NetSecuritas: An Integrated Attack Graph-based Security Assessment Tool for Enterprise Networks

Nirnay Ghosh
School of IT
Indian Institute of Technology
Kharagpur 721302, India
nirnay.ghosh@gmail.com

Ishan Chokshi
Oracle India Pvt. Ltd.
Bangalore 560076, India
ishan.chokshi@oracle.com

Mithun Sarkar
School of IT
Indian Institute of Technology
Kharagpur 721302, India
mithunsarkar@gmail.com

Soumya K Ghosh
School of IT
Indian Institute of Technology
Kharagpur 721302, India
skg@iitkgp.ac.in

Anil Kumar Kaushik
Department of Electronics and
Information Technology
Government of India
akaushik@deity.gov.in

Sajal K Das
Computer Sc. Department
Missouri University of S&T
Rolla, MO 65409-0350, USA
sdas@mst.edu

ABSTRACT

Sophisticated cyber-attacks have become prominent with the growth of the Internet and web technology. Such attacks are multi-stage ones, and correlate vulnerabilities on intermediate hosts to compromise an otherwise well-protected critical resource. Conventional security assessment approaches can leave out some complex scenarios generated by these attacks. In the literature, these correlated attacks have been modeled using *attack graphs*. Although a few attack graph-based network security assessment tools are available, they are either commercial products or developed using proprietary databases. In this paper, we develop a customized tool, *NetSecuritas*, which implements a novel heuristic-based attack graph generation algorithm and integrates different phases of network security assessment. NetSecuritas leverages open-source libraries, tools and publicly available databases. A cost-driven mitigation strategy has also been proposed to generate network security recommendations. Experimental results establish the efficacy of both attack graph generation and mitigation approach.

Categories and Subject Descriptors

H.2.0 [**Information Systems Applications**]: General—*security, integrity, protection*

Keywords

Network security, Vulnerability assessment, Penetration testing, Attack graph, Mitigation strategy

1. INTRODUCTION

In today's scenario, securing a network from sophisticated intrusions is one of the primary concerns for network administrators. Such attacks have become prominent with the growth of the Internet and web technology. The Internet provides seamless access to remote servers, while glitches in web-based programming have allowed adversaries to install *backdoors* into these servers, leading to the leakage of sensitive information. Even though the critical resources in a network are well-protected by firewalls, yet vulnerabilities existing in other hosts from which the critical assets are reachable, can be used as pivot to launch correlated attacks.

Port scanners, such as *Nmap*[1], *Angry IP scanner*[2], *ScanMetender*[3] and vulnerability assessment frameworks, like *Nessus*[4], detect open ports and vulnerabilities related to the services bind to these ports, respectively. But a limitation with these scanners and assessment frameworks is that *these tools find vulnerabilities per host basis and do not identify all conditions for a correlated attack to take place.* Usually, a vulnerability existing in a particular version of an application has a corresponding *exploit*. The exploit gives unauthorized access to a system, which enables an attacker to execute arbitrary code (known as *payload*), for launching the actual attack. Such attacks may include *installing an agent, privilege escalation, denial-of-service*, and so on.

Traditional network security analysis (e.g., penetration testing) involves an active assessment of the system for any weaknesses, technical flaws or vulnerabilities. Although a few automated penetration testing tools are available, their wide usage is inhibited due to commercial licensing and susceptibility against installing backdoors which may compromise an organization's interests in more than one ways. To evaluate the security of any network and get a holistic and complete picture, it is necessary that the network administrator takes these correlated attacks into account, and does not have the apprehension of unauthorized information leakage. A tool that depicts a succinct representation of different attack scenarios jeopardizing the security of a network is the *attack graph* [12], also known as the *exploit-dependency graph*. Throughout the paper, we will use the

ICDCN '15, January 04 - 07 2015, Goa, India

http://dx.doi.org/10.1145/2684464.2684494.

[1]http://nmap.org/
[2]http://www.angryip.org
[3]http://www.levenfus.com/scanmetender
[4]http://www.nesssus.org

terms "attack graph" and "exploit-dependency graph" interchangeably, with similar implication.

Assessing the security strength of any network through attack graph requires consideration of the following actions: (1) finding system/application level vulnerabilities, (2) modeling different attack scenarios which threaten to compromise critical resources, and (3) recommending mitigation measures. In this work, we present a security management tool, called *NetSecuritas*, which integrates the above steps of attack graph-based network security assessment and thus recommends cost-effective mitigation actions. The framework leverages open-source scanner, exploit databases and methodologies available in the public domain to facilitate security assessment. It gathers vulnerability and other network information and uses them to generate attack graphs. Finally, a novel mitigation technique has been proposed and implemented to generate security recommendations. The attack graph generation algorithm developed in NetSecuritas has been observed to scale polynomially both in terms of space and time. The mitigation actions recommended by our tool takes into consideration both attack graph properties and predefined security budget of any organization.

The rest of the paper is as follows. Section 2 gives an overview on existing attack graph-based tools and summarizes their limitations. Section 3 presents the design of the architecture of NetSecuritas. In Section 4, we present the technique by which NetSecuritas acquires and manages network-specific information. Section 5 describes a novel attack graph modeling technique using the acquired informaiton. In Section 6, a customized attack graph generation algorithm has been proposed. In Section refmitigate, mitigation strategy to generate network securing recommendations has been described. Section 8 gives the performance analysis of the attack graph generation algorithm and justifies the selection of mitigation metrics. Finally, a conclusion is drawn in Section 9.

2. RELATED WORK

In this section, we provide a brief overview of the existing attack graph-based network security assessment tools.

NetSPA (Network Security Planning Architecture) [2] system takes network specific input from custom database on the host and software types, and attacker's initial location during runtime to generate a complete attack graph with multiple targets. Information regarding the vulnerabilities, firewall rules, and the network topology are entered manually into the database. Then attack graphs are generated using a depth-limited forward-chaining depth-first algorithm. NetSPA has been used to implement Multi-prerequisite graph (or MP) graph [7].

In [24] a new interactive tool is developed to provide a simplified and more intuitive understanding of key weaknesses discovered by the attack graph analysis. Separate *treemaps* are used to display host groups in each subnet and hosts within each treemap are grouped based on reachability, attacker privilege level, and prerequisites.

GARNET (Graphical Attack graph and Reachability Network Evaluation Tool) [25] is an interactive visualization tool for attack graph analysis. Some noteworthy capabilities and features of GARNET are to: (i) support for comparable *what-if* analysis for determining the effects of different changes in the network, (ii) model adversaries with multiple skill levels to initiate attacks from either inside or outside the network, and (iii) compute security metrics for complex networks to quantify security and compare different networks in terms of their security.

Topological Vulnerability Analysis (TVA) [8] generates a graph of dependencies among exploits representing all possible attack paths without having to enumerate them explicitly. The polynomial time algorithm described in [1] is used to construct and analyze the attack graph. TVA also generates recommendations to improve network security by changing network configurations or applying different patches. Output of various network scanning and logging tools are provided to TVA to model the various network elements and attack events.

MulVAL [14] expresses existing vulnerability-database and output of scanning tools in Datalog and feeds them into the MulVAL reasoning engine. The reasoning engine consists of a collection of Datalog rules consisting of the operating system behaviors and interactions between various components in the network. The inputs to MulVAL's analysis are: *advisories, host configuration, network configuration, principals, interaction, and policy.* From such inputs, it analyzes the security risks of the software vulnerabilities in a correlated fashion and generates security alerts.

In [16], a Bayesian network-based risk management framework is proposed to quantify the likelihood of attacks in a network using the metrics defined in the *Common Vulnerability Scoring System (CVSS)* [18]. Furthermore, a genetic algorithm has been proposed, which uses this information and user specified cost model to recommend optimal mitigation plans.

We analyzed the above mentioned assessment tools and observed that they have at least one of the following issues:

1. Most of these tools are commercial products.
2. The vulnerability and exploit databases used or customized are not available in the public domain.
3. The mitigation approach does not simultaneously combine the attack graph properties and network security budget.
4. The attack graph generation algorithm is either not clearly explained or does not scale for large networks.

To address these issues, we develop a customized security management tool, NetSecuritas, using open-source libraries and publicly available database. The mitigation strategy considers both logically generated attack scenarios, as well as real security costs, and the customized algorithm developed for the attack graph generation is experimentally found to be scalable.

3. NETSECURITAS DESIGN AND ARCHITECTURE

This section summarizes the web-based interface of NetSecuritas and briefly describes important modules of its architecture.

3.1 Web-based Interface

The web interface of NetSecuritas provides a user friendly environment for the underlying modules. Instead of a standalone application, it has been developed as a web based application due to the following reasons:

- *Portability*: Installation of the application in the user's computer (as required for stand-alone application) is not required.
- *Platform Independence*: The only requirement of this application is a web browser.
- *Secure*: All modules are invoked at the server end and information generated by a user (e.g., port scan, attack graph) are stored in the server which is available only to that particular user.

NetSecuritas has been coded in Java language in the form of Java Server Pages (JSP) and Servlet. Asynchronous JavaScript and XML (AJAX) has been used in some cases to display the overall progress of a time intensive process. As per implementation, different functional blocks of NetSecuritas along with their interconnections are given in Figure 1.

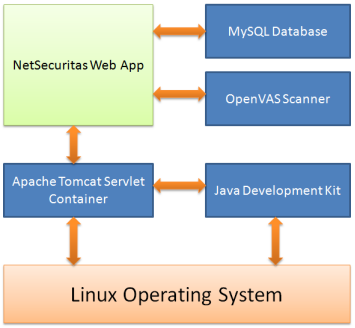

Figure 1: NetSecuritas Block Diagram

3.2 Architecture

NetSecuritas is an integrated network security assessment tool which implements different phases of penetration testing, and appends attack modeling and recommendation of mitigation plans.

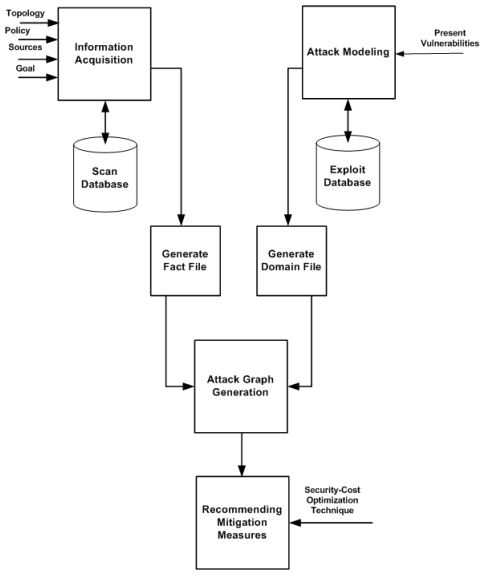

Figure 2: NetSecuritas Architecture

The architecture of NetSecuritas, as illustrated in Figure 2 consists of six major modules:

- *Information Acquisition*: This module collects information related to a network which is subjected to security assessment, and stores them in a *scan database*.
- *Attack Modeling*: In this module, we propose a technique to model exploits exported from an open source database, *Metasploit*[5]. The input to this module is a set of vulnerabilities detected in the network under contention.
- *Generate Fact File*: The fact file encodes the configuration and connectivity information of each host in the network.
- *Generate Domain File*: The domain file models the available exploits in terms of a set of preconditions and effects (postconditions).
- *Attack Graph Generation*: In this module, we generate the attack graph using a customized heuristic-driven algorithm. The input to the generation algorithm are the fact and domain files.
- *Recommending Mitigation Measures*: This module proposes different network securing measures by studying the trade-off between the *level of security achieved* and the *cost of mitigation*.

In the subsequent sections, we describe in details various modules of NetSecuritas.

4. INFORMATION ACQUISITION

One of the important aspects of the information acquisition module is to detect open ports and vulnerable services running on them. For this, two methodologies from traditional penetration testing have been implemented in NetSecuritas:

- *Port scan*: It is a mechanism by which a software application, designed to probe a server or host for open ports, identify running services/applications on a host.
- *Vulnerability assessment*: It is the process of identifying, quantifying, and prioritizing (or ranking) the vulnerabilities in a system.

It has been observed that different port scanners have their specific format to represent the scan results. However, due to the lack of standards, it is difficult for our tool to take scan inputs from these scanners and store them in the internal scan database. Thus, we create a standard scan result schema rather than appending/dropping fields from the database for different formats. Every scan report (preferably in XML format) which a user obtains from the customized tools can be imported into our framework, after checking its validity against the specified *XML Schema Definition (XSD)*. We observe that among different port scanners, *OpenVAS*[6] generates the maximum information following a scan operation. Reports from other scanners have been found to be subsets of that from OpenVAS. However,

[5]http://www.metasploit.com/
[6]http://www.openvas.org/

all information from OpenVAS scan report is not directly required for the attack graph generation. Therefore, for integrating OpenVAS scanner with our tool, we customize it as per requirement. A java based client, termed as OpenVAS client, has been developed to communicate with the remote OpenVAS server. It opens a secured socket at port 9390 to communicate with the OpenVas manager which, in turn, communicates with the scanner. We install NetSecuritas scanner separately in the individual subnets that constitute a network. This is because, for the majority of the scanners, transmitted probe packets are blocked by the firewall as the best practice for security.

4.1 Generation of Fact File

A fact file (in XML format) encodes the configuration and connectivity of each host in the network. Fact file generation module collects data from the information acquisition phase and generates the file automatically. Following are the inputs to fact file generator:

- *List of hosts and services*: This information is retrieved from the vulnerability scan results stored in the scan database.
- *Host to host connectivity*: This is computed from the topology information and network firewall rules.
- *Sources of attack*: It comprises either external sources, viz. the *Internet*, or internal sources, such as a vulnerable host from any subnet.
- *Goals of attack*: It comprises of the critical hosts and the privileges to be achieved.

5. ATTACK MODELING

Modeling of attacks or exploits, obtained from Metasploit, is done so that its properties can be represented in a machine-readable form which is encoded in the domain file.

5.1 Generation of Domain File

The domain file (in XML format) consists of a list of exploits, each of which is represented by two sets of conditions:

1. *Precondition*: A precondition set consists of those conditions which are required to be satisfied conjunctively to triggering any exploit.
2. *Postcondition*: A postcondition set contains those conditions which are generated disjunctively after execution of an exploit. These conditions form a subset of precondition set for a subsequent exploit.

In [19], preconditions for any exploit have been identified as the following set: (i) availability of vulnerable version of the service/application, (ii) connectivity with the target host, (iii) privilege requirement on target host, and (iv) existence of the corresponding vulnerability. On contrary, the set of postconditions is limited to: (i) privilege level gained, and (ii) service/vulnerability disabled.

In a real-world scenario, executing an exploit requires additional system-level constraints to be satisfied. Moreover, the effect generated by executing an exploit is dependent on the type of *payload* used and is not restricted to privilege escalation only. For the present work, exploits available from an open source database, Metasploit, have been considered. It is a platform for writing, testing, and using exploit codes. It has exploits for checking the vulnerabilities of different platforms like *aix, bsdl, FreeBSD, hpux, Linux, Solaris, Windows*, and so on. They are available as ruby codes, however, do not have any standard documentation which makes their usage inconvenient. From these codes, we develop a customized database for storing different attributes relevant to the exploits. Information retrieved from the ruby codes for a particular exploit are used to model the preconditions essential for its exploitation and the postconditions it generates after its execution. The information parsed from the exploit codes are as follows:

- *Name*: name of the vulnerability.
- *Reference*: unique identifier for the vulnerability viz. *CVE-ID*[7], *Bugtraq-ID*[8], *OSVDB-ID*[9] etc.
- *Privileged*: whether or not the exploit module requires or grants privileged access.
- *Platform*: platform on which the exploit works, for example, Windows, Linux, etc.
- *Architecture*: architecture required on the target host for exploit to be executed, e.g. i386, x64_86, etc.
- *Targets*: version/edition of application or operating system which is vulnerable.
- *Registered_options*: depicting the IP address of the remote server and the port number on which the vulnerable service is listening.
- *Description*: information about the effect of exploit execution given in natural language.

Therefore, as evident from the attribute list presented above, modeling of an exploit requires a generic precondition set with the following attributes: (i) vulnerability, (ii) proper version of the vulnerable application, (iii) platform on which the application is running, (iv) architecture, version, and edition of the operating system, (v) privilege level required to exploit the vulnerability, (vi) port number to which the service is bind, (vii) network layer connectivity between attacker and the target.

The postcondition or effect generated by an exploit has been obtained by analyzing its *description* attribute which is composed in natural language and does not have any standard notation scheme. Moreover, for some exploits, it does not provide any conclusive idea about what effect it will generate after execution. To address this problem, descriptions related to a particular exploit is also obtained from two other sources: (i) *OSVDB (Open Source Vulnerability Database)*, and (ii) *Bugtraq*. Finally, a set of *keywords* or *key-phrases* are formed from the descriptions obtained from these three sources. These keywords and key-phrases give a notion about the possible effect the exploit may generate on successful execution. Some of them which characterize exploit effects are: *"arbitrary_code_execution", "service_crash", "privilege_escalation", "installing_agent"*, and so on.

We store these exploit-specific information in a customized database whose schema is given in Table 1 (PK represents

[7]http://cve.mitre.org/
[8]http://www.securityfocus.com/bid/
[9]http://osvdb.org/

Table 1: Exploit Database Schema

Relation	Attributes
Exploit	Id (PK), ExploitDetailName, ExploitAdminName, ActionName, Privilege, UserDescription, AdminDescription
Target	Id (PK), ExploitId (FK), TargetDetailName, TargetAdminName, PortNumber
Operating System	Id (PK), ExploitId (FK), OSDetailName, OSAdminName
Architecture	Id (PK), ExploitId (FK), ArchDetailName, ArchAdminName
Platform	Id (PK), ExploitId (FK), PlatformDetailName, PlatformAdminName
PostConditions	Id (PK), ExploitId (FK), PostConditionDetails, PostConditions
Reference	Id (PK), ExploitId (FK), ReferenceType, ReferenceNumber

the primary key and FK is the foreign key of any relation). An interface has been developed to enable users to manually enter the missing attributes after an exploit code has been parsed.

As is evident from Table 1, most of the relations have an attribute dedicated to the administrator of NetSecuritas. This enables the administrator to remove conflicts or discrepancies in exploit details that may originate at the time of importing the exploit details from the Metasploit database or during manual entry by other users. Manual entry is required for some attributes whose values are either not complete, or have no standard notation, or not available from Metasploit database, viz. *UserDescription, TargetDetailName, OSDetailName, ArchDetailName, PostConditionDetails*, and so on.

6. ATTACK GRAPH GENERATION

In this section, we formally define the attack graph and provide the details of a customized algorithm for its generation.

6.1 Exploit Dependency Graph Model

In [4], the authors propose a graph based approach to model multi-stage, multi-host attack scenarios known as an attack graph. Exploit-dependency graph [10] is one of the variants of the attack graph. Model checker-based exploit dependency graph generation has been reported in [6] [9].

An *exploit dependency graph* consists of a number of attack paths (or, attack scenarios), each of which is a logical succession of exploits and conditions. Multiple attack scenarios can be combined without loss of generality to generate the exploit dependency graph. As explained in Section 5.1, conditions in an attack graph are of two types: (i) precondition, and (ii) postcondition. To generate the attack graph, a set of *initial conditions* and *goal conditions* are required. Initial conditions refer to those network states which are available by default. Goal conditions are those which are to be achieved to compromise a network. With the above notions of exploit-dependency graph, the formal definition is given as follows [23]:

DEFINITION 1. *(Exploit-dependency graph). Given a set of exploits E, a set of security conditions C, a relation* **require** *$R_r \subseteq C \times E$, and a relation* **imply** *$R_i \subseteq E \times C$, an attack graph* **AG** *is an acyclic directed graph $AG(E \cup C, R_r \cup R_i)$, where $E \cup C$ is the vertex set and $R_r \cup R_i$ is the edge set.*

It can be comprehended from Definition 1 that an exploit-dependency graph is a *bipartite* graph with two disjoint sets of vertices, namely, *exploit* and *condition*. The edges are also of two types: (i) the *require* edge captures the conjunctive nature of the conditions to activate an exploit, and (ii) the *imply* edge identifies those conditions which are yielded after an exploit is successfully executed.

6.2 Algorithm to Generate Exploit Dependency Graph

In NetSecuritas, we develop a heuristic-based algorithm (refer to Algorithm 1) to generate exploit-dependency graphs. It takes a list of exploits (from the domain file) and a list of hosts (from the fact file) as input, and generates exploit dependency graph in the form of an adjacency list. *Exploit* is a data structure that contains three other structures of type *condition*:

1. *Configuration condition*: specifies configuration parameters such as platform, operating system, version, service, vulnerability which are essential for the execution of exploit.

2. *Connectivity condition*: specifies connectivity parameters which are required for an attacker to execute exploit. Connectivity parameters are TCP and IP level reachabilities.

3. *Effect*: post conditions generated by the exploit.

A list of hosts, called *hostList*, contains elements of the *host* data structure which specifies configuration of host, services running, vulnerabilities, connectivity with other hosts, and so on. The proposed algorithm follows a backward chaining approach in which it starts from the goal state and finds paths that terminate at the initial conditions. A stack is used to facilitate backtracking and stores elements of type *node* which can be either *exploit* or *condition*. To model attacks whose source is external to the target network, we formulate a dummy host *Internet*. Based on the firewall rules of the network, *Internet* may have access to some services in the network.

In the backward chaining approach, the algorithm starts with a goal node, finds list of exploits that can yield the desired condition. For multiple such exploits, it makes a decision to chose the *easiest* one. This exploit is considered for next iteration while others are pushed into the stack and processed later. In the next iteration, the algorithm finds a list of hosts (including *Internet*) through which that exploit can be executed. Once again, the algorithm makes decision to pick the host which is *easiest* to exploit and push others into the stack. All the nodes in a path are stored in a

queue and flagged. Whenever the exploration is completed, successfully or in the dead end, the algorithm backtracks and evaluates other hosts or exploits from the stack. Thus, all the host and vulnerabilities are evaluated at least once. Hence, no vulnerability which is exploitable and contributes in at least one attack path will be ignored. Two heuristics are computed to select an *easiest* exploit or host:

1. *CVSS exploitability score of vulnerability*: This heuristic is based on the assumption that when multiple vulnerabilities exist in a host, the attackers choose the one *easiest* to exploit. This can be measured by the *CVSS exploitability score* [18].

2. *Number of vulnerabilities in a host*: As per the connectivity in the network, an exploit can be executed from multiple hosts. An attacker can use any of these hosts as intermediate steps to reach to his goal. Our assumption is the host with higher number of vulnerabilities is more likely to be detected and used by the attacker to execute the exploit. When more than one hosts have the same number of vulnerabilities, CVSS exploitability score of vulnerabilities is used to resolve the tie.

7. RECOMMENDING MITIGATION MEASURES

Ideally, any administrator intends make the network secure as well as completely operational. However, a major concern with achieving a desired level of security is the cost incurred to the organization. In reality, organizations have a predefined budget for security purpose, within which they have to make their networks secure. Hence, there is a trade-off between the targeted security level and the fixed budget. In this situation, the security experts have to make decisions as regards to prioritizing or properly ordering the network hardening actions. In NetSecuritas, we define a novel mitigation metric which will be computed for all exploit nodes in the generated attack graph, and based on the values obtained, an appropriate security action will be recommended. The cost of this action can be given as input either in terms of the vulnerability patching cost or the impact generated on the network if a certain service is blocked.

7.1 Mitigation metric

The proposed mitigation metric combines both the logical representation of attack scenarios and real security actions to be taken. It comprises of the following two sub-metrics:

7.1.1 Graph Metric

Graph metric is based on the in-degree measures of exploit nodes in the attack graph. In-degree can be an indicator of the number of attacks in which the exploit is involved. Intuitively, an exploit with higher in-degree will be involved in more attack paths. Therefore, if $indeg_1, indeg_2, \ldots, indeg_N$ are the in-degree measures of N nodes of an exploit dependency graph, then, at a particular instant, the graph metric is given as:

$$\mu_{grp} = MAX\{indeg_1, indeg_2, \ldots, indeg_N\} \quad (1)$$

7.1.2 Cost Metric

Input: exploitList, hostList
Output: Exploit Dependency Graph
from goal condition $G(H)$;
initialize *stack*;
initialize *path*;
form a starting node *startNode*;
$push(stack, G(H))$;
while *Stack is not empty* **do**
  $currentNode \leftarrow getTOS(stack)$;
  if $currentNode \in path$ **then**
    $backtrack(stack, path)$;
    continue;
  end
  if *path exists from currentNode* **then**
    $path.add(currentNode)$;
    $savePath(currentNode)$;
    $backtrack(stack, path)$;
    continue;
  end
  if *currentNode is of type condition* **then**
    $exploitList \leftarrow findRequiredExploit(currentNode)$;
    if $exploitList == \phi$ **then**
      $backtrack(stack, path)$;
      continue;
    end
    $heuristicalSortExploit(exploitList)$;
    for *each exploit* $\in exploitList$ **do**
      $push(stack, exploit)$;
    end
  end
  else
    $conditionList\ C \leftarrow$
    $findExploitPostcondition(currentNode)$;
    if $C == \phi$ **then**
      $backtrack(stack, path)$;
      continue;
    end
    if $startNode \in C$ **then**
      remove *startNode* from C;
      $path.add(startNode)$;
      $savePath(path)$;
      $path.remove(startNode)$;
      if $C == \phi$ **then**
        $backtrack(stack, path)$;
        continue;
      end
    end
    $heuristicalSortHost(C)$;
    for *each condition* $\in C$ **do**
      if $condition \in path$ **then**
        remove *condition* from C;
      end
      else
        $push(stack, condition)$;
      end
    end
  end
end

Algorithm 1: Exploit Dependency Graph Generation

In exploit dependency graph, for each node, there exists a service s and its corresponding vulnerability v. Therefore, two possible network securing options/actions are: (1)patching vulnerability v, and (2)blocking service s. Some cost is associated with both these actions. While, the vulnerability patch is expressed in monetary terms, the cost of stopping a service is proportional to the impact it generated on the organization.

The cost incurred in patching a vulnerability is obtained

from the *CVSS Remediation Level* [18], which is one of the temporal metrics defined under the common vulnerability scoring system. Four qualitative categories under remediation level are: (i) official-fix (OF), (ii) temporary-fix (TF), (iii) workaround (W), (iv) unavailable/not defined (U). For estimating the cost of patches, we need to quantify the above mentioned qualitative categories. Assigning values to qualitative categories is subjective, and depends on the security experts of organizations. Implementing all such estimation methods is not feasible. However, NetSecuritas allows administrators to import their tailor-made inputs for evaluating the cost and impact parameters.

In this paper, we present one such method of estimation. Let the values assigned to the qualitative categories be as follows: $OF = 4, TF = 3, W = 2, U = 1$. We scale down the quantitative values, such that cost of patching (c_v) lies within the range [0, 1]. If λ is the scaling function, then we have: $\lambda^{OF} = 1.0, \lambda^{TF} = 0.75, \lambda^{W} = 0.5, \lambda^{U} = 0.25$.

Similarly, if a service is blocked for security reasons, the impact generated on the organization is proportional to its *criticality*. The higher the criticality of service, the greater will be the impact, and vice-versa. We define three qualitative categories to characterize criticality of a service: *(i) high (H), (ii) medium (M),* and *low (L)*. For estimating the cost of impact, these qualitative categories are quantified as: $H = 10, M = 5$, and $L = 1$. Likewise, we scale down down the quantitative values, such that cost of blocking a service (c_s) lies within the range [0, 1]. If λ is the scaling function, then we have: $\lambda^{H} = 1.0, \lambda^{M} = 0.5, \lambda^{L} = 0.1$.

For any node i in the exploit dependency graph that has maximum in-degree measure, the cost metric is given as the minimum of costs associated with the two actions:

$$\mu^{i}_{cost} = MIN\{c^{i}_{v}, c^{i}_{s}\} \quad (2)$$

7.2 Mitigation Strategy

We propose a greedy approach-based mitigation strategy, which comprises of two steps:

- Selection of exploit node from the present attack graph based on μ_{grp} measures.
- Selection of mitigation action based on μ_{cost} value.

In every iteration, the node to be eliminated from the exploit dependency graph is determined by the graph metric (using Equation 1), and the action to be taken is decided by the cost metric (using Equation 2). This continues until the summation of costs of subsequent mitigation actions surpasses the predefined security budget. In each iteration, a node is removed from the exploit dependency graph, a new graph is regenerated, and the mitigation metrics for the new graph is computed.

As explained above, the cost metric is subjective and varies from one organization to another. It will be difficult for NetSecuritas to pre-compute the cost of patching a vulnerability or blocking a service. Moreover, the cost budget and the level of security to be achieved are organization-specific. These parameters are independent of the attack graph and are to be provided as input by a network administrator to obtain optimal mitigation strategy. However, as the graph metric is based on the generated exploit dependency graph, NetSecuritas presents a summary of in-degree measures of all nodes to the user from which the mitigation actions can be decided.

8. RESULTS AND DISCUSSIONS

In this section, we analyze the results related to exploit dependency graph generation and compare our methodology with the existing ones in the literature. Also, we illustrate the efficacy of choosing in-degree of attack graph nodes as one of the mitigation metrics.

8.1 Analysis of Exploit Dependency Graph Generation Algorithm

The exploit dependency graph generation algorithm (refer to Algorithm 1) starts from the specified goal condition and moves backwards till it reaches the initial condition. The algorithm first finds the *easiest* path from the goal node to the start node (which may be *Internet*) based on the above discussed heuristics. As the algorithm starts from the goal condition and terminates only after reaching the initial condition, it can be claimed that if there exists one and only one attack path, the algorithm guarantees to find it. This proves the *completeness* of the proposed algorithm.

8.1.1 Time Complexity

According to Algorithm 1, whenever a path is found, all the nodes constituting the path are flagged to indicate that a path exists from them. Now, in any further path exploration, if the search reaches to any of those "flagged" nodes, the algorithm knows that at least one path already exists from there onwards. Hence that exploration is called *successful* and a new path is saved. This way, every exploit and every condition are explored only once. Thus, time complexity can be estimated as: $T(n, e) = O(n * T(condition) + e * T(exploit))$ where, n = total number of hosts, e = total number of exploits, $T(condition)$ = time required for exploring a condition node, and $T(exploit)$ = time to explore an exploit node. In the worst case, if it is assumed that a condition is generated by e number of exploits, then $T(condition) = O(e)$. Similarly, exploring an exploit node is about finding all preconditions or all the hosts from which the exploit can be executed. In the worst case, a particular exploit may be present in all the hosts of the network, resulting in the upper-bound time complexity as $T(exploit) = O(n)$.

Thus, the total time complexity is $T(n, e) = O(n * O(e) + e * O(n)) = O(ne)$. Therefore, unless the number of exploits is exponential to the size of the network size, the proposed algorithm yields the exploit dependency graph in polynomial time. However, in realistic scenario, for a large network, with stringent access policy and vulnerability assessment strategy, the availability of exponential number of exploits is infeasible.

8.1.2 Space Complexity

In the worst case, if a completely connected topology is considered, such that every host is allowed to access every service from all the other hosts, then every exploit can be executed from all the other hosts. Thus at any instance, the maximum number of nodes that can be pushed into the stack is given by: $n+(n-1)+(n-2)+\ldots+1+e = n(n+1)/2+e = O(n^2 + e)$. Therefore, similar to the time complexity, the space requirement is also polynomial to the number of hosts in the network, if e is bounded by a polynomial.

8.1.3 Performance

For measuring the performance of our attack graph generation module, we simulated two types of networks:

1. *Flat topology*: Consists of a single subnet, in which each host is connected to every other host and can access any service running in any of the hosts.
2. *Enclave topology*: It is equivalent to networks of academic institutions and medium to large size enterprise networks. Here, a network is divided into multiple subnets.

Figures 3 and 4 show the running time of Algorithm 1 with respect to the number of hosts, for flat and enclave networks, respectively. Both the graphs show linear time increase with increased number of hosts. For enclave network structure, extra time is incurred for computing reachability of the hosts.

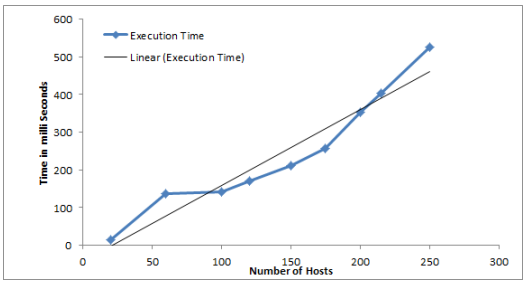

Figure 3: Running time of Algorithm 1 on Flat Network Topology

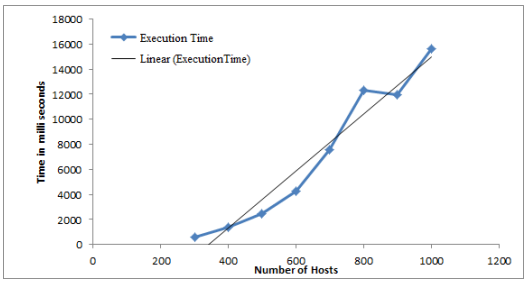

Figure 4: Running time of Algorithm 1 on Enclave Network Topology

8.2 Comparison with Reported Works

Literature survey on attack graphs show that researchers have used both custom algorithms [5] [13] [15] [22] as well as formal methods [3] [11] [17] [19] [21] [22] to generate attack graphs. Formal methods typically involve representation of attacks, networks, and vulnerabilities in some formal language and providing them as input to the model checkers. This, in turn, generates attack paths as counter-examples to show that a security condition is violated. Attack graphs with one single goal as well as those with multiple goals have been generated for network security assessment. The results of research works which deal with generation of attack graphs are presented in Table 2.

8.3 Analysis of Mitigation Strategy

In Section 7, we have explained an exploit-dependency graph based mitigation strategy. It reduces the number of attack scenarios from an attack graph, by recommending mitigation actions. In Figures 5, 6, and 7, we study the effects of four mitigation strategies (in-degree, out-degree, sum of in- and out-degrees, random weight) on the numbers of iterations required to reduce an attack graph. The figures show the results corresponding to the number of attack paths, edges, and nodes, respectively.

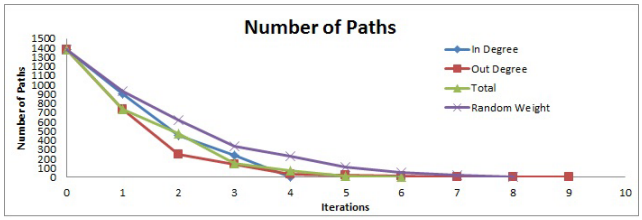

Figure 5: Effect on the Number of Attack Paths

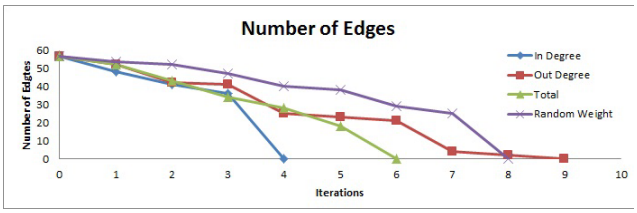

Figure 6: Effect on the Number of Edges

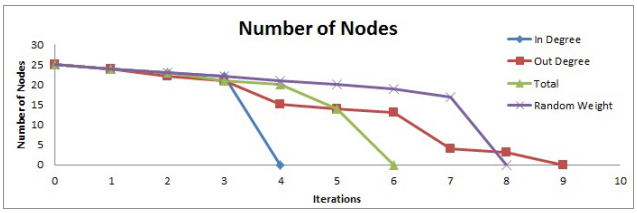

Figure 7: Effect on the Number of Nodes

As is evident from these studies, the steepest decrease in all graph-related parameters occurs when vulnerabilities are patched according to the maximum in-degree. This validates our choice of in-degree as a part of attack graph-based mitigation metric. However, the action to be taken on the physical network for mitigation is driven by cost of patch and impact on the enterprise if a service is blocked.

9. CONCLUSION

Security in enterprise networks has become a major concern with the proliferation of the Internet and web-based technology. Sophisticated cyber attacks combine vulnerabilities existing on different hosts and use them as pivots to compromise targeted resources. The attack graph (or exploit dependency graph) models such correlated multi-stage attack scenarios, thus giving an administrator a holistic idea of a network's security loopholes. In this paper, we presented the description of a network security assessment tool, NetSecuritas, which integrates different phases of penetration

Table 2: Comparative Study with Related Work

Reference	Approach	Results	Remarks
Ammann, 2002 [1]	Custom algorithm has been developed. A small test network with 3 hosts/6 vulnerabilities has been taken.	The algorithm grows at $O(n^6)$ with the size of the network.	Finds shortest path which can be reached to the goal. Scales to only hundreds of nodes.
Dawkins, 2004 [5]	Formal method to generate attack chaining trees.	Shows poor scaling results.	Generates full attack graph and finds out a "minimum cut set".
Jajodia, 2003 [8] [12]	Customized algorithm to automatically generate attack graphs. A test network comprising of 3 hosts/4 vulnerabilities has been taken	Base computation grows as n^6.	Computes attack graph using vulnerability and reachability information from *Nessus* and makes recommendations to prevent access to critical resources.
Ritchey, 2000 [17]	Modeling of network hosts, connectivities, attacker's point of view, and exploits using *SMV* model checker. A network consisting of 4 hosts has been used for case study.	Poor	Scalability problem as the size of the state space increases. Modeling of hosts, vulnerabilities, and exploits are done using arrays. This prevents dynamic addition and also their sizes have direct impact on the state space.
Sheyner, 2002 [19]	Uses *NuSMV* model checker to automatically generate attack graphs. The proposed methodology has been tested against a network with 3 hosts/4 vulnerabilities.	Poor	Generates attack graph with the test network in 5 seconds. But for a network with 5 hosts/8 vulnerabilities, it takes 2 hours to generate the graph.
Swiler, 2001 [20]	Proof-of-concept attack graph generation tool. A test network with 2 hosts/5 vulnerabilities have been taken for case study.	Poor	Builds a full attack graph first and then finds out the shortest paths to specified goals by assigning some weights on the edges.
Proposed Work	Uses a customized back-chaining algorithm to generate attack path from manually coded input specification files.	The time complexity is $O(ne)$ and the space complexity is $O(n^2e)$ for n hosts and e exploits.	Two heuristics have been adopted to avoid exploration of the subgraph which has been generated earlier. Backtracking scheme also ensures completeness of the algorithm.

testing. The tool customizes open-source scanner and parses information from the exploit framework. The attacks have been modeled in terms of a set of preconditions and postconditions. A customized algorithm has been developed for efficient generation of the exploit dependency graph. We devise a mitigation strategy, based on the graph property and the security budget, to generate recommendations. Analysis of results demonstrate the scalability of the tool with respect to the size of the network, and the practicality of the proposed mitigation metric.

Acknowledgment

The work is partially supported by a research grant from the Department of Electronics and Information Technology (DeitY), Ministry of Communication and Information Technology, Government of India under Grant No. 12(14)/09-ESD, dated 11-Jan-2010. The works of S. K. Das is partially supported by the US National Science Foundation under Award Numbers CNS-1404677 and DGE-1433659.

10. REFERENCES

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

[2] Artz, M. *NetSPA: A Network Security Planner*. PhD thesis, Massachusettes Institute of Technology, May 2002.

[3] Chen, F., Su, J., and Zhang, Y. A scalable approach to full attack graphs generation. In *Engineering Secure Software and Systems*, F. Massacci, J. Redwine, SamuelT., and N. Zannone, Eds., vol. 5429 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 150–163.

[4] Cynthia Phillips, L. P. S. A graph based system for network-vulnerability analysis. NSPW 98 Proceedings of the 1998 workshop on New Security Paradigms, ACM, pp. 71–79.

[5] Dawkins, J., and Hale, J. A systematic approach to multi-stage network attack analysis. In *Proceedings of the Second IEEE Internation Information Assurance Workshop (IWIA '04)* (2004), IEEE Computer Society, pp. 48–56.

[6] Ghosh, N., and Ghosh, S. A planner-based approach to generate and analyze minimal attack graph. *Applied Intelligence 36* (2012), 369–390.

[7] Ingols, K., Lippmann, R., and Piwowarski, K. Practical attack graph generation for network defense. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)* (December 2006), pp. 121–130.

[8] Jajodia, S., Noel, S., and O'Berry, B. Topological analysis of network attack vulnerability. In *Managing Cyber Threats: Issues, Approaches and Challenges* (2005), vol. V, Springer US, pp. 247–266.

[9] Jha, S., Sheyner, O., and J.Wing. Two formal analyses of attack graphs. Proceedings of the 15th

IEEE Computer Security Foundations Workshop (CSFW02).

[10] Lingyu Wang, Anyi Liu, S. J. Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts. *Computer Communication 29, Issue15* (September 2006), 2917–2933.

[11] Liu, X., Fang, C., Xiao, D., and Xu, H. A goal-oriented approach for modeling and analyzing attack graph. In *Information Science and Applications (ICISA), 2010 International Conference on* (April 2010), pp. 1–8.

[12] Noel, S., Jajodia, S., O'Berry, B., and Jacobs, M. Efficient minimum-cost network hardening via exploit dependency graph. In *Proceedings of* 19^{th} *Annual Computer Security Applications Conference (ACSAC 2003)* (2003), pp. 86–95.

[13] Ortalo, R., Deswarte, Y., and Kanniche, M. Experimenting with quantitative evaluation tools for monitoring operational security. In *IEEE Transactions on Software Engineering, 25(5)* (October 1999), pp. 633–650.

[14] Ou, X., Govindavajhala, S., and Appel, A. W. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th USENIX Security Symposium* (July 31 – August 5 2005), pp. 113Ű–128.

[15] Phillips, C., and Swiler, L. P. A graph-based system for network-vulnerability analysis. In *Proceedings of the Workshop on New Security Paradigms (NSPW)* (22-26 September 1998), pp. 71–79.

[16] Poolsappasit, N., Dewri, R., and Ray, I. Dynamic security risk management using bayesian attack graphs. *Dependable and Secure Computing, IEEE Transactions on 9*, 1 (2012), 61–74.

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

[18] Schiffman, M. Common vulnerability scoring system (cvss). *http://www.first.org/cvss/ (accessed on October 2014)*.

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

[20] Swiler, L. P., Phillips, C., Ellis, D., and Chakerian, S. Computer-attack graph generation tool. In *Proceedings of the 2nd DARPA Information Survivability Conference & Exposition (DISCEX II)* (2001), vol. II, IEEE Computer Society, pp. 307–321.

[21] Templeton, S., and Levitt, K. A requires/provides model for computer attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms* (18-21 September 2001), ACM Press, pp. 31–38.

[22] Tidwell, T., Larson, R., K.Fitch, and Hale, J. Modelling internet attacks. In *Proceedings of the Second Annual IEEE SMC Information Assurance Workshop* (June 2001), IEEE Press, pp. 54–59.

[23] Wang, L., Noel, S., and Jajodia, S. Minimum cost-network hardening using attack graphs. *Computer Communications, 29(18)* (November 2006), 3812–3824.

[24] Williams, L., Lippmann, R., and Ingols, K. An interactive attack graph cascade and reachability display. *VizSEC 2007* (2008), 221–236.

[25] Williams, L., Lippmann, R., and Ingols, K. GARNET: A graphical attack graph and reachability network evaluation tool. *Visualization for Computer Security* (2008), 44–59.