Appl Intell (2012) 36:369–390
DOI 10.1007/s10489-010-0266-8

A planner-based approach to generate and analyze minimal attack graph

Nirnay Ghosh · S.K. Ghosh

Published online: 25 November 2010

Abstract In the present scenario, even well administered networks are susceptible to sophisticated cyber attacks. Such attack combines vulnerabilities existing on different systems/services and are potentially more harmful than single point attacks. One of the methods for analyzing such security vulnerabilities in an enterprise network is the use of *attack graph*. It is a complete graph which gives a succinct representation of different attack scenarios, depicted by *attack path*s. An attack path is a logical succession of *exploits*, where each *exploit* in the series satisfies the preconditions for subsequent *exploits* and makes a causal relationship among them. Thus analysis of the attack graph may help in assessing network security from hackers' perspective. One of the intrinsic problems with the generation and analysis of such a complete attack graph is its scalability. In this work, an approach based on *Planner*, a special purpose search algorithm from *artificial intelligence* domain, has been proposed for time-efficient, scalable representation of the attack graphs. Further, customized algorithms have been developed for automatic generation of attack paths (using *Planner* as a low-level module). The analysis shows that generation of *attack graph* using the customized algorithms can be done in polynomial time. A case study has also been presented to demonstrate the efficacy of the proposed methodology.

Keywords Network security · Attack graph · Attack path · Exploit · Planner

N. Ghosh · S.K. Ghosh (✉)
School of Information Technology, Indian Institute of Technology, Kharagpur 721302, India
e-mail: skg@iitkgp.ac.in

N. Ghosh
e-mail: nirnay.ghosh@gmail.com

1 Introduction

Today's enterprise networks are becoming increasingly vulnerable against intrusions and sophisticated cyber attacks. Therefore, a network administrator has to analyze the security requirements of the network in such a way, that it becomes sufficiently secure as well as operational. Present day's security threats involve multi-stage, multi-host attacks. In such correlated attacks, vulnerabilities existing in different hosts are combined to compromise a target. Network securing technologies include some efficient network scanners such as Nessus,[1] Retina,[2] Nmap,[3] CyberCop[4] and so on. These scanning tools are useful as far as detecting vulnerabilities local to a system but do not identify all conditions for a complete attack, or how different vulnerabilities existing in different systems are correlated to produce multi-stage, multi-host attacks. One such tool that gives description about the correlated attacks in a network is the *attack graph*. Each node in an attack graph represents an *exploit* which an attacker utilizes in various stages of an attack. The *edges* constitute the *security conditions* that are required for successful execution of an *exploit* or results obtained after an *exploit* is executed. Traditionally, attack graphs are complete graphs which depict all possible attack scenarios. This included both successful attacks as well as those which do not end up reaching a *goal* state. Such complete attack graphs contain redundant nodes and edges and are also difficult to visually apprehend and analyze. These issues can be addressed by generating a *minimal attack graph*. The minimal

[1] http://www.nesssus.org.
[2] http://www.eeye.com/html/products/Retina.
[3] http://www.insecure.org/nmap/index.html.
[4] http://www.nai.com.

attack graph consists of those edges which terminate to the *goal* (or target) node i.e., it consists of only successful attack scenarios. This paper focusses on generation and analysis of a minimal attack graph using *Planner*, an AI technique, which may facilitate an administrator in securing the enterprise network.

A fairly good number of research works have been done on generation and scalable representation of attack graphs. Researchers have used either custom algorithms or formal methods to build attack graphs. In some literatures, *attack graph* is also termed as the *exploit dependency graph* [1]. Different custom algorithms [2–5] that attempt to construct *full* attack graphs scale exponentially to the number of hosts present in the network. Formal methods used in [3, 6–8] for generation of attack graphs although explore the entire space of allowable attack paths but do not scale efficiently. Moreover, *model checkers* used for graph generation have *state space explosion* problem and do not handle a realistic set of *exploits* even for a moderate sized network. To improve the complexity of graph generation, some of the approaches [1, 9, 10] introduced an explicit assumption of *monotonicity*. This means once an attacker has gained certain level of privileges on a particular host, he does not have to regain them in the near future. This removes the concept of back-tracking from the attack graphs and the complexity is improved from exponential to polynomial one. However, the attack graph which is generated based on *monotonic* assumptions are still not scalable and also contains a number of redundant paths. This scalability issue has been addressed in [11–13] where instead of generating *full* attack graphs, the concept of *minimal* attack graph has been introduced. It is an attack graph where all the attack paths terminate to a particular *goal* node. But the generation of *minimal* attack graphs still takes polynomial time (in some literatures [9] it has been specified as $O(n^6)$, where n is the number of hosts). In [14], a *multi-prerequisite graph* has been proposed that scales nearly linearly with the size of the network.

In the present work, *SGPlan*, a variant of *Planner* has been used to generate *minimal* attack paths, which are eventually collapsed to form a *minimal* attack graph using custom algorithms in polynomial time. This paper primarily focusses on *time efficient generation of a minimal attack graph using a model-checker that removes visualization problems and avoids state-space explosion*.

The organization of the rest of the papers is as follows. Section 2 gives a brief overview of attack graphs and *Planner*. Section 3 describes the proposed methodology along with a case study. Section 5 analyzes the time complexity of the proposed approach and a conclusion is drawn in Sect. 6.

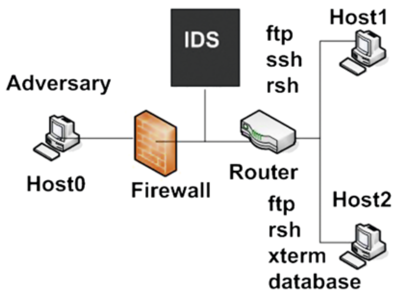

Fig. 1 Example network

Table 1 Host description

Host	Services	Vulnerabilities	OS
Host0	ssh	ssh buffer overflow	Linux
	ftp	ftp rhost overwrite	
	rsh		
Host1	ftp	ftp rhost overwrite	Linux
	xterm	local xterm buffer overflow	
	rsh		
	mysqld		

Table 2 Connectivity-limiting firewall policies

Relation	Host0	Host1	Host2
Host0	localhost	ftp, ssh	ftp
Host1	Any	localhost	ftp
Host2	Any	ftp	localhost

2 Background

In this section, a brief overview of attack graph model has been presented with the help of an example network. Beside this, an overview of *Planner* has also been given to illustrate how it has been utilized for attack graph generation.

2.1 Attack graph model

Attack graph models different vulnerabilities existing on different hosts and combines them to generate different attack scenarios. Essentially, an attack graph consists of a number of attack paths which are logical succession of **exploits**.[5] Each exploit in the series satisfies the pre-conditions for a subsequent exploit and makes a cause-effect relationship among each other. A subset of nodes in the attack graph are termed as the *goal* nodes. A *goal* node is typically described by an exploit which exists on a target machine, execution

[5] A software code that utilizes an existing vulnerability to make a system behave in an unprecedented way.

of which leads to the compromise of a critical resource in that machine. Two types of attack graphs have been used in previously reported works which are as follows:

– **Complete/Full Attack Graph**: In this type of attack graph, all possible attack scenarios (both successful as well as unsuccessful) are depicted.
– **Minimal Attack Graph**: Minimal attack graph consists of only attack scenarios that terminate to a particular goal node.

In the present work, the concept of minimal attack graph has been used. An illustration of attack graph along with a network configuration is given in Fig. 1 [15]. The network (refer to Fig. 1) consists of two internal hosts viz. *Host1* and *Host2*, which are separated from the external host (i.e. *Host0*) by a firewall. The attacker has compromised *Host0* and aims at compromising the *database* in *Host2*. The host descriptions of the network are given in Table 1. The connectivity-limiting firewall policy for the given network has been composed in Table 2. The firewall allows the inbound *ftp* and *ssh* packets to communicate with *Host1* and *Host2* while it blocks other packets. Within the network, the firewall enables the internal hosts to connect to remote servers on any port.

Combining the vulnerabilities present in different hosts (refer to Table 1) using the connectivity-limiting firewall policies (refer to Table 2), corresponding minimal attack graph (shown in Fig. 2) is generated. It can be visualized as a directed graph where the exploits are shown in *oval* and the *security conditions* are shown in *plaintext*. For example, *ftp_rhosts(0, 1)*, an instance of the generic exploit *ftp_rhosts*, represents that the attacker on *Host0* creates a "*.rhost*" file on *ftp* home directory to establish a remote trust relationship with *Host1*. Similarly, *trust(0, 1)* represents a *security condition* where a trust relationship is established between *Host0* and *Host1*. Generally, a node in the attack graph represents an *exploit* and an edge represents either a *condition* or an *available exploit* or the *privilege* gained after the application of the *exploit*. As evident from Fig. 2, the attack graph consists of three *attack paths* that terminate to a desired *goal* node, i.e., *local_bof(2, 2)*. The result of successfully executing *local_bof(2, 2)* exploit enables an attacker to obtain *root* privilege on *Host2*.

2.1.1 Formal definitions

In this subsection, a formal definition of attack graph (refer to Fig. 2) along with its related terminologies are presented. Formally, an attack graph may be defined following the footsteps of [16] as,

Definition 1 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 $AG(E \cup C, R_r \cup R_i, C_0, C_{goal})$, where $E \cup C$ is the vertex set and $R_r \cup R_i$ is the edge set, C_0 is a set of initial conditions, C_{goal} is a set of goal conditions: $C_0 \subseteq C$ and $C_{goal} \subseteq C$.

An important observation related to attack graph is that a *require* relation is always a *conjunction* of security conditions, whereas an *imply* relation is always a *disjunction* of the same. Therefore, a security condition may be formally defined as,

Definition 2 A security condition $c \in C$ between a source host h_S and a target host h_T in an attack graph **AG** is a triplet (c, h_S, h_T), where c is either a precondition required for launching an instantiated exploit $e \in E$ or a postcondition resulted after launching.

Any node in an attack graph is essentially an instantiated exploit which can be launched between a pair of hosts if a set of security conditions is *conjunctively* satisfied. Successful execution of the exploit generates a *disjunctive set* of results or effects. Therefore, given a set of instantiated exploits E and a set of hosts H, and a set of conditions C, formal definition of a node in an attack graph is given as,

Definition 3 Given security conditions $c_1, c_2, \ldots, c_n, \forall c_i \in C$, a node $v \in V$ in an attack graph AG is a triplet (e, h_S, h_T), where $e \in E$ is an instantiated exploit, h_S is the source host, h_T is the target host and exploit e can be launched iff the relation $c_1 \wedge c_2 \wedge \cdots \wedge c_n$ is satisfied.

An edge in an attack graph is either a transition between a pair of instantiated exploits or between an instantiated exploit and a *security condition* or vice-versa or it may be the *privilege* gained after the application of the exploit. Formal definition of an edge in an attack graph is as follows,

Definition 4 An edge $e \in E$ in an attack graph AG is a directed arc either between a pair of nodes or between a node and a security condition or vice versa.

2.1.2 Motivation behind efficient attack graph generation

As network of hosts continues to grow in number and complexity, they are becoming more and more vulnerable to sophisticated attacks. Such attacks combine vulnerabilities existing on different machines to compromise a critical resource in the network. Therefore, protecting network resources from such correlated, multi-stage, multi-host attacks has become an area of active research. Attack graph can be visualized as a tool that provides a succinct representation

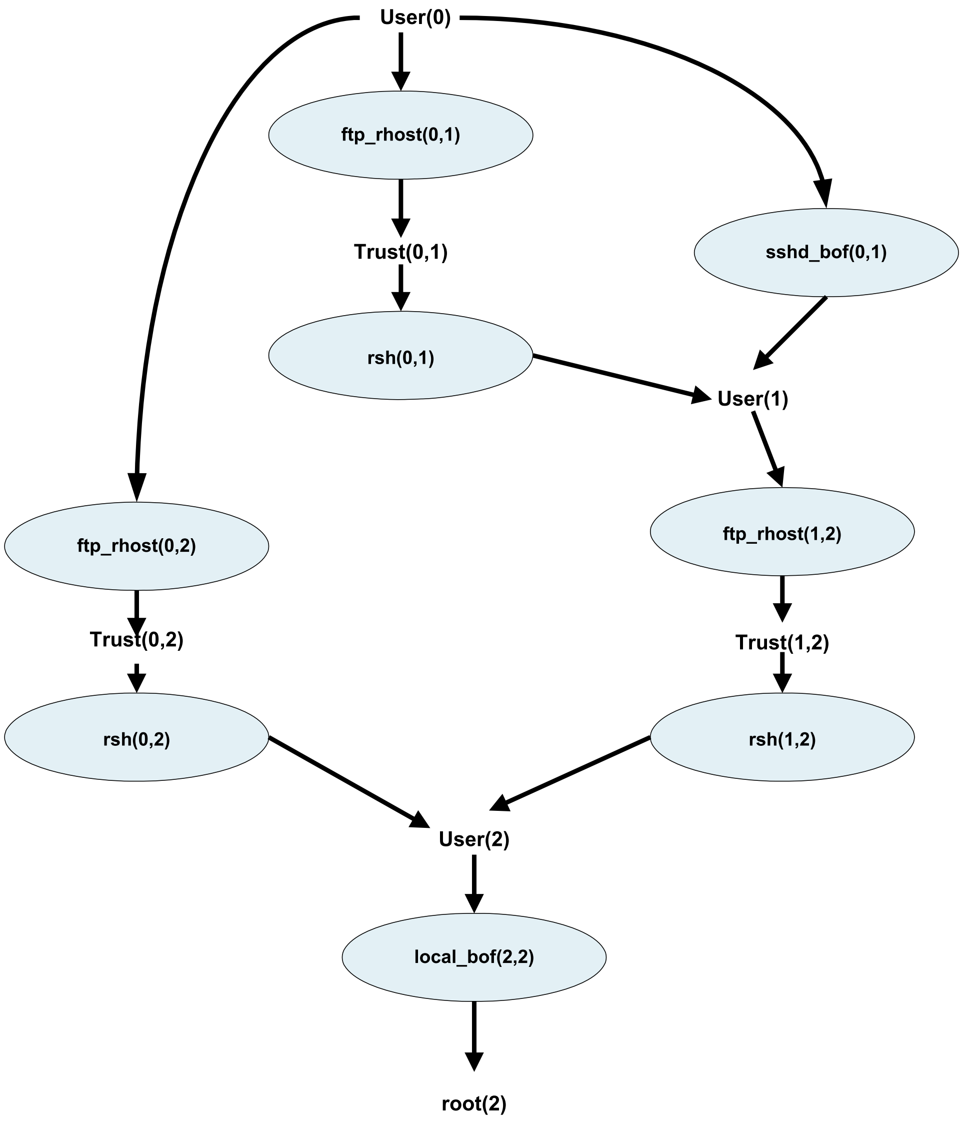

Fig. 2 Attack graph for example network

of different attack scenarios for a particular network. Traditionally it is a complete graph that has an inherent scalability problem and requires exponential time for generation. Moreover, complete attack graphs have redundant nodes and edges and have problems related to understandability and visual presentation. The problem of scalability and visualization have been removed with the introduction of minimal attack graphs which considers only those attack scenarios which terminate to a particular *goal*.

Various *model-checkers* and *custom algorithms* have been used to generate minimal attack graphs in polynomial time. But these *model-checkers* have the problem of *state space explosion* and are not efficient as far as representing exploits in realistic networks. Hence, the motivation of this paper is to propose a methodology to automatically generate minimal attack graphs using a *model-checker* from *artificial intelligence* domain, that does not suffer from *state space explosion* or *combinatorial explosion* problem. The complexity of generating minimal attack graph using the proposed methodology has been compared with previously reported works and has been found out to be efficient under some *special* cases.

In the present work, a methodology based on a technique from *artificial intelligence*, *Planner*, has been proposed to generate *minimal* attack graph. A detailed analysis of the time complexity of generating *minimal* attack graph using the proposed approach is also presented.

2.2 Planner

Artificial Intelligence has a major impact on the definition and development of most of the present day *Environment Decision Support Systems (EDSS)*. The present work on generation *minimal* attack graphs require making a number of decisions before its construction. Hence it is natural to attempt to integrate and enhance these decision outcomes using *artificial intelligence*.

Planner [17, 18] is a special purpose search algorithm in *artificial intelligence* domain for finding out a solution within a large state space. In this work, a variant of *Planner*, called *SGPlan*, is used for finding the attack paths. *Initial state*, *goal* state and the state transition operators are provided as input to the *Planner*. The input specifications are written in *PDDL* [19] (Planning Domain Definition Language) in two files viz. *domain.pddl* and *fact.pddl*. The *domain.pddl* contains un-instantiated predicates and state transition operators. These un-instantiated predicates are initialized by real world entities using a number of *objects* and *STRIPS* operators [19] to represent initial state and goal state in the *fact.pddl*. Appropriate changes in the *fact.pddl* enables the *Planner* to discard the previous plan and search for the new plan.

The *Planner* begins its execution from the initial state with a graph based representation called *plangraph*. The *plangraph* is generated starting from the initial state and successive application of state transition operators. It operates over all instances in the domain and maps states and goals into *actions* [20]. The generation of *plangraph* consumes the major amount of time in the entire attack path identification process. With n number of objects, m number of STRIPS operators each having maximum k number of constant formal parameters, the generation time for a t-level *plangraph* will be polynomial as maximum generated nodes in any action level will be $O(mn^k)$ [17]. The motivation behind selecting *Planner* as a technique for generating *attack paths* are due to the following advantages it provides:

– Prunes unnecessary actions from the system and finds the *shortest path*.
– Allows addition of actions to the plan where ever and whenever they are required.
– Uses richer input language, *PDDL*, to express complex state space domains relatively easier than custom-built analysis engines.
– Allows modeling of network, host, vulnerability, exploits, and connectivity relationships in a more realistic fashion.

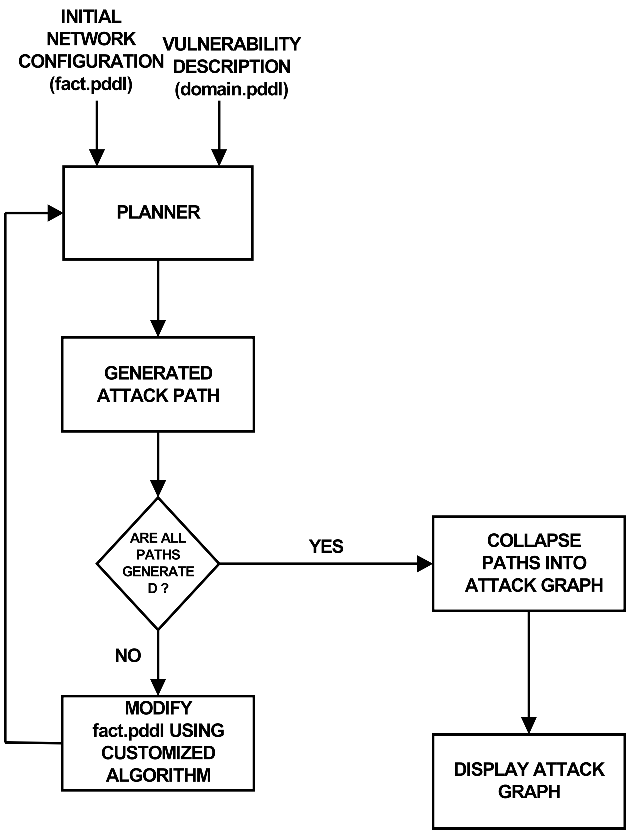
INITIAL NETWORK CONFIGURATION (fact.pddl)
VULNERABILITY DESCRIPTION (domain.pddl)
PLANNER
GENERATED ATTACK PATH
ARE ALL PATHS GENERATE D ?
YES
NO
COLLAPSE PATHS INTO ATTACK GRAPH
MODIFY fact.pddl USING CUSTOMIZED ALGORITHM
DISPLAY ATTACK GRAPH

Fig. 3 Flow chart showing *Planner* actions

– It is based on explicit assumption of *monotonicity* which removes backtracking from attack graph and reduces the generation time from exponential to polynomial.
– *Planner* generates acyclic paths, combining which gives a minimal attack graph.

3 Generation of minimal attack graph using planner

In this section, an approach to generate minimal attack graph using *Planner* has been proposed. The reasons behind preferring generation of minimal attack graph over complete attack graph are as follows:

– Minimal attack graph does not contain redundant nodes and edges; this enables the network administrators to have a *better visualization and understanding of different attack scenarios for a network*.

In the present work, the functional mechanism for *Planner* has been depicted in Fig. 3. It starts with the assumption that the *initial network configuration* and the *vulnerability analysis* have been done apriori and are input to the *Planner* in form of *domain* and *fact* files written in *PDDL* [19]. With the *initial network configurations, connectivity relationships, and vulnerability descriptions*, a shortest attack path is generated. To generate other shortest attack paths, the *fact.pddl* file has to be properly modified. Finally, when all the attack paths are generated, they are collapsed to form a minimal attack graph. In nutshell, the overall mechanism is given below:

1. Initial network configuration and description of the *exploits* (in form of *domain* and *fact* files) are input to *Planner* to generate a shortest attack path.
2. Customized *attack path enumeration* algorithm does automatic modification of *fact.pddl* to generate the next shortest attack path or no path.

Table 3 System characteristics

Host	Services	Ports	Vulnerabilities	CVE-IDs	Operating System
H0	IIS Web Service	80	IIS buffer overflow	CVE-2002-0364	Windows NT 4.0
H1	ftp	21	ftp rhost overwrite	CVE-2008-1396	
	ssh	22	ssh buffer overflow	CVE-1999-1455	Windows 2000 SP1
	rsh	514	rsh login	CVE-1999-0180	
H2	Netbios-ssn	139	Netbios-ssn nullsession	CVE-2003-0661	Windows XP SP2
	rsh	514	rsh login	CVE-1999-0180	
H3	LICQ	5190	LICQ-remote-to-user	CVE-2001-0439	
	Squid Proxy	80	squid-port-scan	CVE-2001-1030	Red Hat Linux 7.0
	Mysql DB	3306	local-setuid-bof	CVE-2006-3368	

3. Customized *attack graph building* algorithm collapses the generated shortest attack paths to generate a minimal attack graph.

Scaling of attack graphs for large network has been a problem reported in a number of literature surveys [11–13]. Most of the authors claim that present techniques of attack graph generation will encounter serious scalability issues once the number of hosts in a network increases to the order of thousand. But, if we observe the problem from a practical enterprise or academic network point-of-view, we will find that it is typically divided into zones, subzones and so on, each with different firewall and access policies forming separate administrative domains. Theoretically, we may have a network zone that consists of thousands of hosts, but in reality, we will hardly find one such. These network zones consist of a *finite* number of hosts and also the number of vulnerabilities existing on them is upper-bounded by a constant. Therefore, time required for encoding the domain file for a particular zone will be proportional to the number of existing generic vulnerabilities. If new vulnerabilities are detected, they can be formally encoded into the domain file in a linearly incremental fashion. Even for encoding the fact file, we do not have to enter N^2 number of connectivity relationships (where N is the number of hosts in the zone) due to restrictive firewall policies. Hence, generating attack graphs for network zones or subzones will not generate too many state variables to cause combinatorial explosion problem. Therefore, if separate attack graphs are generated for each zone and finally they are merged by some heuristics, by modifying the state variables appropriately, a large cumulative attack graph may be obtained. Hence, the present approach may be perceived as a methodology to efficiently generate attack graphs for a network zone in which the number of hosts is bounded by some constant.

In the following Section, the proposed methodology has been explained with the help of a case study.

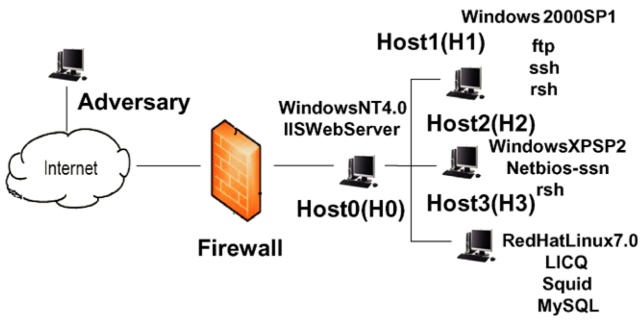

Fig. 4 Test network (*TEST-NET*)

Table 4 Connectivity-limiting firewall policies

Host	Attacker	H0	H1	H2	H3
Attacker	All	Yes	None	None	None
H0	None	All	All	All	All
H1	None	Yes	All	All	All
H2	None	Yes	All	All	All
H3	None	Yes	All	All	All

3.1 Case study

A network similar to [21] has been considered (refer to Fig. 4) as the test network (*TEST-NET*) which consists of *four* hosts viz. *Host*0 ($H0$), *Host*1 ($H1$), *Host*2 ($H2$), and *Host*3 ($H3$). $H3$ is taken as the target machine or *goal* and the MySQL[6] database running on that machine is the critical resource. The system characteristics of the hosts in *TEST-NET* are composed in Table 3. These data are available in Nessus, NVD,[7] Bugtraq.[8] Each *generic* vulnerability present in Table 3 has a set of preconditions and effects [21,

[6]http://www.mysql.com.

[7]http://nvd.nist.gov/.

[8]http://www.securityfocus.com/archive/.

22]. The preconditions and effects of one of the *generic* vulnerabilities viz. *IIS buffer overflow* is given below:

– **Preconditions**:

1. IIS Web Service running on target
2. IIS buffer overflow vulnerability exists
3. Attacker's privilege on target >= user
4. Attacker is able to access some services (in transport layer) running on target.

– **Effects**:

1. IIS Web Service is disabled on target
2. Attacker gains root level privilege on target

The firewall in *TEST-NET* (refer to Fig. 4) allows external hosts to connect to IIS web service running on port 80 on $H0$. The connection to all other ports are blocked. The internal hosts are allowed to connect on any port within the network. The connectivity limiting firewall policy is presented in Table 4. In Table 4, *All* indicates that a source host may connect to any port on a destination host and *None* signifies that the source machine is prevented from accessing any port on the destination machine. Depending upon connectivity limiting firewall policies, each *generic* exploit has some instances which are referred to as *instantiated* exploits [9]. Some of the *instantiated* exploits are as follows:

– *IIS_bof(0,0)—IIS-buffer-overflow* exploited from *Host*0 on *Host*0.
– *ftp_rhosts (0,1)—rsh* trust from *Host*0 to *Host*1.
– *squid_port_scan (1,3)—squid-port-scan* done from *Host*1 on *Host*3.
– *LICQ_remote_to_user (1,3)—LICQ-remote-to-user* exploit is used by *Host*1 to obtain user level privilege on *Host*3.
– *local_setuid_bof (3,3)*—Root privilege is obtained on *Host*3 by exploiting local buffer overflow exploit.
– *netbios_ssn_nullsession (1,2)*—User level privilege is obtained on *Host*2 from *Host*1 by executing *netbios-ssn-nullsession* exploit.

3.2 Identification of attack path using GraphPlan

GraphPlan, a variant of *Planner*, is a search algorithm which finds out solution within a large state space. Initial network configuration, attacker's objective, and exploits are considered as inputs to the *GraphPlan*. In this work, *SGPlan 5.2.2*,[9] a variant of *GraphPlan*, is used as an attack path identification component. *SGPlan* has been preferred to other variants of *GraphPlan* viz. *LPG-td*,[10] *Metric-FF*,[11] as it supports numeric predicates or fluents, derivative predicates, and durative predicates [23].

3.2.1 Domain *and* fact *files*

As mentioned in Sect. 2.2, *Planner* requires *two* files viz. *domain.pddl* and *fact.pddl* for its operation. An instance of *domain.pddl* [24, 25] for the test network (refer to Fig. 4) is given in Table 5. The *domain.pddl* encodes the following:

– **Network configurations**—depicting the services running on hosts, for e.g. *ftp ?H* etc.
– **Vulnerability instances**—indicating the type of vulnerabilities present, for e.g. *ftp_rhost_overwrite ?H* etc.
– **Functions**—e.g. *has_priv* to assign privilege levels.
– **Exploit descriptions**—encoding done in terms of *action rule* specification that has four components: *intruder precondition, intruder effect, network preconditions, and network effects*

The *fact.pddl* encodes various network *objects* that includes the hosts, the attacker, the firewall etc. An instance of the *fact.pddl* [24, 25] is given in Table 6. The *fact.pddl* encodes the following attributes:

– **Objects**—includes different network objects viz. hosts, firewall etc.
– **Numerical predicates**—with respect to the functions defined in *domain.pddl*, e.g. *(=(root_priv) 3)* which means *root* privilege has been assigned a value of 3.
– **Initial network configuration**—includes services running on hosts ((*IIS_web_service Host0*), transport layer connectivities (*ssh_port_connectivity Host2 Host1*), and application layer connectivities (*netbios_apps_connectivity Host1 Host2*)
– ***Goal* condition**—given as *(:goal (and (= (has_priv Attacker Host3) 3)))*.

SGPlan uses *domain.pddl* and *fact.pddl* to generate single shortest attack path. Systematic modification of *fact.pddl* enables *SGPlan* to identify alternate shortest attack paths. It depends on the administrator's discretion about which network configurations should be changed to generate newer attack paths. Modification of *fact.pddl* is done by disabling a service running in one of the hosts, or a connectivity between a pair of hosts by placing a *double-semicolon (;;)* before that predicate. From the given *domain.pddl* and *fact.pddl*, the shortest attack path generated by *SGPlan* is as follows.

```
; Time 0.00 ; ParsingTime 0.00 ; NrActions 4

;MakeSpan ; MetricValue ; PlanningTechnique

Modified-FF(enforced hill-climbing search)
  as the subplanner

0.001:(IIS-BUFFOVFLW ATTACKER ATTACKER
       HOST0)[1]
```

[9] http://manip.crhc.uiuc.edu/programs/SGPlan/sgplan5.html.

[10] http://www.zeus.ing.unibs.it/lpg/.

[11] http://www.members.deri.at/~joergh/metric-ff.html.

Table 5 Domain.pddl for *TEST-NET*

```
(define(domain attackgraph)
(:requirements :strips :fluents :equality)
(:predicates (IIS_web_service ?H)
(ftp ?H)
(ssh ?H)
(rsh ?H)
(netbios_ssn ?H)
(LICQ_chat_service ?H)
(squid_proxy ?H)
(IIS_bof ?H)
(ftp_rhost_overwrite ?H)
(rsh_login ?H)
(ssh_bof ?H)
(netbios_ssn_nullsession ?H)
(LICQ_remote_to_user ?H)
(local_setuid_bof ?H)
(IIS_port_connectivity ?S ?T)
(ftp_port_connectivity ?S ?T)
(ssh_port_connectivity ?S ?T)
(squid_port_connectivity ?S ?T)
(LICQ_port_connectivity ?S ?T)
(rsh_port_connectivity ?S ?T)
(netbios_port_connectivity ?S?T)
(IIS_apps_connectivity ?S ?T
(ftp_apps_connectivity ?S ?T)
(ssh_apps_connectivity ?S?T)
(squid_apps_connectivity?S?T)
(LICQ_apps_connectivity?S?T)
(rsh_apps_connectivity?S ?T)
(netbios_apps_connectivity?S?T))
(IIS_apps_connectivity ?S ?T)
(ftp_apps_connectivity ?S ?T)
(ssh_port_connectivity ?S ?T)
(squid_port_connectivity ?S ?T)
(LICQ_apps_connectivity?S?T)
(rsh_apps_connectivity ?S?T)
(netbios_apps_connectivity?S?T))
(:functions (has_priv ?A ?H)
(root_priv)
(user_priv)
(none_priv))
(:functions
(port_scan ?A ?H)
(port_scan_not_done))
(port_scan_done)
(:action IIS-buffovflw
:parameters
(?A
?S
?T)
:precondition
(and (>=(has_priv ?A?S)(user_priv))(IIS_web_service?T)
(IIS_port_connectivity ?S ?T) (IIS_bof?T(<(has_priv ?A?T)
(root_priv))))
:effect
(and (not(IIS_bof ?T)) (assign (has_priv ?A?T)(root_priv))
(not (IIS_web_service ?T)))
)
```

```
1.002:(SQUID-PORT-SCAN ATTACKER HOST0
       HOST3)[1]

2.003:(LICQ-REMOTE-TO-USER ATTACKER HOST0
       HOST3)[1]

3.004:(LOCAL-SETUID-BUFFOVRFLW ATTACKER
       HOST3)[1]
```

SGPlan generated attack path may be represented in the following way:

$$
\begin{aligned}
\textit{Attacker} &\rightarrow \textit{IIS\_bof}(Att, H0)\\
&\rightarrow \textit{squid\_port\_scan}(H0, H3)\\
&\rightarrow \textit{LICQ\_remote\_to\_user}(H0, H3)\\
&\rightarrow \textit{local\_setuid\_bof}(H3, H3).
\end{aligned}
$$

For generating an alternate attack path, the transport layer connectivity between *Host*0 and *Host*3 on *Squid_proxy* service is disabled (refer to Table 7). This is done by placing a *double-semicolon (;;)* before the predicate *squid_port_connectivity (Host0, Host3).*

SGPlan will discard the previous solution and generate the following *shortest* attack path.

```
; Time 0.00 ; ParsingTime 0.00
; NrActions 5 ;

MakeSpan ;MetricValue ; PlanningTechnique

Modified-FF(enforced hill-climbing search)
  as the subplanner

0.001:(IIS-BUFFOVFLW ATTACKER ATTACKER
       HOST0)[1]

1.002:(SSH-BUFFOVFLW ATTACKER HOST0
       HOST1)[1]

2.003:(SQUID-PORT-SCAN ATTACKER HOST1
       HOST3)[1]
```

Table 6 Fact.pddl for *TEST-NET*

(define (problem Attack)	(IIS_bof Host0)
(:domain attackgraph)	(ssh_bof Host1)
(:objects	(ftp_rhost_overwrite Host1)
Host0	(rsh_login Host1)
Host1	(netbios_ssn_nullsession Host2)
Host2	(LICQ_remote_to_user Host3)
Host3	(local_setuid_bof Host3)
Attacker	(IIS_port_connectivity Attacker Host0)
)	(ssh_port_connectivity Host0 Host1)
(:init	(ssh_apps_connectivity Host0 Host1)
(= (has_priv Attacker Attacker) 3)	(ssh_port_connectivity Host2 Host1)
(= (has_priv Attacker Host0) 1)	(ssh_apps_connectivity Host2 Host1)
(= (has_priv Attacker Host1) 1)	(ssh_port_connectivity Host3 Host1)
(= (has_priv Attacker Host2) 1)	(ssh_apps_connectivity Host3 Host1)
(= (has_priv Attacker Host3) 1)	(ftp_port_connectivity Host0 Host1)
(= (root_priv) 3)	(ftp_apps_connectivity Host0 Host1)
(= (user_priv) 2)	(ftp_port_connectivity Host2 Host1)
(= (none_priv) 1)	(ftp_apps_connectivity Host2 Host1)
(= (port_scan Attacker Host3) 0)	(ftp_port_connectivity Host3 Host1)
(= (port_scan_not_done) 0)	(ftp_apps_connectivity Host3 Host1)
(= (port_scan_done) 1)	(netbios_port_connectivity Host0 Host2)
(IIS_web_service Host0)	(netbios_apps_connectivity Host0 Host2)
(ssh Host1)	(netbios_port_connectivity Host1 Host2)
(ftp Host1)	(netbios_port_connectivity Host1 Host2)
(rsh Host1)	(netbios_port_connectivity Host1 Host2)
(netbios_ssn Host2)	(squid_port_connectivity Host0 Host3)
(squid_proxy Host3)	(squid_port_connectivity Host1 Host3)
(LICQ_chat_service Host3)	(squid_port_connectivity Host2 Host3)
	(LICQ_port_connectivity Host0 Host3)
	(LICQ_port_connectivity Host1 Host3)
	(LICQ_port_connectivity Host2 Host3))
	(:goal (and(= (has_priv Attacker Host3) 3))))

```
3.004:(LICQ-REMOTE-TO-USER ATTACKER HOST1
       HOST3)[1]

4.005:(LOCAL-SETUID-BUFFOVRFLW ATTACKER
       HOST3)[1]
```

SGPlan generated attack path may be represented in the following way:

$$
\begin{aligned}
Attacker &\rightarrow \mathit{IIS\_bof}(Att, H0)\\
&\rightarrow \mathit{ssh\_bof}(H0, H1)\\
&\rightarrow \mathit{squid\_port\_scan}(H1, H3)\\
&\rightarrow \mathit{LICQ\_remote\_to\_user}(H1, H3)\\
&\rightarrow \mathit{local\_setuid\_bof}(H3, H3).
\end{aligned}
$$

In the following section, a customized algorithm is presented which enumerates all possible acyclic attack paths by automatically modifying *fact.pddl*.

3.3 Attack path enumeration algorithm

Planner requires a *domain* and a *fact* file to generate shortest paths. The *domain.pddl* file (refer to Table 5) encodes *network configurations, vulnerability instances, functions,* and *exploit descriptions*. The *fact.pddl* encodes *network objects, numerical predicates, initial network configuration,* and *goal condition*. *Planner* generates single *shortest* path on each run if *fact.pddl* is properly modified. However, to generate all possible solutions (shortest paths), there is a need for a customized algorithm that will use *Planner* as a lower level module. Since each solution is an attack path, a set of all acyclic paths may be obtained. A customized *attack path enumeration* algorithm (refer to Algorithm 1) given in [26] is used in this paper. It executes *Planner* at the low-level by performing automatic modification of *fact.pddl*. In [26], the authors have defined various data structures used in these algorithms. A brief description of the algorithms has

Table 7 Modified Fact.pddl

```
(define (problem Attack)
(:domain attackgraph)
(:objects
Host0
Host1
Host2
Host3
Attacker
)
(:init
(= (has_priv Attacker Attacker) 3)
(= (has_priv Attacker Host0) 1)
(= (has_priv Attacker Host1) 1)
(= (has_priv Attacker Host2) 1)
(= (has_priv Attacker Host3) 1)
(= (root_priv) 3)
(= (user_priv) 2)
(= (none_priv) 1)
(= (port_scan Attacker Host3) 0)
(= (port_scan_not_done) 0)
(= (port_scan_done) 1)
(IIS_web_service Host0)
(ssh Host1)
(ftp Host1)
(rsh Host1)
(netbios_ssn Host2)
(squid_proxy Host3)
(LICQ_chat_service Host3)
(IIS_bof Host0)
(ssh_bof Host1)
(ftp_rhost_overwrite Host1)
(rsh_login Host1)
(netbios_ssn_nullsession Host2)
(LICQ_remote_to_user Host3)
(local_setuid_bof Host3)
(IIS_port_connectivity Attacker Host0)
(ssh_port_connectivity Host0 Host1)
(ssh_apps_connectivity Host0 Host1)
(ssh_port_connectivity Host2 Host1)
(ssh_apps_connectivity Host2 Host1)
(ssh_port_connectivity Host3 Host1)
(ssh_apps_connectivity Host3 Host1)
(ftp_port_connectivity Host0 Host1)
(ftp_apps_connectivity Host0 Host1)
(ftp_port_connectivity Host2 Host1)
(ftp_apps_connectivity Host2 Host1)
(ftp_port_connectivity Host3 Host1)
(ftp_apps_connectivity Host3 Host1)
(netbios_port_connectivity Host0 Host2)
(netbios_apps_connectivity Host0 Host2)
(netbios_port_connectivity Host1 Host2)
(netbios_port_connectivity Host1 Host2)
(netbios_port_connectivity Host1 Host2)
;;(squid_port_connectivity Host0 Host3)
(squid_port_connectivity Host1 Host3)
(squid_port_connectivity Host2 Host3)
(LICQ_port_connectivity Host0 Host3)
(LICQ_port_connectivity Host1 Host3)
(LICQ_port_connectivity Host2 Host3))
(:goal (and(= (has_priv Attacker Host3) 3))))
```

also been given. In the present work, a detailed description of the working of the algorithms and their complexity analysis have been presented.

3.3.1 Description of the algorithm

The *attack path enumeration* algorithm automatically runs *Planner* in each iteration to generate either a new path, or repeats a previously generated path, or generates no path. Its major function includes automatic modification of *fact.pddl* files. In the process, it blocks one or more service(s) so that a new solution (*attack path*) gets generated.

The inputs to this *attack path enumeration* algorithm are *domain.pddl*, and *fact.pddl*. A 2-D array, *Path*, has been declared to store newly generated attack paths in form of an adjacency matrix. Each *action* defined in *domain.pddl* is declared as a structure (called *node*). Within a *node* structure, one of the fields is *Critical* (type *Boolean*). Depending upon whether a node is *critical* or not, 1 or 0 is assigned respectively. The definition for *critical* and *non-critical* node are as follows:

– *Critical* nodes: Nodes in an *attack path* which need to be blocked invariably to generate another attack path.
– *Non-Critical* nodes: Nodes which are dependent on one or more critical nodes' effect such that if they are not executed, the corresponding preconditions are not generated.

From the exploit descriptions given in [21, 22], it may be observed that *LICQ_remote_to_user* exploit is only applicable if an attacker has the port number on which this application is running. This will be possible if he successfully executes the exploit *squid_port_scan* to obtain the port numbers of different applications. Therefore, execution of *LICQ_remote_to_user* exploit must be preceded

```
Input: Domain.pddl, Fact.pddl
Output: An exhaustive set of attack paths Path
while true do
    run planner to obtain path $i$;
    if no path found then
        while Stopped is not empty do
            service ← Pop(Stopped);
            restart service corresponding to service;
            $idx1$ ← path to which service belongs;
            $idx2$ ← index in $Critical_{idx1}$ of service;
            if $idx2$ is not the last index in $Critical_{idx1}$
            then
                $idx2 \leftarrow idx2 + 1$;
                service ← $Critical_{idx1}[idx2]$;
                stop service corresponding to service;
                Push(service, Stopped);
            end
        end
        if Stopped is empty then
            $false\_run \leftarrow false\_run + 1$;
            if $false\_run = 2$ then
                print paths from Path;
                Exit;
            end
        end
    end
    if path $i$ already exists then
        service ← $Critical_i[0]$;
        stop service corresponding to service;
        Push(service, Stopped);
    end
    if path $i$ is new then
        add path $i$ to Path;
        restart all stopped services;
        Stopped ← $\Phi$;
        foreach node in path $i$ do
            if service of node is critical & $flag == 0$
            then
                Enqueue(node, Marker);
                add node to $Critical_i$;
                $flag = 1$;
            end
        end
        if Marker is not empty then
            node ← Dequeue(Marker);
            stop service of node;
            Push(service, Stopped);
        end
    end
end
```

Algorithm 1: Attack path enumeration algorithm

Table 8 Critical and non-critical exploits

Critical	Non-Critical
IIS_bof	rsh_login
ssh_bof	LICQ_remote_to_user
ftp_rhost_overwrite	local_setuid_bof
netbios_ssn_nullsession	squid_port_scan

by *squid_port_scan* which implies that the former is *non-critical*.

Similarly, the exploit description of *local_setuid_bof* requires only *user* level privilege as its precondition and gives the *root* privilege once it is successfully executed. Now, there are a number of *exploits* which generate *user* level privilege as their effects viz. *netbios_ssn_nullsession*, *LICQ_remote_to_user*, *rsh_login* and so on. Therefore, execution of *local_setuid_bof* must be preceded by all these exploits, which implies that the former is not *critical* i.e. this *node* cannot be reached unless its preceding *exploits* are successfully executed. Depending upon the set of *exploits* used in the present work, the *critical* and *non-critical* ones may be composed in Table 8.

The remaining fields of a *node* structure are as follows:

- *name* (character array): Contains the name of the *node* (i.e. an *exploit* or a service);
- *source* (integer): Host IDs viz. 0 for *Host*0, 1 for *Host*1;
- *target* (integer): Host IDs;
- *flag* (boolean): To track whether a *node*(or service) has already been blocked. If blocked, the value is 1, otherwise it is 0.

Data structures The following data structures have been used by *attack path enumeration* algorithm:

Marker (*type*: Queue): The *Marker* data structure is used to store the nodes of a particular attack graph. When a new attack path is generated, depending upon the value of the *critical* and *flag* fields, the nodes are inserted into this queue. Once the *nodes* are inserted, they are marked by means of the *flag* field so that the same node is not inserted again. To generate the next *attack path*, the service corresponding to the first element in the queue is dequeued and blocked. Here, blocked implies disabling the service by placing *double-semicolon (;;)* before the corresponding *predicate* in *fact.pddl* file. It is the top-level service which is required to be blocked for generation of a new *attack path*. This service is pushed in to a stack.

Stopped (*type*: Stack): This data structure maintains information about which services under a *top-level* service have been blocked. *Top-level* service implies a node which is hierarchically above a subset of nodes in an attack path. It keeps track of the services that are being stopped when a repeat path or no path is found. In such situations, the algorithm backtracks to a previous state by popping elements from *Stopped*.

Critical (*type*: 2-D Array): It maps the *critical* services corresponding to the generated paths. The algorithm checks the *critical* fields of the nodes that constitute a new *attack path* and then inserts them into the array. It comes into use when no path is generated. The path number of a previously repeated path is used as an index to access the *critical* services. These *critical* services are blocked one by one by pushing into *Stopped*.

false_run (*type*: Integer): Used as a flag to identify conditions when no path is generated even after a complete iteration of all necessary network modifications have been done. This condition occurs when the top-level service from *Stopped* has been popped and no path is still obtained. Under this scenario, the value of *false_run* is incremented to 1. The top level service is popped to allow the algorithm to search for any alternate path that includes the top-level service. If no such path is found, then *false_run* is incremented and the algorithm terminates by concluding no more unique path can be obtained.

Path (*type*: 2-D Array): This 2-D array is used to store the generated attack paths in an adjacency matrix format. The columns of the matrix contain the path number while the rows depict the nodes which are included in a particular attack path. A different data structure, such as, adjacency list, may also have been used if there exists space constraints. However, the underlying philosophy would have remain the same.

Logic used Customized *attack path enumeration* algorithm works on the logic that there might be some exploits which have the same *immediate predecessor exploit* as another exploit in another path. Such a scenario is illustrated in Fig. 5 from which the following observations can be drawn:

– *Exploit A* which depends on some other exploit (*Immediate predecessor*) which has other dependent exploits (*Exploit B*).
– If the basic network conditions for *Exploit A* are blocked, *Planner* will attempt to output a path that does not include *A*. This corresponds to pushing *Exploit A* into *Stopped*.
– This would generate a newer path since the preceding exploit (*Immediate predecessor*) on which some other exploit (corresponds to *Exploit B*) could depend is still active.
– Each such exploit should be blocked in an iterative fashion to open possibilities of finding newer paths. This would lead to a state where all the attack paths for the network have been enumerated.

However, the assumption, that blocking an exploit will always result in a new attack path, is not right. This is due

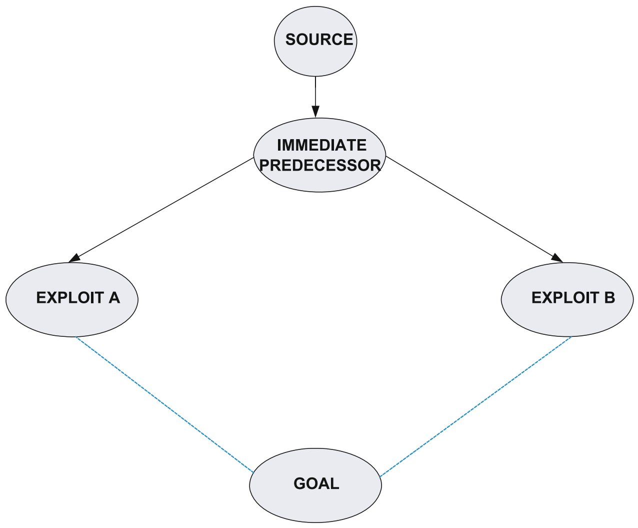

Fig. 5 Attack path generation logic

to the fact that *Planner* will always output the *shortest path* based on its algorithm. This may result in a repeat path instead of being a new path. If a repeat path is found, other exploits constituting the repeat path needs to be blocked so that the algorithm can proceed. This resorts to a form of *controlled brute force* method. There could also be scenarios where no path is found. This is possible if exploits have been blocked in some wrong order. This requires restating of blocked services (corresponding to the exploit) in the reverse order of their stopping. This in essence is a backtracking algorithm, which helps in escaping from infeasible solutions that might occur as the algorithm proceeds.

In the following subsection, an instance of execution of *attack path enumeration* algorithm has been presented which shows how the values of different data structures change with each run.

3.4 Execution of attack path enumeration algorithm

As stated in Sect. 3.2.1, the first minimal attack path that *Planner* outputs when the *fact.pddl* file is not modified is given by *Path*0.

Path 0:

$$\begin{aligned}&\mathit{Attacker} \rightarrow \mathit{IIS\_bof}(\mathit{Att}, H0)\\&\qquad \rightarrow \mathit{squid\_port\_scan}(H0, H3)\\&\qquad \rightarrow \mathit{LICQ\_remote\_to\_user}(H0, H3)\\&\qquad \rightarrow \mathit{local\_setuid\_bof}(H3, H3)\end{aligned}$$

Since the generated *minimal attack path* is new, it has to be inserted into the 2-D array *Path* in form of an adjacency matrix. This is shown in Table 9. As the *attack path enumeration* algorithm proceeds to generate the next *minimal attack*

Table 9 2-D array representation of generated attack path

Nodes/Paths	IIS(0, 0)	squid(0, 3)	LICQ(0, 3)	local_bof(3, 3)
0	1	1	1	1

paths, the changes in the values of different data structures take place in the following way:

1. *Stopped* $\leftarrow \phi$ [refer to Line-32 in Algorithm 1]
2. *Marker* $\leftarrow \{$*IIS_bof*$(0,0)$,*squid_port_scan*$(0,3)\}$ [refer to Line-35 in Algorithm 1]
3. *Critical* [*idx*1][*idx*2] [refer to Line-36 in Algorithm 1]

idx1/idx2	0	1
0	IIS_bof(0, 0)	squid_port_scan(0, 3)

4. *Dequeue(Marker)*$\Rightarrow$*Dequeue* $\leftarrow \{$*IIS_bof*$(0,0)\}$ [refer to Line-41 in Algorithm 1]
5. *Block* $\leftarrow \{$*IIS_bof*$(0,0)\}$ [corresponds to blocking "*IIS_port_connectivity Attacker Host0*" in *fact. pddl* (refer to Table 6)].
6. *Marker* $\leftarrow \{$*squid_port_scan*$(0,3)\}$
7. *Stopped* $\leftarrow \{$*IIS_bof*$(0,0)\}$ [refer to Line-43 in Algorithm 1]

With the current set of values, *attack path enumeration* algorithm makes necessary modifications in *fact.pddl* file and runs *Planner*. This results in **no path**.

To generate a path (either a new path or a repeat path), the algorithm performs the following operations:

1. *Service* $\leftarrow$ *Pop*(*Stopped*) $\Rightarrow$ *Pop*(*IIS_bof*$(0,0)$) [refer to Line-5 in Algorithm 1]
2. *Restart*(*IIS_bof*$(0,0)$) [corresponds to unblocking "*IIS_port_connectivity Attacker Host0*" in *fact.pddl* (refer to Table 6)]
3. *idx*1 $\leftarrow 0$, since the service (*IIS_bof*$(0,0)$) popped out belongs to *Path*0 [refer to Line-7 in Algorithm 1]
4. *idx*2 $\leftarrow 0$, since the service (*IIS_bof*$(0,0)$) popped out is the 0^{th} *critical node* in *Path*0 [refer to Line-8 in Algorithm 1]
5. *idx*2 $\leftarrow$ *idx*2 $+ 1 \Rightarrow$ *idx*2 $\leftarrow 1$ [refer to Line-10 in Algorithm 1]
6. *Service* $\leftarrow$ *Critical*[0][1] $\leftarrow \{$*squid_port_scan*$(0,3)\}$ [refer to Line-11 in Algorithm 1]
7. *Block* $\leftarrow \{$*squid_port_scan*$(0,3)\}$ [corresponds to blocking "*squid_port_connectivity Host0 Host3*" in *fact. pddl* (refer to Table 7)]
8. *Stopped* $\leftarrow \{$*squid_port_scan*$(0,3)\}$ [refer to Line-13 in Algorithm 1]

The above set of values is used by the algorithm to run *Planner*. This results in a new path given as:

Path 1:
Attacker $\rightarrow$ *IIS_bof*$(Att, H0)$
$\rightarrow$ *ssh_bof*$(H0, H1)$
$\rightarrow$ *squid_port_scan*$(H1, H3)$
$\rightarrow$ *LICQ_remote_to_user*$(H1, H3)$
$\rightarrow$ *local_setuid_bof*$(H3, H3)$

This newly generated path (*Path*1) is inserted into the 2-D array *Path* in the way depicted in Table 10.

The remaining *minimal attack paths* generated by the customized *attack path enumeration* algorithm are as follows:

Path 2:
Attacker $\rightarrow$ *IIS_bof*$(Att, H0)$
$\rightarrow$ *netbios_ssn_nullsession*$(H0, H2)$
$\rightarrow$ *squid_port_scan*$(H2, H3)$
$\rightarrow$ *LICQ_remote_to_user*$(H2, H3)$
$\rightarrow$ *local_setuid_bof*$(H3, H3)$

Path 3:
Attacker $\rightarrow$ *IIS_bof*$(Att, H0)$
$\rightarrow$ *ftp_rhost_overwrite*$(H0, H1)$
$\rightarrow$ *rsh_login*$(H0, H1)$
$\rightarrow$ *squid_port_scan*$(H1, H3)$
$\rightarrow$ *LICQ_remote_to_user*$(H1, H3)$
$\rightarrow$ *local_setuid_bof*$(H3, H3)$

Path 4:
Attacker $\rightarrow$ *IIS_bof*$(Att, H0)$
$\rightarrow$ *ssh_bof*$(H0, H1)$
$\rightarrow$ *netbios_ssn_nullsession*$(H1, H2)$
$\rightarrow$ *squid_port_scan*$(H2, H3)$
$\rightarrow$ *LICQ_remote_to_user*$(H2, H3)$
$\rightarrow$ *local_setuid_bof*$(H3, H3)$

Path 5:
Attacker $\rightarrow$ *IIS_bof*$(Att, H0)$
$\rightarrow$ *ftp_rhost_overwrite*$(H0, H1)$
$\rightarrow$ *rsh_login*$(H0, H1)$
$\rightarrow$ *netbios_ssn_nullsession*$(H1, H2)$
$\rightarrow$ *squid_port_scan*$(H2, H3)$

Table 10 2-D array representation of generated attack paths

Nodes/Paths	IIS(0, 0)	squid(0, 3)	LICQ(0, 3)	local_bof(3, 3)	ssh_bof(0, 1)	squid(1, 3)	LICQ(1, 3)
0	1	1	1	1	0	0	0
1	1	0	0	1	1	1	1

$\rightarrow LICQ\_remote\_to\_user(H2, H3)$

$\rightarrow local\_setuid\_bof(H3, H3)$

Path 6:

$Attacker \rightarrow IIS\_bof(Att, H0)$

$\rightarrow netbios\_ssn\_nullsession(H0, H2)$

$\rightarrow ssh\_bof(H2, H1)$

$\rightarrow squid\_port\_scan(H1, H3)$

$\rightarrow LICQ\_remote\_to\_user(H1, H3)$

$\rightarrow local\_setuid\_bof(H3, H3)$

Path 7:

$Attacker \rightarrow IIS\_bof(Att, H0)$

$\rightarrow netbios\_ssn\_nullsession(H0, H2)$

$\rightarrow ftp\_rhost\_overwrite(H2, H1)$

$\rightarrow rsh\_login(H2, H1)$

$\rightarrow squid\_port\_scan(H1, H3)$

$\rightarrow LICQ\_remote\_to\_user(H1, H3)$

$\rightarrow local\_setuid\_bof(H3, H3)$

3.5 Attack graph building algorithm

The minimal attack paths obtained by using *attack path enumeration* algorithm (refer to Algorithm 1) are collapsed to form the minimal attack graph. The *attack graph building* algorithm (refer to Algorithm 2) takes as input a set of attack paths given by the 2-D array *Path*, a set of *nodes* that constitute the paths. In each iteration, the algorithm finds out the nodes that constitute each attack path and draws directed arc among them. For subsequent iterations, if a node is obtained for the *first* time, the algorithm generates that node. For other occurrences, it draws either incoming or out-going edges from that node. Using Algorithm 2, the attack graph shown in Fig. 6 can be generated.

```
Input: 2-D array Path containing a set of paths, a set
       of nodes N
Output: Attack graph
1 Enumerate each node in $N$;
2 foreach $i = 1$ to TotalNumberOfPaths do
3     foreach $j = 1$ to TotalNumberofNodes do
4         if $Path[i][j] = 1$ then
5             Draw a directed edge from $i$ to $j$;
6         end
7     end
8 end
```

Algorithm 2: Attack graph building algorithm

In Fig. 6, the circles represents the *nodes* in the attack graph that contain the *exploits* which the attacker has utilized in different stages of the attack. The texts in the attack graph represent the *conditions* obtained by utilizing *exploits* or viceversa.

4 Proof of correctness

Model checkers have been used to generate attack graphs in [6, 8]. In [3, 7, 27], the authors have proposed formal languages to model networks, hosts, vulnerabilities, host connectivities, and exploits and provide them as input. The model checkers are explicitly build to handle large state spaces and generates counterexamples efficiently. In the present work, security of a computer network has been encoded in a finite state description and then assertions are written in temporal logic to violate a security condition "an attacker can never acquire certain rights on a given host".

Generation of attack graphs is essential for vulnerability analysis of a network of hosts. Manual generation of attack graph is tedious, error-prone, and impractical as the network size increases. The proposed approach attempts to automate this process. However, if the generation process has to be automated, it is required to ensure that the produced graph is both *exhaustive* as well as *succinct* [8]. These properties may be defined as follows:

1. *Exhaustive*: the generated attack graph covers all possible attacks.
2. *Succinct*: the generated attack graph contains only those conditions and exploits from which the intruder can reach his goal.

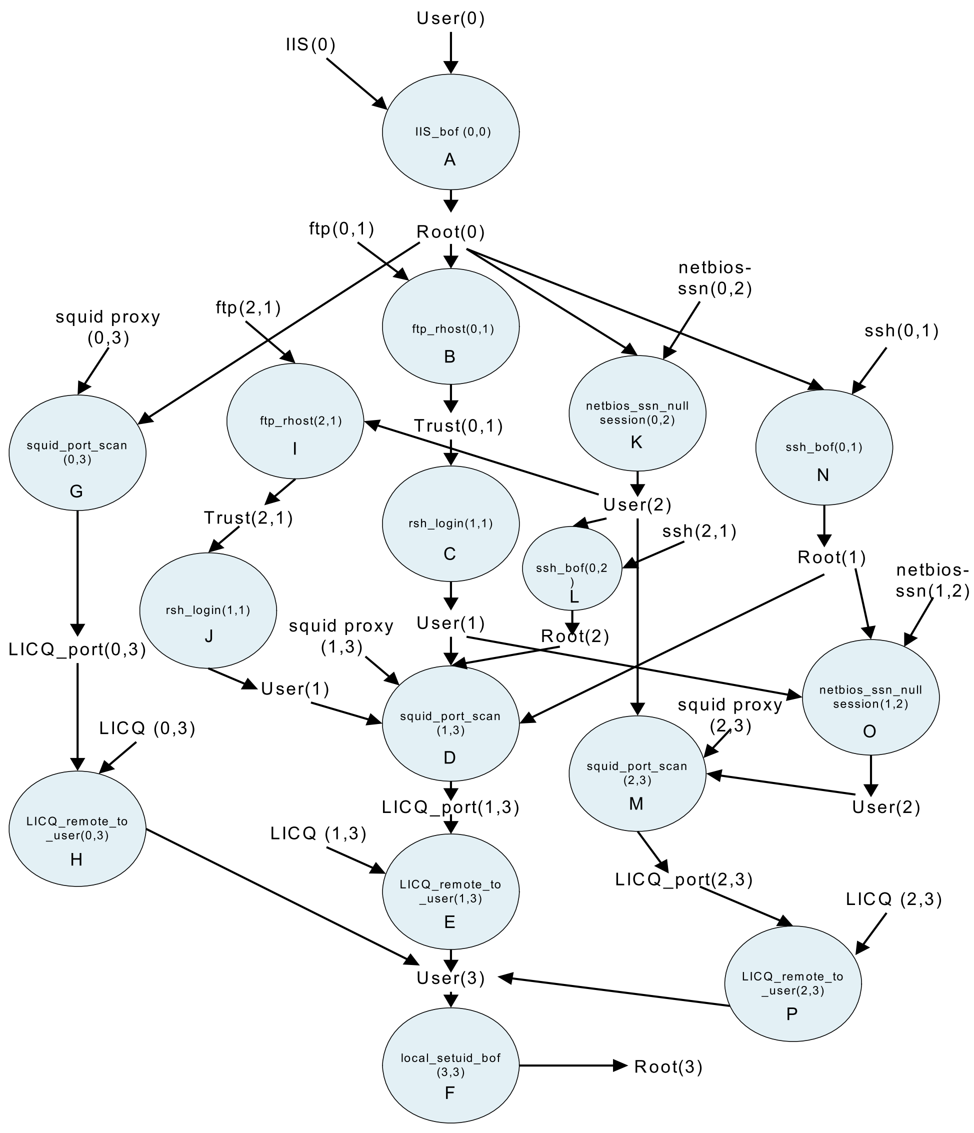

Fig. 6 Attack graph for *TEST-NET*

The correctness of the present work may be ensured, if it is proved that the attack graph generated by the proposed approach preserves both of these properties.

From the formal definition of attack graph (refer to Sect. 2.1.1) it is evident that the input model to the system is $(C, E, R_r, R_i, C_0, C_{goal})$, where C is the set of conditions, E is the set of exploits, R_r and R_i are transition relations, and C_0 is the set of initial states and C_{goal} is the set of goal states such that $C_0 \subseteq C$ and $C_{goal} \subseteq C$. An attack graph depicts different ways in which an intruder can break into a network and successfully reaches the *goal* condition. The property p that an intruder cannot reach the goal condition can be expresses as:

$$p = AG(\neg goal)$$

When this property is false, there are paths that lead from the initial conditions to the *goal* condition. The precise meaning of goal depends on the network. In the present work, goal condition implies to "obtaining root privilege on host H3". By restricting the domain and range of the transition relations R_r and R_i to C_{goal}, we obtain the transition relations R_r^p and R_i^p. Therefore, the attack graph is $AG = (E \cup C, R_r^p, R_i^p, C_0, C_{goal})$. The exhaustive and succinct properties of the generated attack graph can be formally stated by the following lemmas:

Lemma 1 (Exhaustive) *An execution sequence a of the input model* $(C, E, R_r, R_i, C0)$ *violates the property* $p = AG(\neg goal)$ *if and only if a is an attack path in the attack graph* $AG(E \cup C, R_r^p R_i^p, C_0, C_{goal})$.

Lemma 2 (Succinct) *A condition c of the input model* (C, E, R_r, R_i, C_0) *is in the attack graph AG if and only if there is an attack path in AG that contains c.*

The proofs of these lemmas are as follows:

Proof (Exhaustive) Let $a = c_0e_0c_1 \cdots c_{n-1}e_{n-1}c_n$ be a (finite) execution of the input model such that c_n is a goal condition. To prove that a is an attack path in AG, it is sufficient to show:

1. $c_0e_0 \subseteq R_r^p$
2. $e_{n-1}c_n \subseteq R_i^p$
3. $\forall 0 \le k \le n, c_k \in C$ and $e_k \in E$

Since the goal condition holds at $c_{n-1}e_{n-1}c_n$ and for all k there is a path from $c_{k-1}e_{k-1}c_k$ to $c_{n-1}e_{n-1}c_n$ in the input model, by definition every $c_{k-1}e_{k-1}c_k$ along a violates $AG(\neg goal)$. Therefore, by construction, every c_k is in C_{goal} and every $c_{k-1}e_{k-1} \in R_r^p$ and $e_{k-1}c_k \in R_i^p$. This satisfies (1), (2), and (3) immediately.

Suppose that $a = c_0e_0c_1 \cdots c_{n-1}e_{n-1}c_n$ is an attack path in the attack graph AG. By construction, all nodes and vertices of a are also nodes and vertices in the input model. Since a is an attack path, $c_0e_0 \in R_r^p$ and $e_{n-1}c_n \in R_i^p$. Therefore, $c_0e_0 \in R_r$ and $e_{n-1}c_n \in R_i$. So, a is an execution of the input model, and p is false in its final (goal) condition. It follows that a violates the property $AG(\neg goal)$. □

Proof (Succinct) By construction of attack path enumeration algorithm (refer Algorithm 1 in Sect. 3.3.1), all paths generated for the attack graph are reachable from an initial set of conditions/exploits, and all of them violates $AG(\neg goal)$. Therefore, for any such condition c and/or exploit e in the input model, there is path a_1 from an initial condition to c and/or e, and there is a path a_2 from c and/or e to a goal condition.

The concatenation of a_1 and a_2 is an execution a of the input model that violates $AG(\neg goal)$. By Lemma 1, a is an attack path in AG. Since a contains c, the proof is complete. □

5 Complexity analysis

Complexity analysis of the proposed approach requires consideration of *three* major aspects:

1. *Planner* complexity
2. Running time for *attack path enumeration* algorithm which executes *Planner* in each iteration
3. Complexity of *attack graph building* algorithm

5.1 Planner complexity

In Sect. 2.2, it has been stated that the time complexity for generating a t–level *plangraph* at any action–level is $O(mn^k)$ where the notations have their usual meanings [17]. In *domain.pddl*, it may be noted that we have used only three formal parameters viz. A, S, and T to represent *an attacker, a source, and a destination* respectively. These three parameters are sufficient to realize any *action*. So k in this case is a constant. Again, the number of *STRIP* operators that have been used for generating the attack paths is bounded by the number of *generic* vulnerabilities existing in the network. Therefore, the time complexity to generate attack paths in any action–level is $O(mn^3)$, where n is the number of objects used in the *fact.pddl* i.e., mainly the number of *hosts* in the network and m is the number of *generic* vulnerabilities present in the hosts of the network.

5.2 Complexity of attack path enumeration algorithm

A detail analysis of the *attack path enumeration* algorithm (refer to Algorithm 1) is shown below:

- Line 1: The number of times the **While-loop** will execute is proportional to q, which is the total number of generated attack paths.
- Line 2: Running *Planner* to generate a path requires $O(mn^3)$ time.
- Line 3: The **If-loop** checks if no path is generated in constant time.
- Line 4: The **While-loop**, which checks if *Stopped* is empty or not, will carry out the checking in constant time. In worst case, this checking may be done v number of times, where v is the total number of instantiated exploits.
- Line 5: Popping any service from *Stopped* takes constant amount of time.
- Line 6: Restarting a service essentially requires modification of *fact.pddl* file. The time to restart any service will be proportional to the size of the file. Considering *fact.pddl* file to have n number of objects, the size may be given by $O(n)$.
- Line 7: Assigning *idx*1 the path number, in worst case, may require examining q number of paths and v number of *instantiated exploits*. Hence, the total time to execute this statement is proportional to qv.
- Line 8: Finding out $idx2$ (given $idx1$) from 2-D array *Critical* will take at most v amount of time in worst case.
- Lines 9–14: The **If-loop**, which checks if $idx2$ is not the last index of *Critical*[*idx*1], will perform this checking in constant time. All the steps that are being executed within this loop runs in constant time except the one which stops services by modifying *fact.pddl* file. As stated above, this step will be executed in $O(n)$ time. Therefore, total time taken by the **While-loop** will be given by $= vc[c + n + qv + v + c\{c + n + c\}] = (nv + qv^2 + v) = v(n + qv + 1)$
- Lines 16–21: The **If-statement**, which determines if *Stopped* is empty, runs in constant time.
- Line 24: The **If-statement** which checks if path i already exists, requires checking of q number of paths each having at most v number of services in worst case. Therefore, number of comparisons is proportional to qv.

- Line 25: Determining the service present in *Critical*$[i][0]$ location takes place in constant time.
- Line 26–27: Stopping service again requires $O(n)$ time and pushing it into *Stopped* requires constant time. Therefore, the total time taken in this **If-statement** is given as $= qv(c+n+c) = qv(n+1)$.
- Line 29: The **If-statement**, which checks if path i is new, requires checking of q number of paths each having at most v number of services in worst case. Therefore, number of comparisons is proportional to qv.
- Line 30: Adding new path to 2-D array *Path* requires marking at most v number of instantiated exploits against the corresponding path number.
- Line 31: Restarting services will again yield a complexity of $O(n)$.
- Line 33: The **For-loop** examines at most v exploits in a particular path. Therefore, the number of iterations will be upper-bounded by v.
- Line 34: The **If-statement** which checks *critical* and *flag* fields for each *node*, in worst-case, runs for at most v number of times and each checking is done in constant time.
- Lines 35–37: Enqueuing a service into *Marker* takes constant amount of time. Similarly, adding the *node* in *Critical* array will again be done in constant time. Therefore, the overall time taken by the **For-loop** is given as $= v\{vc(c+c+c)\} = v^2$.
- Lines 40–44: The **If-statement** to find out whether *Marker* is empty will run in constant time. The operations within this **If-statement** i.e. dequeuing the first service, blocking it, and then pushing it into *Stopped* will take constant, $O(n)$, and constant time respectively.
- Therefore, total time taken by the **If-statement** which checks if path i is new, is given as $= qv(v+n+v^2+n)$

Hence, total running time for *attack path enumeration* algorithm, taking into account the running time for *Planner* can be given by (neglecting other lower-order terms of q and v): $q\{mn^3 + v(n+qv+1) + qv(n+1) + qv(v+n+v^2+n)\} = q(mn^3 + 3qvn + vn + qv^3 + 2qv^2) \Rightarrow (qmn^3 + q^2v^3)$.

5.3 Complexity of attack graph building algorithm

The algorithm for generating the attack graph (refer to Algorithm 2) is dependent upon the number of *nodes* that constitutes the attack paths and the number of paths which are generated. Therefore, the running time of *attack graph building* algorithm is always bounded by $O(qv)$. Therefore, the worst-case complexity of enumerating the *attack paths* and collapsing them to form an attack graph may be given as $O(qmn^3 + q^2v^3 + qv) \Rightarrow O(qmn^3 + q^2v^3)$.

5.4 Discussion

In [9], the computation in the initial marking phase of the algorithm grows as n^6v, where n is the number of *hosts*. In [8], the complexity of the graph generation algorithm is NP-complete. In the proposed approach, finding *shortest attack paths* and then combining these paths to generate *minimal attack graph* takes place in $O(qmn^3 + q^2v^3)$.

In a given directed graph with v number of nodes, the paths connecting a *source* node and a *sink* node may include $0, 1, 2, 3, \ldots, v$ number of intermediate nodes. Mathematically, total number of possible paths $(P(v))$ between a *source* and a *sink* in a graph may be given as,

$$P(v) = \binom{v}{0} + \binom{v}{1} + \binom{v}{2} + \binom{v}{3} + \cdots + \binom{v}{v-1} + \binom{v}{v} \quad (1)$$

Again, *binomial formula* for expanding any power of the sum $(x+y)$ is as follows:

$$(x+y)^v = \binom{v}{0}x^v + \binom{v}{1}x^{v-1}.y + \binom{v}{2}x^{v-2}.y^2 + \binom{v}{3}x^{v-3}.y^3 + \cdots + \binom{v}{v-1}x.y^{v-1} + \binom{v}{v}y^v \quad (2)$$

Putting $x = y = 1$ in (2), the *binomial expansion* formula is reduced as,

$$2^v = \binom{v}{0}1^v + \binom{v}{1}1^{v-1}.1 + \binom{v}{2}1^{v-2}.1^2 + \binom{v}{3}1^{v-3}.1^3 + \cdots + \binom{v}{v-1}1.1^{v-1} + \binom{v}{v}1^v \quad (3)$$

From (1) and (3), total number of possible paths between a *source* and a *sink* becomes,

$$P(v) = 2^v \quad (4)$$

Therefore, the time required $(T(n,v))$ for generation of *minimal* attack graph using *Planner* becomes,

$$T(n,v) = C_1.qmn^3 + C_2.q^2v^3 = C_1.2^v mn^3 + C_2.2^{2v}v^3$$

[using (4)]

In majority of well-managed networks, having large number of hosts, and stringent firewall policies, vulnerabilities on most of the hosts are usually patched. Owing to these, the attack graphs corresponding to these networks have limited connectivity and are sparse [16]. The attack paths in an attack graph depict different attack scenarios by combining the vulnerabilities existing on different hosts in the network to reach a particular *goal* node. But in real-world scenario, due to connectivity limiting firewall rules and difficulty in availability of exploits, actual attacks do not include all the vulnerabilities present in the network. This leads to the fact that the number of intermediate nodes (exploits) between *source* nodes and a *goal* node in an attack path is upper-bounded by a constant number of vulnerabilities. Intuitively, it can be concluded that the length of the longest attack path is $O(l)$, rather than $O(v)$, where l is a constant. Considering this observation, total number of attack paths that can be generated can be given as, $P(v) = 2^l$, i.e., $P(v) = \text{constant}$.

Using this argument in the expression for computing the time required to generate *minimal* attack graph,

$$\begin{aligned} T(n,v) &= C_1.2^v mn^3 + C_2.2^{2v} v^3 \\ &= C_1.2^l mn^3 + C_2^{2l}.v^3 \\ &= C_3.mn^3 + C_4.v^3 \\ &\Rightarrow O(mn^3 + v^3) \end{aligned}$$

In the above time complexity expression, m is the number of *STRIPS-operators* that is encoded in *domain.pddl* file and it is proportional to the number of *generic vulnerabilities* existing in the network. The relationship between *generic* and *instantiated* vulnerabilities for a given number of hosts in a network can be established by the following three examples.

– **Example 1**: Consider a network with *three* hosts viz., n_1, n_2, and n_3 each of which having exactly one *generic* vulnerability m_1, m_2, and m_3 respectively. Therefore, number of possible *instantiated* vulnerabilities(v) will be given as, $m_1(n_1,n_1)$, $m_1(n_1,n_2)$, $m_1(n_1,n_3)$, $m_2(n_2,n_2)$, $m_2(n_2,n_1)$, $m_2(n_2,n_3)$, $m_3(n_3,n_3)$, $m_3(n_3,n_1)$, and $m_3(n_3,n_2)$ which is proportional to n^2, where n is the number of hosts in the network.

– **Example 2**: Consider another network with *three* hosts viz., n_1, n_2, and n_3 each of which having arbitrary number of *generic* vulnerabilities. Let the vulnerabilities for host n_1 are m_{11}, m_{12}, vulnerabilities for host n_2 are m_{21}, m_{22}, and m_{23}, and those for host n_3 is m_{31}. Therefore, number of possible *instantiated* vulnerabilities are, $m_{11}(n_1,n_1)$, $m_{11}(n_1,n_2)$, $m_{11}(n_1,n_3)$, $m_{12}(n_1,n_1)$, $m_{12}(n_1,n_2)$, $m_{12}(n_1,n_3)$, $m_{21}(n_2,n_2)$, $m_{21}(n_2,n_1)$, $m_{21}(n_2,n_3)$, $m_{22}(n_2,n_2)$, $m_{22}(n_2,n_1)$, $m_{22}(n_2,n_3)$, $m_{23}(n_2,n_2)$, $m_{23}(n_2,n_1)$, $m_{21}(n_2,n_3)$, $m_{31}(n_3,n_3)$, $m_{31}(n_3,n_1)$, and $m_{31}(n_3,n_2)$.

  Hence, total number of possible *instantiated* exploits is proportional to the product $m \times n$.

– **Example 3**: Let a network consists of *three* hosts viz., n_1, n_2, and n_3 and each of them has as many number of *generic* vulnerabilities as the number of hosts. Let the vulnerabilities for host n_1 are m_{11}, m_{12}, and m_{13}. Host n_2 has the vulnerabilities m_{21}, m_{22}, and m_{23} and those for host n_3 are m_{31}, m_{32}, and m_{33}. Number of possible *instantiated* vulnerabilities are as follows, $m_{11}(n_1,n_1)$, $m_{11}(n_1,n_2)$, $m_{11}(n_1,n_3)$, $m_{12}(n_1,n_1)$, $m_{12}(n_1,n_2)$, $m_{12}(n_1,n_3)$, $m_{13}(n_1,n_1)$, $m_{12}(n_1,n_2)$, $m_{12}(n_1,n_3)$, $m_{21}(n_2,n_2)$, $m_{21}(n_2,n_1)$, $m_{21}(n_2,n_3)$, $m_{22}(n_2,n_2)$, $m_{22}(n_2,n_1)$, $m_{22}(n_2,n_3)$, $m_{23}(n_2,n_2)$, $m_{23}(n_2,n_1)$, $m_{21}(n_2,n_3)$, $m_{31}(n_3,n_3)$, $m_{31}(n_3,n_1)$, $m_{31}(n_3,n_2)$, $m_{32}(n_3,n_3)$, $m_{31}(n_3,n_1)$, $m_{31}(n_3,n_2)$, $m_{33}(n_3,n_3)$, $m_{31}(n_3,n_1)$, and $m_{31}(n_3,n_2)$,

  Therefore, total number of *instantiated* vulnerabilities will be $v = m \times n = n^2 \times n = n^3$.

The above time complexity expression can be tested against different cases to obtain the following variations:

- **Case 1**: Each host is having constant number of *generic* vulnerability. In this case, total number of *generic* vulnerabilities will be, $m = k \times n$, where k is an arbitrary constant. The number of *instantiated* vulnerabilities becomes $v = n \times m = kn^2$.

  Therefore, the time complexity to generate *minimal* attack graph becomes,

$$\begin{aligned} T(n,v) &= O(mn^3 + v^3) \\ &= O(kn^4 + (kn^2)^3) \\ &\Rightarrow O(n^4 + n^6) \\ &\Rightarrow O(n^6) \end{aligned}$$

- **Case 2**: The hosts in the network have *random* number of vulnerabilities. Total number of *generic* vulnerabilities is, $m = m_1 + m_2 + m_3 + \cdots + m_n$. In this work, the random variables are assumed to have *integer* values. Let X and Y be two independent discrete random variables with distribution functions $p_1(x)$ and $p_2(x)$ respectively. If $Z = X + Y$, then the distribution function $(p_3(x))$ for random variable Z will be given as [28]:

$$p_3(j) = \sum_k p_1(k).p_2(j-k) \tag{5}$$

  for $j = \ldots, -2, -1, 0, 1, 2, \ldots$.

  In this work, the distribution function to find the total sum of *generic* vulnerabilities (m) existing on n discrete hosts (using (5)) is as follows:

$$p_n(m) = \sum_{k,i=1}^{n} p_i(k).p_{n-i}(m-k) \tag{6}$$

  Considering common distribution function p on the integers to compute the sum of n independent random variables $S_n = X_1 + X_2 + X_3 + \cdots + X_n$, (6) becomes,

$$p(m) = \sum_{k=1}^{n} p(k).p(m-k) \tag{7}$$

where, $p_1(x) = p_2(x) = \cdots = p_n(x) = p(x)$.

Therefore, computing the distribution function to find out total number of *generic* vulnerabilities existing in n hosts requires $O(n^2)$ time.

The sum of n independent random variables having common distribution function p can be represented as [28],

$$S_n = S_{n-1} + X_n \tag{8}$$

Similarly, using (8), the sum of number of *generic* vulnerabilities existing over n hosts is given as,

$$M_n = M_{n-1} + m_n \tag{9}$$

Recursively solving (9) using $m_n = p$ and $M_0 = 0$, the following relation may be obtained,

$$M_n = n \times p \tag{10}$$

Therefore, computation of the total number of *generic* vulnerabilities present in a n host network yields a time complexity of $O(np) \Rightarrow O(n^3)$.

The number of *instantiated* vulnerabilities (v) can be obtained in $O(mn)$ time i.e., $O((np).p)$ or $O(n^4)$ time

Using the above relations in time complexity expression,

$$\begin{aligned} T(n, v) &= O(mn^3 + v^3) \\ &= O\big((n^3 \times n^3) + (n^4)^3\big) \\ &\Rightarrow O(n^6 + n^{12}) \\ &\Rightarrow O(n^{12}) \end{aligned}$$

The above case studies convey that for a *well-protected* and *large* network, if the number of *generic* vulnerabilities present in each host is bounded by a *constant*, complexity for generation of a *minimal* attack graph by the proposed approach produce better performance than the reported works in [8, 9]. *Case 2* suggests that if the number of vulnerabilities existing on the hosts is following a random distribution, the time complexity remains polynomial and is upper-bounded by $O(n^{12})$.

5.5 Comparison with related works

Literature survey on attack graphs show that researchers have used both custom algorithms [2–5] as well as formal methods [3, 6–8] to generate attack graphs. Formal methods typically involve representation of attacks, networks, vulnerabilities, and connectivities 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 breached. Attack graphs with one single goal as well as those with multiple goals have been generated for network security assessment. The present work aims at generating *minimal* attack graph i.e. the graph in which all attack paths terminate to a particular goal node. As shown in Sect. 5.4, the time complexity of generating minimal attack graphs for networks are: (i) $O(N^6)$, when each host contains a constant number of vulnerabilities, and (ii) $O(N^{12})$, when each host contains a random number of vulnerabilities. The results of research works which deal with generation of attack graphs that considers single goal are presented in Table 11. These results have been partially obtained from [29]. Most of the other reported works have either not analyzed their methodology of generating attack graphs or used other variants of attack graphs viz. *Multi-Prerequisite (MP) graph*, *Logical graph*, and so on. Some of the works have not presented algorithms to automate the usage of the model-checkers. In the present work, *Planner* takes $O(N^3)$ time to manually generate individual attack paths. Therefore, with respect to attack path enumeration, the present work thrives better than most of the previous works as far as running time is concerned. However, only when the attack graph generation process is automated, the total running time is attaining some higher-order polynomial value. Hence, there is a genuine trade-off between automatic generation of attack graph and scaling with respect to the size of the network.

One of the problems with generation of attack graphs has been gathering the requisite information. Modeling a correlated attack requires obtaining the *preconditions* and *postconditions* relevant to each exploit. But there is no publicly available database which maintains such a repository. Although some vulnerability databases exist, but they are either proprietary or do not contain the machine-readable details required to accurately generate the attack graphs. Hence, the present work deals with only those vulnerabilities/exploits whose descriptions are used in previous reported works in some form or other. Moreover, researchers claim that is it not possible to obtain reachability information by a single vulnerability scan of a network. This is because, in a network, firewalls contain numerous access control rules, network address translation (NAT) rules that represent a group of IP addresses. Therefore, a single scan from one IP address, either internal or external to the network, will show only a few of these rules. The present work addresses this problem by stating that attack graph is an attacker's perspective of the network which an administrator obtains by its accurate generation. The attack graph may be used for defending a network and its resources and adopting appropriate security measures. This is because, in reality, a network administrator may patch hundreds of loopholes in his network, but an adversary requires only one exploitable machine to penetrate into the network.

Table 11 Comparative study with related works

Paper	Approach	Results	Remarks
Ammann, 2002 [9]	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 [4]	Formal method to generate attack chaining trees.	Shows poor scaling results.	Generates full attack graph and finds out a "minimum cut set" where the goal cannot be reached if any single vulnerability is removed.
Jajodia, 2003 [1, 10]	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 [6]	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 [8]	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 [30]	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 Planner to generate shortest attack path from a given domain and fact file. Customized algorithms have been developed to automate the generation of attack graph by executing planner as a low-level module.	The complexity is: (1) $O(N^6)$ if the hosts have constant number of generic vulnerabilities, (2) $O(N^{12})$ if the number of generic vulnerabilities is randomly distributed.	Planner generates individual attack paths in $O(N^3)$ time. But if the mechanism is automated, the time complexity is attending some higher-order polynomial value. The usage of Planner facilitates exploration of large state space and also allows modeling of vulnerabilities in a more realistic manner.

Since, a network administrator is aware of the firewall policies and the access control rules of his network, he may gain the connectivity information and generate the attack graph for security assessment. Therefore, a basic assumption that the present work undertakes is that the information required for generating the attack graph for a network is complete and accurate and it has been gathered by someone, preferably the network administrator, who knows the firewall rules, connectivity relationships, and access control policies.

6 Conclusion

As computer networks continue to grow in size and complexity, they are becoming vulnerable against sophisticated cyber attacks. Such attack combines the vulnerabilities existing on different machines and are potentially more harmful than single-point attacks. Attack graph is a tool that provides a succinct representation of such correlated attacks and facilitates security analysis of a network. It consists of a number of attack paths which in essence are the attack scenarios. Each attack scenario is a logical succession of exploits where any exploit in the series lays the groundwork for subsequent exploits and forms a cause-effect relationship among themselves. The present work focusses on scalable representation by using the concept of *minimal* attack graph. *Minimal* attack graph consists of only successful attack paths, such that, each path terminates at a particular *goal* node. In this work, a method for enumerating *minimal* attack paths and then collapsing these paths to form a *minimal* attack graph has been proposed. *Planner*, a general-purpose search algorithm from *artificial intelligence* domain, has been deployed for efficient generation of *minimal* attack graph. It generates the attack graph in polynomial time, and unlike other *model-checkers*, it does not suffer from *combinatorial explosion* problem. The *domain* and *fact* files for the test network has been encoded in $PDDL$ and are given as input to *Planner*. Customized *attack path enumeration* algorithm runs *Planner* as a low-level module to generate minimal attack paths. In each iteration of the algorithm, automatic modification of *fact.pddl* file takes place. With this new *fact.pddl* either a new path, or a repeat path, or no path is generated. A detailed complexity analysis of the proposed methodology has been done. Analysis shows that the methodology is time efficient in terms of finding the attack paths and building the attack graph than some of the already reported works [8, 9] for *real-world, large-sized, well-protected* networks where the number of *generic* vulnerabilities on each host is bounded by a constant. However, the algorithm generates *minimal* attack graph in polynomial time even if the vulnerabilities are randomly distributed among the hosts present in the network.

Acknowledgements This work is partially supported by a research grant from the Department of Information Technology, Ministry of Communication and Information Technology, Government of India, under Grant No. 12(14)/09-ESD, dated 11-Jan-2010.

References

1. Noel S, Jajodia S, O'Berry B, Jacobs M (2003) Efficient minimum-cost network hardening via exploit dependency graph. In: Proceedings of 19th annual computer security applications conference (ACSAC 2003), Las Vegas, Nevada, pp 86–95
2. Phillips C, Swiler LP (1998) A graph-based system for network-vulnerability analysis. In: Proceedings of the workshop on new security paradigms (NSPW), Virginia, USA, pp 71–79
3. Tidwell T, Larson R, Fitch K, Hale J (2001) Modelling internet attacks. In: Proceedings of the second annual IEEE SMC information assurance workshop, United States Military Academy, West Point, New York. IEEE Press, New York, pp 54–59
4. Dawkins J, Hale J (2004) A systematic approach to multi-stage network attack analysis. In: Proceedings of the second IEEE internation information assurance workshop (IWIA '04), IEEE Computer Society, Washington, pp 48–56
5. Ortalo R, Deswarte Y, Kanniche M (1999) Experimenting with quantitative evaluation tools for monitoring operational security. IEEE Trans Soft Eng 25(5):633–650
6. Ritchey RW, Ammann P (2000) Using model checking to analyze network vulnerabilities. In: Proceedings of the 2000 IEEE symposium on security and privacy, Oakland, CA, pp 156–165
7. Templeton S, Levitt K (2001) A requires/provides model for computer attacks. In: Proceedings of the 2000 workshop on new security paradigms, Ballycotton, County Cork, Ireland. ACM Press, New York, pp 31–38
8. Sheynar O, Jha S, Wing JM, Lippmann RP, Haines J (2002) Automated generation and analysis of attack graphs. In: Proceedings of the 2002 IEEE symposium on security and privacy, pp 273–284
9. Ammann P, Wijesekera D, Kaushik S (2002) Scalable, graph-based network vulnerability analysis. In: Proceedings of CCS 2002: 9th ACM conference on computer and communications security, Washington, DC. ACM Press, New York, pp 217–224
10. Jajodia S, Noel S, O'Berry B (2005) Topological analysis of network attack vulnerability. In: Managing cyber threats: issues, approaches and challenges, vol V. Springer, New York, pp 247–266
11. Ammann P, Pamula J, Ritchey R, Street J (2005) A host-based approach to network attack chaining analysis. In: Proceedings of the 21st annual computer security applications conference (ACSAC), pp 72–84
12. Pamula J, Jajodia S, Ammann P, Swarup V (2006) A weakest-adversary security metric for network configuration security analysis. In: Proceedings of 2nd ACM workshop on quality of protection. ACM Press, New York, pp 31–38
13. Zhang T, Hu MZ, Li D, Sun L (2005) An effective method to generate attack graph. In: Proceedings of the international conference on machine learning and cybernetics (ICMLC), Guangzhou, China, pp 3926–3931
14. Ingols K, Lippmann R, Piwowarski K (2006) Practical attack graph generation for network defense. In: Proceedings of the 22nd annual computer security applications conference (ACSAC '06), pp 121–130
15. Wang L, Islam T, Long T, Singhal A, Jajodia S (2008) An attack graph-based probabilistic security metric. In: Proceedings of the international federation for information processing (IFIP). LNCS, vol 5094, pp 283–296
16. Wang L, Noel S, Jajodia S (2006) Minimum cost-network hardening using attack graphs. Comput Commun 29(18):3812–3824
17. Blum AL, Furst ML (1997) Fast planning through planning graph analysis. J Artif Intell 281–300
18. Bonet B, Geffner H (2001) Planning and control in artificial intelligence: A unifying perspective. Appl Intell 14:237–252
19. Fox M, Long D (2003) Pddl 2.1: An extension to pddl for expression temporal planning domains. J Artif Intell Res, 61–124
20. Martin M, Geffner H (2004) Learning generalized policies from planning examples using concept languages. Appl Intell 20:9–19
21. Sheynar O (2004) Scenario graphs and attack graphs. PhD thesis, Carnegie Mellon University, USA
22. Sheynar O, Wing JM (2004) Tools for generating and analyzing attack graphs. In: Proceedings of the workshop on formal methods for components and objects (FMCO), Leiden, The Netherlands, pp 344–371
23. Chen Y, Hsu C, Wah B (2006) Temporal planning using subgoal partitioning and resolution in sgplan. J Artif Intell Res, 323–369
24. Ghosh N, Ghosh SK (2009) An intelligent technique for generating minimal attack graph. In: Proceedings of the first workshop on intelligent security (security and artificial intelligence, SecArt 2009), 19th international conference on automated planning and scheduling (ICAPS'09), Thessaloniki, Greece, pp 42–51
25. Bhattacharya S (2008) Security risk management of local area network. Master's thesis, School of Information Technology, I.I.T Kharagpur
26. Ghosh N, Ghosh SK (2010) An intelligent approach for security management of an enterprise network using planner. Studies in computational intelligence, vol 275. Springer, Berlin, pp 187–214

27. Cuppens F, Ortalo R (2000) Lambda: A language to model a database for detection of attacks. In: Proceedings of the 3rd international workshop on the recent advances in intrusion detection (RAID). LNCS, vol 1907. Springer, Berlin, pp 197–216
28. Grinstead CM, Snell JL (1997) Introduction to probability. American Mathematical Society, Providence
29. Lippmann RP, Ingols IW (2005) An annotated review of past papers on attack graphs. Technical Report ESC-TR-2005-054, Lincoln Laboratory, Massachusetts Institute of Technology, USA
30. Swiler LP, Phillips C, Ellis D, Chakerian S (2001) Computer-attack graph generation tool. In: Proceedings of the 2nd DARPA information survivability conference & exposition (DISCEX II), vol II. IEEE Computer Society, Los Alamitos, pp 307–321

Nirnay Ghosh is pursuing PhD in Information Technology from School of Information Technology, Indian Institute of Technology (IIT), Kharagpur, India. Prior to this, he received MS in Information Technology from Indian Institute of Technology, Kharagpur, India and B.Tech in Computer Science & Engineering from West Bengal University of Technology (WBUT), India. His present area of research includes network security, attack graph analysis, cloud security.

S.K. Ghosh did his M.Tech and PhD in Computer Science & Engineering from the Indian Institute of Technology (IIT) Kharagpur, India. He is currently an Associate Professor at the School of Information Technology, IIT Kharagpur. Before joining IIT Kharagpur, he worked for Indian Space Research Organization in the area of Satellite Remote Sensing and GIS. His research interests include Network Security and Spatial Web Services. He has over 50 research papers in reputed conferences and journals.