratory) search, the tool will present the user with a menu of vulnerabilities that can each be generated individually. For example, an attacker could compromise a web server or a file server. If there is no vulnerability that depends upon both these vulnerabilities, the graph will not contain a node with their combination. It's possible the attacker cannot achieve both because each attack uses the same one-time vulnerability. It's possible the attacker can achieve both with less effort than the sum of the two individual attacks because they use common subattacks. The user can specify particular subsets of independent vulnerabilities as interesting sets at the start of an exploratory search, or he can select a set from the menu afterward. Then the tool will generate backward from this new goal node, directly generating the paths to this set of vulnerabilities. Note: backward generation is still unimplemented.

3.3.3 Directed Cycle Elimination

The attack graph should have no directed cycles, since it is never to an attacker's advantage to perform steps that simply return to a state he has already achieved. We can eliminate cycle-causing edges before adding them to the graph, and though the test is straightforward and theoretically efficient, we expect it to take a substantial amount of time in practice. We apply templates only if they generate a new state in at least one machine. This suppresses self-loops. In practice we believe cycles will be extremely rare so the current system will eliminate them in postprocessing as necessary.

3.3.4 Scalability

The size of attack graphs will grow with the number of templates and machines on the network under analysis. We cannot yet determine whether we can adequately control and/or manage this explosion. Redundant paths can add an unacceptable exponential growth, so the elimination of redundant paths and nodes as outlined above is critical for scaling. Careful template formulation with very specific criteria can eliminate some of this explosion by modeling network behavior precisely. Aggregation and hierarchy will also help control this growth. As the networks we analyze become large, we must identify ways to decompose the networks in a hierarchical fashion, so that we analyze one security environment or subnet at a time, then feed the results to the next level up. This will help address scaling issues, it may lessen the computational requirements, and it will allow the user to "drill down" to parts of the graph that are logical groupings.

One logical form of graph decomposition groups similar machines in a security environment. For example, if there are 20 Windows NT workstations on a LAN that all have basically the same configuration,

these 20 machines can be aggregated into one for the purposes of the attack graph. In networks where many machines are configured in a similar manner with slight differences, there may be a large number of machine combinations (for example, machines A, B, C, and D may all have one vulnerability in common, and machines A, B, and C may have another, and machines A, C, and D may have another). Determining how to group these subsets in an efficient way so there is as little overlap and redundancy in the paths as possible is quite challenging. To aggregate machines properly, we must identify what machines are "similar enough" to be grouped together, and then we must identify how the edge weights change based on the size of the group. For example, a password-guessing vulnerability that is present in a group of 100 single-user machines would have a higher probability of succeeding once than a password-guessing vulnerability present in a group of 5 single-user machines.

4. Results

We ran the graph generator on a small but realistic example. Figure 10 shows a network with 2 machines, sleepy and grumpy. The boxes show the attributes of the two machines that are used by the attack templates. The attack templates (not pictured) include some well-known, common exploits in use today to compromise systems. Only five attack templates were selected to keep the graph small.

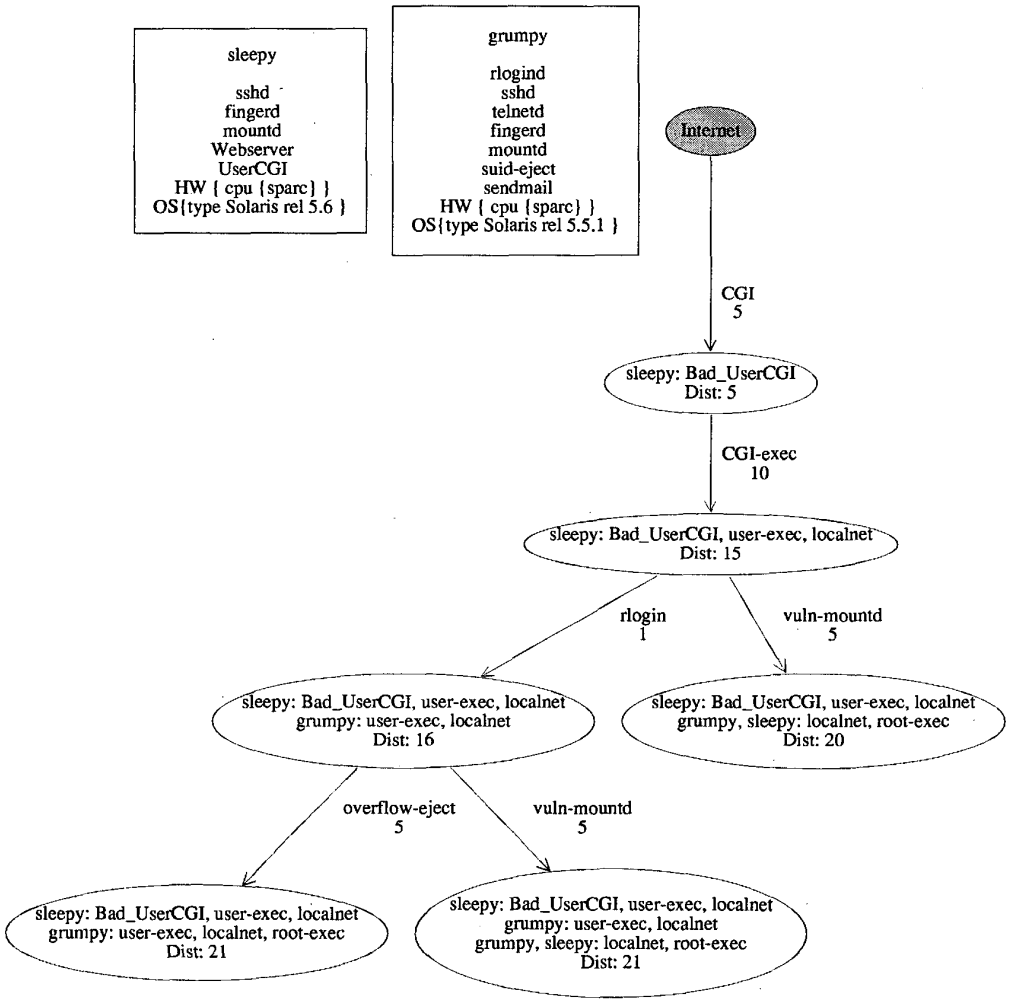

Figure 10. Attack Graph Example

In Figure 10, sleepy is a machine running a webserver with user-level CGI-bin capability and the ability of users to create their own CGI scripts. It is assumed that rlogin and mountd are not passed through the router, rendering such attacks not viable from the Internet. However, sleepy is compromised from the Internet by exploiting a weakness in a poorly-coded CGI script. (Such scripts are common, as most people are unaware of secure coding techniques.) Once an arbitrary remote command can be executed, it paves the way for a larger class of exploits involving the compromise of local machines. In this case, common vulnerabilities were exploited local to the network, e.g. mountd and rlogind. Grumpy is a machine with an older operating system and has additional weaknesses, such as a buffer overflow in eject().

The nodes contain the modified configuration data for each machine. As nodes are traversed, modifications to the configuration data of each machine are incorporated. With this particular example, the graph shows that it is just as easy to get root privileges on both machines as it is to get root privileges on one of them individually. This would likely change with more precise edge weights. The most difficult step (highest edge weight) is that of the attacker finding a weakness in an insecure CGI-bin.

Figure 10 is a small example of a real attack graph. We have generated more complicated, larger graphs but could not include them due to space limitations. We are currently designing experiments to investigate what will be a larger problem in combinatorial explosion during the graph generation: number of templates or number of machines.

Conclusions

This paper has presented a method for vulnerability analysis of computer networks. The method is based on attack graphs which represent attack states and the transitions between them. The attack graph can be used to identify attack paths that are most likely to succeed, or to simulate various attacks. The attack graph could also be used to identify undesirable activities an attacker could perform once he entered the network. The major advance of this method over other computer security risk methods is that it considers the physical network topology and actual machine configurations in conjunction with the set of attacks possible against that configuration. The attack graph represents the dynamics of "system states" in the network changing as the attacker's penetration increases over time. Network configuration/ monitoring/scanning tools take a snapshot of the system configuration and vulnerabilities at one time, but the attack graph method allows for configurations changing over time as the attack occurs and defenses degrade. The dynamic aspect of the graph gives unique insight into the behavior of the system. The graph analysis tool goes beyond tools which check a "laundry list" of services or conditions that are enabled on a particular machine. The graph can show new attack paths which can by-pass the static fortifications of a typical security system.

We are developing a robust attack graph tool with a graphical interface. The tool is designed to be fairly easy to use and to link to vulnerability and configuration databases from commercial tools. We plan to evaluate the utility of this approach for testing intrusion detection systems and/or for identifying where an attacker might next move once an intruder has been detected within a network. The attack graph could also be the basis for identifying the most cost-effective set and placement of defenses.

5. Future Work

The main focus of our work for the remainder of this year involves enhancing and completing the major portions of our tool to make it production quality: adding more redundancy elimination to the graph generator, finishing the intermediate database and associated programs. In the future, we would like to add features such as defense placement and mission path analysis.

Defense placement is an optimization problem. We are given an attack graph and a set of possible defenses, each with a cost associated with placing the defense, and a defense budget in units of the defense cost metric (e.g., financial, loss of service, etc.) We are also given a vector representing attacker-cost increases for each edge in the attack graph. We must select an affordable set of defenses that maximally increase attacker cost as measured by increases in the length of shortest paths in the attack graph. We can initially assume the edge-weight effects are independent, however in general, combinations will have nonlinear effects where a pair can have either a stronger or weaker effect than the sum of the individual effect. This problem has a set-covering flavor: find a combination of defenses that individually won't cover all the holes but collectively will (and whose total cost will remain within the budget). Since set-cover is theoretically difficult, we will have to approximate the optimal defense placement.

We would like to use the graph generator to create "mission paths" in addition to attack paths (note: mission here refers to a business or military mission, such as ensuring successful completion of a logistics command order). The mission graph is different from the attack graph in that the mission graph represents all ordered sets of steps to ensure mission success, not the possible set of attacker steps to defeat the mission. The mission graph is used to identify critical resources (e.g., machines and circuits) to accomplish a particular task. After a mission path evaluation, a network administrator may use tools like ping and traceroute to diagnose communication path availability and network latency for critical nodes and links.

There is much synergy between mission graphs and attack graphs. We can "attack" the mission graph to select a set of attacker goals. This is the defense placement problem, where the attacker is "defending" against mission success. We can then generate an attack graph backward from these goals to see how the attacker might destroy a mission. We can then "attack" this attack graph to select a set of defenses that protect against these mission-damaging attacks. Furthermore, by rerunning mission-graph generation with each possible defense (configuration change), we can generate the defense costs needed as input for defense selection. Implementing a defense strategy on a particular machine could have a widespread effect on the attack graph, since it affects the weight on every edge involving that machine and every attack affected by the defense. The associated increase in mission-path length can measure the cost of each defense.

6. References

[1] Anderson, J. P., *Computer Security Technology Planning Study.* Technical Report ESDTR -73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972.

[2] Common Criteria: The Common Criteria represents the outcome of a series of efforts to develop criteria for evaluation of IT security that are broadly useful within the international community. NIST is one of the government agencies involved in the formulation of these criteria. For more information, see: http://www.commoncriteria.org/

[3] Cormen, T. H, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press/McGraw-Hill, 1990.

[4] Dacier, M., Y. Deswarte, and M. Kaaniche. "Quantitative Assessment of Operational Security: Models and Tools." LAAS Research Report 96493, May 1996.

[5] Meadows, C., "A representation of Protocol Attacks for Risk Assessment", *Network Threats,* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 38, R. N. Wright and P.G. Neumann editors, American Mathematical Society, pp. 1-10.

[6] Moskowithz, I.S, and M. H. Kang, "An Insecurity Flow Model", *Proceedings of the Sixth New Security Paradigms Workshop",* Langdale, Cumbria, UK, September, 1997, pp. 61-74.

[7] Naor, D. and D. Brutlag, "On suboptimal alignment of biological sequences," *Proceedings of the 4th annual Symposium on Combinatorial Pattern Matching*, Springer Verlag, 1993, pp. 179-196.

[8] Ortalo, R., Y. Deswarte, and M. Kaâniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security", in *Dependable Computing for Critical Applications 6 (DCCA'6)*, (M. Dal Cin, C. Meadows and W.H. Sanders, Eds.), Grainau, Germany, March 5-7 1997, Dependable Computing and Fault-Tolerant Systems, vol.11, pp.307-328, ISBN 0-8186-8009-1, IEEE Computer Society Press, 1998.

[9] Ortalo, R., Y. Deswarte, and M. Kaâniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security", in *IEEE Transactions on Software Engineering*, Vol. 25, No. 5, September/October 1999.

[10] Phillips, C. A. and L. P. Swiler, "A Graph-Based System for Network Vulnerability Analysis." In the ACM *Proceedings from the 1998 New Security Paradigms Workshop*, pp. 71-79.