Network Vulnerability Analysis Through Vulnerability Take-Grant Model (VTG)

Conference Paper *in* Lecture Notes in Computer Science · December 2005

DOI: 10.1007/11602897_22 · Source: DBLP

Hamid Reza Shahriari
Amirkabir University of Technology
38 PUBLICATIONS **250** CITATIONS
SEE PROFILE

Reza Sadoddin
University of New Brunswick
9 PUBLICATIONS **135** CITATIONS
SEE PROFILE

Rasool Jalili
Sharif University of Technology
128 PUBLICATIONS **707** CITATIONS
SEE PROFILE

Network Vulnerability Analysis Through Vulnerability Take-Grant Model (VTG)

Hamid Reza Shahriari, Reza Sadoddin, Rasool Jalili, Reza Zakeri, and Ali Reza Omidian

Network Security Center, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran*
{shahriari, rzakery, omidian}@mehr.sharif.edu
saededdi@ce.sharif.edu, jalili@sharif.edu

Abstract. Modeling and analysis of information system vulnerabilities helps us to predict possible attacks to networks using the network configuration and vulnerabilities information. As a fact, exploiting most of vulnerabilities result in access rights alteration. In this paper, we propose a new vulnerability analysis method based on the Take-Grant protection model. We extend the initial Take-Grant model to address the notion of vulnerabilities and introduce the vulnerabilities rewriting rules to specify how the protection state of the system can be changed by exploiting vulnerabilities. Our analysis is based on a bounded polynomial algorithm, which generates the closure of the Take-Grant graph regarding vulnerabilities. The closure helps to verify whether any subject can obtain an access right over an object. The application of our results have been examined in a case study which reveals how an attacker can gain an unauthorized access right by exploiting chain of vulnerabilities.

1 Introduction

The distribution and complexity of computer networks and the large number of services provided by them, makes computer networks vulnerable to cyber attacks. Currently several tools exist which analyze a host vulnerabilities in isolation, but to protect networks against attacks, we need to consider the overall network vulnerabilities and the dependency between services provided by the hosts.

Services may provide an acceptable level of security when considered in isolation, but a combination of these secure services may lead to subtle attack scenarios. For example, the file transfer protocol (ftp) and the hypertext transfer protocol (http) offered simultaneously in a same host, may allow the attacker to write in a web directory using the ftp service. This causes the web server to execute a program written by the attacker. Consequently, comprehensive analysis of network vulnerabilities needs considering individual hosts as well as their relationships.

The complexity of analyzing network vulnerabilities can be augmented as the number of hosts and services increases. Facing current enormous networks, automated approaches are necessary to analyze vulnerabilities.

* This research was in part supported by a grant from I.P.M (No. CS1383-4-04).

Some approaches have been proposed in the literature to analyze network vulnerabilities from the point of view of the relations between individual hosts and network configurations [1], [2], [3], [4], [5] . Such approaches mainly use model checking and graph-based techniques to generate and analyze an attack graph; the task has been done in exponential time. In [6], [7] polynomial time approaches have been suggested for the same problem without any specific upper bound on polynomial degree.

In this paper, we extend the Take-Grant protection model to address the concept of vulnerabilities, which allow an entity to change the protection state of the system and violate security policies. We propose a framework to model vulnerabilities based on their preconditions and postconditions, and an algorithm to analyze the model in bounded polynomial time with the size of protection system graph. The proposed algorithm can generate possible attack scenarios as well.

The remainder of this paper is organized as follows: Firstly, the previous works on Take-Grant protection model and network vulnerability analysis are reviewed. Then, our Vulnerability Take-Grant model is introduced as an extension to the Take-Grant model. The way to exploit some vulnerabilities can be represented in the extended model is shown in section 5. Our approach to vulnerability analysis comes in the next section. The application of Vulnerability Take-Grant model in a real network will be also examined in section 7. Finally, we conclude and propose future areas of research.

2 Related Work

The Take-Grant protection model was first developed by Jones et al. [8] in which the *safety problem*[1] could be solved in linear time. They provided the necessary and sufficient conditions under which rights and information could be transferred between two entities of the protection system and a linear time algorithm to test those conditions. Applications of the Take-Grant model to various systems have been explored separately [9], [10], [11], [12], and [13]. Extending the initial Take-Grant model also has been experienced by Frank and Bishop [14]. They proposed a method of extending the Take-Grant model to add notion of the cost of information or right flows and finding the most likely path in order of costs. Besides decidability, time complexity of the deciding algorithm has also been emphasized in nearly all previous works. These features have made the Take-Grant model more attractive than other formal access control models.

Based on the authors' knowledge, the Take-Grant protection model has not been used for host or network vulnerability analysis so far. Previous approaches for network vulnerability analysis mainly used model checking and graph-based techniques whose time complexity is either exponential or polynomial. Such approaches mainly depend on some off-the-shelf tools for scanning individual host vulnerabilities. Vulnerability scanner tools such as Nessus [15] scan hosts to discover vulnerabilities in the configuration. However, they do not investigate how a combination of configurations on the same host or among hosts on the same network can contribute to the vulnerabilities.

[1] The safety problem is defined in [22] as follows: Given an initial configuration of a protection system, whether a subject *s* can obtain some access right *r* over an object *o*?

The NetKuang system tries to assess beyond host vulnerabilities. It is an extension to a previous work on building a rule-based expert system, named Kuang [1] .Dacier [2] proposed the concept of privilege graphs. Privilege graphs are explored to construct an attack state graph, which represents different ways in which an intruder may reach a certain goal, such as root access on a host.

Ritchey and Ammann [3] used model checking for vulnerability analysis of networks via the model checker SMV. They could obtain only one attack corresponding to an unsafe state. The experiment was restricted to only specific vulnerabilities. However, the model checking approach has been used in some other researches to analyze network vulnerabilities [6], [16]. The model checking has the scalability problem which some researchers tried to overcome [6]. Ramakrishnan and Sekar [4] used a model checker to analyze a single host system with respect to combinations of unknown vulnerabilities. The key issue in their research was checking of infinite space model using model abstraction. Swiler et al. presented a method in [17] for generating attack graphs. Their tool constructs the attack graph by forward exploration.

In [5] CSP was used to model and analyze TCP protocol vulnerabilities. In this approach, the model checker FDR2 was used to verify some simple security properties and find attack scenarios. CSP has been used widely in modeling and analyzing security protocols [18] and verifying intrusion detection systems [19].Noel et al. presented TVA in [7] and [20] and investigated it more in [21]. In this approach, exploits are modeled as pre/post-conditions and a specific tool has been used to construct the attack graph. Encoding each exploit individually resulted in a large and complex model.

In our approach, similar vulnerabilities are represented in a single model. For example, all buffer overflow vulnerabilities are treated similarly. Moreover, this reduces the size of the model and cost of analysis. Moreover, our approach finds the attack paths using an algorithm in bounded polynomial time with the size of protection system graph.

3 Take-Grant Protection Model

The Take-Grant protection model is a formal access control model, which represents transformation of rights and information between entities inside a protection system. This model was presented first by Jones et al. [8] to solve the “Safety Problem”. They showed that using Take-Grant model, the safety problem is decidable and also can be solved in linear time according to the number of subjects and objects of the system.

In this model the protection state is represented as a directed finite graph. In the graph, vertices are entities of the system and edges are labeled. Each label indicates the rights that the source vertex of the corresponding edge has over the destination vertex. Entities could be subjects (represented by ●), objects (represented by ○) or play the both roles (represented by ⊗). The set of basic access rights is denoted as $R=\{t,g,r,w\}$ which t, g, r and w respectively stand for take, grant, read, and write access rights. To model the rights transfer, Take-Grant protection model uses a set of rules called de-jure rules. These rules transfer the Take-Grant graph to a new state which reflects the modification of protection state in an actual system. The de-jure rules are *take*, *grant*, *create* and *remove*. The take and grant rules are described briefly as:

1. Take rule: Let x, y, and z be three distinct vertices in a protection graph G_0 and let x be a subject. Let there is an edge from x to y labeled γ where $t \in \gamma$, an edge from y to z labeled β. Then the take rule defines a new graph G_1 by adding an edge to the protection graph from x to z labeled α, where $\alpha \subseteq \beta$. Fig 1.(a) shows the take rule graphically.
2. Grant rule: Let x, y, and z be three distinct vertices in a protection graph G_0 and let x be a subject. Let there is an edge from x to y labeled β where $g \in \gamma$, an edge from x to z labeled β. Then the grant rule defines a new graph G_1 by adding an edge to the protection graph from y to z labeled α, where $\alpha \subseteq \beta$. Fig.1(b) shows the grant rule graphically.

Having the take right over another subject or object means that its owner can achieve all rights of the associated subject or object unconditionally. However, obtaining the rights through the grant rule requires cooperation of the grantor.

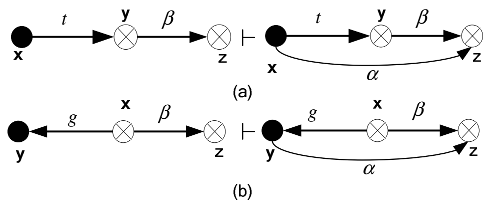

Fig. 1. (a) take rewriting rule. (b) grant rewriting rule.

4 The Vulnerability Take-Grant Model

The initial Take-Grant model is extended to address the notion of vulnerability. To use advantages of the Take-Grant model, it is critical to preserve the model abstraction. Without loss of generality, just for simplicity, here we only consider vulnerabilities which increase the attacker access rights.

The set of all possible vulnerabilities for a single host (which henceforth will be referred as *VLN*) can be found easily using vulnerability scanner tools such as Nessus. The *vulnerability* function associates a set of vulnerabilities to each vertex. More formally:

$$vulnerability: V \rightarrow 2^{VLN} \tag{1}$$

where *V* stands for the Vulnerability Take-Grant graph vertices and 2^{VLN} is the power set of *VLN*.

Henceforth, we refer to Vulnerability Take-Grant graph as VTG graph. Beside the initial Take-Grant rights, we need the following access rights:

1. **x**, which represents the execution right of a subject over an object.
2. **o**, which stands for ownership and represents the ownership of a subject over an object. This right specifies which subject currently owns an object.
3. **h**, which stands for hosting and represents a machine hosts an entity.

Thus, we extend the right set to be $R = \{t, g, r, w, x, o, h\}$.

We define the function *rights* to show the set of rights each entity has over another entity. More formally:

$$rights(u,v): V \times V \rightarrow 2^R \tag{2}$$

In this model, vulnerabilities of each entity are denoted by the label of related vertex. We present some examples of the model in the next section.

5 Modeling Vulnerabilities

The Vulnerability Take-Grant model is used to model vulnerabilities which their exploit can be demonstrated by a change in access rights. The change is represented by some rules we call them *vulnerability rewriting rules* (*VRR).* To demonstrate how vulnerabilities can be modeled using VTG, some groups of vulnerabilities are used as examples following by their graphical representation. In later sections of this paper, we focus more on the model.

5.1 Buffer Overflow Vulnerabilities

Buffer overflow vulnerabilities (BOF) are reported as the most exploited ones among network attack [23]. We model all vulnerabilities of this type as a rewriting rule. Assume a process *p* (having BOF) is running on the host *m* with the privilege of user account *a*; and the attacker *A* has the execution right over *p*. Now A can exploit BOF and execute his arbitrary code with the privilege of the user account *a*.

Fig. 2(a) depicts the *buffer overflow rewriting rule* and demonstrates how exploiting the *BOF* vulnerability results in a change in access rights. As shown, after exploiting *BOF*, the attacker achieves the new *take* access right (*t*) over user account *a*. We use the notation {*BOF*} as a vertex label to represent this vulnerability.

5.2 Weak Password Vulnerability

The weak password vulnerability (*WP*) arises when a user account with a weak password exists on a host *m* and the host provides a *login* service to other users (similar to what is common is web-based services). Assume the user *u* has an account *a* on host *m* and has chosen a weak password for it. Also assume this host provides a login service which provided by process *p*. Now the attacker *A* can guess the password of user *u* and take all the privileges of user account *a*.

Fig. 2(b) depicts the *password cracking rewriting rule* and demonstrates how exploiting the *WP* vulnerability results in a change in access rights. As shown, after exploiting *WP,* the attacker achieves the *take* access right (*t*) over user account *a*. We use the notation {*WP*} as a vertex label to represent this vulnerability. In addition, we use the vertex label {*Login*} to show the login service provided by process *p*. In fact, providing the login service is not a vulnerability, but the same notation is used for Fig. 2(b) depicts the *password cracking rewriting rule* and demonstrates how exploiting the *WP* vulnerability results in a change in access rights. As shown, after exploiting *WP,* the attacker achieves the *take* access right (*t*) over user account *a*. We use the notation {*WP*} as a vertex label to represent this vulnerability. In addition, we use the vertex label {*Login*} to show the login service provided by process *p*. In fact, providing the login service is not a vulnerability, but the same notation is used for vulnerabilities and services to preserve consistency and simplicity of the model.

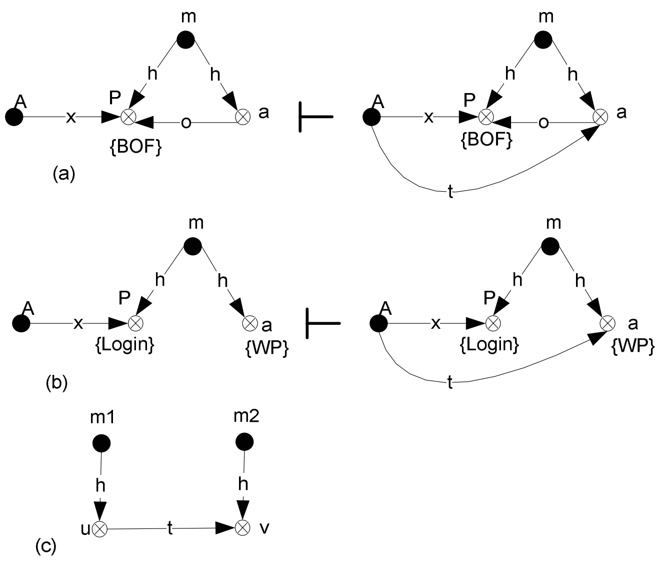

Fig. 2. Modeling vulnerabilities: (a) Buffer Overflow (b)Password Cracking (c) rhost vulnerability

5.3 Trust Vulnerabilities

Sometimes a user trusts another user and allows him/her to access resources. One of the best examples of such vulnerabilities is the *rhost* facility in UNIX. The *rhost* vulnerability occurs when a user trusts another user on a host or on the network. On operating systems such as UNIX and Windows NT based operating systems, users are allowed to define a list of their trustees in a file. In UNIX-based operating systems, typically this file is named *.rhosts* and is located in the user's home directory. These trustees take all the access rights of the user who trusts them.

The attacker does not need to run any program or malicious code to exploit this vulnerability. Fig. 2(c) demonstrates how this vulnerability can be modeled in *VTG.* Assume user account *v* is trusted by user account *u*. This trust is shown in *VTG* graph by a *take* edge from *u* to *v*. The vertex label {*rhost*} is used to represent this vulnerability. It should be mentioned that this vulnerability does not need any new rewriting rule, because no action is required to exploit it and we can add the related edges and vertex labels while we are building the *VTG* graph.

6 Analyzing the Model

In this section, we present a method for network vulnerability analysis using VTG model and investigate its efficiency for a set of vulnerabilities. Our analysis is based on the following question:

"Is it possible for attacker **A** to achieve access right **r** over **y** or not?"

or more formally, having the initial VTG G_0, is there a VTG graph G_k having an edge in G_k labeled *r*, and the sequence of transitions $\vdash^*$, such that $G_0 \vdash^* G_k$?

Rights in the *Take-Grant* protection model (and of course in VTG), can be transferred either conditionally or unconditionally. It is also the case in application of this model in vulnerability analysis. The attacker can exploit some vulnerabilities unconditionally while some others involve cooperation of other system subjects which grant some rights either unknowingly or intentionally. Our focus, here, is to consider unconditional capability of an attacker to acquire rights. To be precise, we are interested in the following question:

"Can attacker **A** achieve access right **r** over **y** unconditionally?"

Conditional transformation of rights has been investigated in the previous works on *Take-Grant* protection model. Authors in [8] and [24] dealt with this question provided that all the subjects in the system would cooperate. Snyder introduced the concept of "*stealing*" of rights and provided the necessary and sufficient conditions under which rights could be stolen if no owner of right *r* would grant it to other subjects or objects.

Grant rules are useless when our focus is on unconditional transformation of rights. What we mean by unconditional transformation of rights can be defined more formally in *VTG* by the predicate *can•access*:

Definition 1. The predicate *can•access*(α, **x, y,** VTG_0) is true for the right α, the vertex **x** (as subject), the vertex **y** (as subject or object), and the graph VTG_0; if there exist protection graphs $VTG_1, \ldots, VTG_n$ such that $VTG_0 \vdash^* VTG_n$ using only *take* and *vulnerability* rewriting rules, and there is an edge from **x** to **y** labeled α in VTG_n.

To answer the predicate *can•access*(α, **x, y,** VTG_0), it is needed to construct VTG's closure regarding to *de-jure* and *vulnerability rewriting rules*. First, we define the concept of *closure*:

Definition 2. Let A be the set of some rewriting rules. We define G^A the closure of G if all possible rules of A have been applied in G^A and no more rewriting rules can be applied in it.

The initial state of VTG graph is changed by both *de-jure* and *vulnerability* rewriting rules. Let's G^{dejure} be the closure of G regarding to *de-jure* rewriting rules and G^{VRR} be the closure of G regarding to *vulnerability rewriting rules*. It may be possible to apply one set of rewriting rules after constructing a closure using the other set of rewriting rules.

To capture all the possible attack paths, a *complete* closure is needed. We use the following psudo-code to construct a *complete* closure in which all the possible rewriting rules have been applied and no new rule can be applied anymore.

```
Gen_complete_Closure(G)
1-  Let list F initially contain all ordered pairs of the from (e, r) where e
    denotes edges labeled t, and r denotes the associated right.
2-  While (! IsEmpty(F))
    //applying all possible de-jure rules
3-    While (! IsEmpty(F))
4-        Let (e,r) = head(F)
5-        For each take rule applicable through e
6-            Add the resulting edge and its associated right to F, if it has
              not been inserted yet.
7-        Delete (e,r) from F
    //applying BOF rewriting rules
8-    for all $v \in V$
9-      if $BoF \in vulnerability(v)$ then
10-           Add an edge labeled t from all accounts having access to v to
              the owner of v.
11-           Add the above edge and its associated right to F, if it has not
              been added yet.
    //applying password cracking rewriting rules
12 -   for all $M \in Hosts$ //Hosts is the set of all machines in the system
13-           Add an edge labeled t from all accounts having login access
              to M to accounts having weak passwords in M.
14-           Add the above edge and its associated
              right to F, if it has not been added yet.
```

Theorem 1 deals with the correctness and time complexity of *Gen_complete-Closure* algorithm.

Theorem 1. *Gen_Complete_Closure* constructs the complete closure of G correctly in $O(V^4)$.

Proof: At first, we prove that lines 2-7 deal with constructing G_i^{dejure} given the input graph G_i at the beginning of the *i*th round of the algorithm. We should prove that the algorithm adds all the possible edges and rights and no multiple edges exist between vertices. Let $L=\{(R_1,r_1), (R_2,r_2),\ldots(R_n,r_n)\}$ be a sequence of applied rules leading to a correct G_i^{dejure} closure, where R and r stand for related *rules* and *rights* respectively. Assume there are some rights in *L* which are not produced by our algorithm and let (R_k, r_k), $1 \le k \le n$, to be the first such ordered pair appearing in *L*. We define the rights *t* in Fig. 1 the *basic right* of the *take* rule. The *basic right* of R_k should have been already added to graph by one of the rules R_1 to R_{k-1}. These rules have been applied by our algorithm similarly; so the *basic right* of R_k has been added to F and should be considered by the algorithm which leads in addition of r_k and contradicts the initial assumption that r_k has not been added by *Gen_complete_Closure* Algorithm. Moreover, the condition of line 6 in the algorithm makes sure that no ordered pair will be added to F repeatedly.

No we show that lines 9-14 of the algorithm constructs G_i^{VRR}(closure of G regarding to *buffer overflow* and *password cracking* vulnerability rewriting rules*)* correctly given the input graph G_i in the *i*th round of the algorithm. It is obvious that all the *buffer overflow* and *password cracking* rewriting rules are applied once by the algorithm. It's sufficient to prove that there is no need to consider any vulnerable vertex in *VTG* more than once. The applied rewriting rules add an edge labeled *t* to *VTG*. This operation doesn't make a previously considered vertex a candidate for applying a new *vulnerability* rewriting rule, because having an edge labeled *t* is not a part of precondition of any *vulnerability* rewriting rule.

No multiple rights (and their associated edges) will be added by algorithm, hence the list F will contain $O(V^2)$ ordered pairs at most. To apply the necessary *take* rules in line 5, it is sufficient to consider all the adjacent edges to the current edge *e,* and it will take *O(V)* at most. The cost of adding new edges and their associated rights would be of *O(1)* because it only requires checking the associated edges in the constructed graph. Every edge and its associated right will be added to and removed from list *F* at most once, thus time complexity of lines *2-7* is $O(V^3)$ in overall. The cost of applying *buffer overflow* and *password cracking* rewriting rules will be of *O(V)* and $O(V^2)$, respectively. We have just shown that the outer loop of the algorithm will be executed at most V^2 times. Thus the time complexity of lines *8-14* will be of $O(V^4)$. Consequently, the time complexity of the algorithm is $O(V^4)$.■

Having a *complete* closure, we can answer the *can•access* predicate which was defined at the beginning of this section in O(1). Therefore the following theorem holds:

Theorem 2. Let *A* be the union of the *take* and *vulnerability rewriting rules*. We can construct G^A in polynomial time and verify the *can•access* predicate in constant time.

It is worthy of note that the initial cost of constructing the *complete* closure will be paid once and the attacker's capability to access the network resources can be answered in constant time afterwards. Moreover, the algorithm can be modified to generate *attack path*. The *attack path* can be tracked by assigning text labels to rights when applying rewriting rules. The assigned text describes how the vulnerabilities are exploited or the *de-jure* rules are applied as well as the subjects and objects involved in the rules. Fig. 3 depicts how we can generate a new label from two previously generated ones. Assume that rights *p* and *q* have been already added by rewriting rules and text labels *Label(p)*and *Label(q)* contain the attack scenarios which lead to addition of these rights respectively. Moreover, assume we can now apply a new rewriting rule and obtain the new right *r*. The associated text label of *r*, *Label(r),* can be of the following form:

Label(*r*) = {*Label*(*p*), *Label*(*q*), "having access rights *p* and *q*, we can apply rewriting rule *x* and achieve right *r*" }

Subsequently, *Label(r)* contains the complete attack scenario acquiring right *r*.

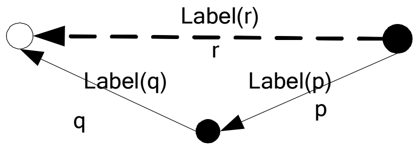

Fig. 3. Generating attack scenario labels

7 Case Study

In this section, we represent the application of Vulnerability Take-Grant model and the acquired results in vulnerability analysis of a typical network. Besides the previously introduced rewriting rules, we need some general rules to analyze the real world vulnerabilities. One of these general rules which addressed here arises from the fact that each user's access rights are subset of root's access rights. This fact can be shown in VTG model as a set of *take* edges drawn from *root* account to other user accounts defined on the same host.

Fig. 4 shows a local network. The attacker is outside the network. The firewall configuration allows remote users to just have access to web and mail services. The attacker goal is to gain access to *Ali*'s files hosted on Saman. On the machine *NSC*, *HTTP* and *SMTP* services are listening to the associated ports. These services are running with the user privileges *apache* and *root* respectively. Also SSH and SMB services are running on the machine FS with user privilege *root* and RPC service is running on Saman with the same user privilege.

Using the Nessus scanner, we found that the services *HTTP* on *NSC*, *SMB* on *Saman* and *RPC* on *FS* have buffer overflow vulnerability. Moreover, we found that the user account *root* on the machine *FS* suffers from weak password vulnerability and the user *Ali* has added the account manager from machine *FS* to its *.rhost* file.

This network's VTG model is represented in Fig. 5. To avoid congestion, unnecessary relations between hosts are ignored in the figure, and the new rights added as the impacts of the vulnerabilities are showed by dotted edges, and the attacker final path is showed by dashed edges.

By using *Gen_Complete_Closure* alghorithm described in the previous section and applying the rewriting rules on the above VTG graph, G^A is generated.

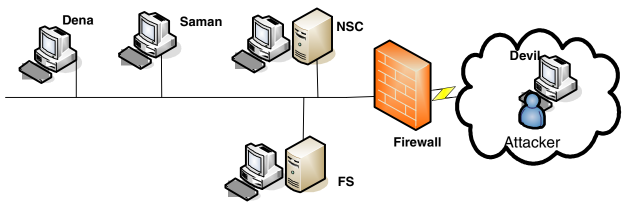

Fig. 4. The example network topology

As mentioned above, the Attacker's goal is to access *Ali*'s file on the Saman. Attacker is allowed to access *Ali*'s file if and only if there is an edge from Attacker to *Ali* in G^A including right r in its set of access rights. The attack path which brings the Attacker to the *Ali*'s file is shown in dashed line in Fig. 5. We can obtain the attack path by using the previously described technique. One possible attack scenario is as follows:

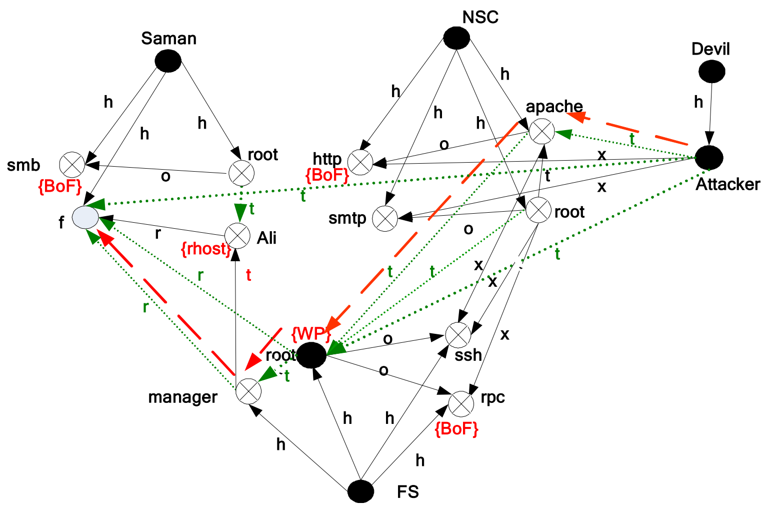

Fig. 5. Part of G^A, generated for the case study network using *Gen_Complete_Closure*

1. The *Attacker* exploits the *HTTP* buffer overflow vulnerability on the machine *NSC* and gains the user privilege apache on this machine.

2. Now the *Attacker* has access to *SSH* service on machine FS and can try to guess *root* password.

3. After finding the *root* password, the *Attacker* has all the rights of user account *manager* on machine *FS*.

4. Pretending to be *manager*, the *Attacker* acquires *Ali*'s access rights on machine *Saman*.

5. Consequently, the *Attacker* reaches its final goal, which is having access to file *f* on machine *Saman*.

8 Conclusions and Future Works

In this paper, we introduced a new method for network venerability analysis which is based on the Take-Grant protection model. This method affords the possibility of representing the protection state of a network with a formal model. The attacker's capability to access the resources of network can be analyzed by the model. We also introduced the *complete* closure concept to address all the possible ways of exploiting vulnerabilities and presented an algorithm to construct the *complete* closure graph in $O(V^4)$. With *complete* closure, the safety problem could be answered in constant time. Besides analyzing vulnerabilities, the proposed method could generate possible attack scenarios.

It is possible to use the model for more comprehensive analysis. Answering to questions such as finding the *critical vulnerable path*, finding the *shortest path of accessing a right* and finding *minimum cost path of accessing rights* (considering the possibilities or difficulties of exploiting different vulnerabilities) can represent further applications of Take-Grant model in vulnerability analysis. Reducing the time complexity of the analysis can be considered as well. The proposed algorithm constructs the *complete* closure in bounded polynomial time and answers to *safety problem* in constant time. Considering the similarity of *de-jure* and *vulnerability* rewriting rules, it may be possible to analyze the vulnerabilities by an algorithm just like *can●steal* in linear time. The nature of Take-Grant model makes it most suitable for analyzing the vulnerabilities based on changes in access rights. Extending this model to cover a broader set of vulnerabilities will be of particular interest. This suggests several avenues of research. First, it can be studied how to model the vulnerabilities which decrease the access rights. Secondly, it is interesting to generalize this method for analyzing vulnerabilities based on a suitable taxonomy of vulnerabilities and their preconditions and postconditions.

References

1. D. Zerkle, and K. Levitt: NetKuang – A Muti-Host Configuration Vulnerability Checker. Proceedings of the sixth USENIX UNIX Security Symposium, San Jose, CA, 1996.
2. M. Dacier, Y. Deswarte: Privilege Graph: An Extension to the Typed Access Matrix Model. Proceedings of Third European Symposium on Research in Computer Security (ESORICS 94), (Brighton, UK), Lecture Notes in Computer Science: Computer Security, 875, pp.319-334, Springer-Verlag, 1994.
3. R.W. Ritchey, P. Ammann: Using Model Checking to Analyze Network Vulnerabilities. Proceedings of IEEE Symposium on Security and Privacy, pages 156–165, May 2001.
4. C.R. Ramakrishnan, R. Sekar: Model-Based Analysis of Configuration Vulnerabilities. Journal of Computer Security, vol. 10, no. 1/2, pp. 189-209, 2002.
5. H. R. Shahriari, R. Jalili: Using CSP to Model and Analyze Transmission Control Vulnerabilities within the Broadcast Network. Proceedings of the IEEE International Networking and Communication Conference (INCC'2004), June 2004, pp. 42-47.
6. P. Ammann, D. Wijesekera, S. Kaushik: Scalable Graph-Based Network Vulnerability Analysis. Proceedings of 9th ACM Conference on Computer and Communications Security, Washington, DC, November 2002.
7. S. Noel, B. O'Berry, C. Hutchinson, S. Jajodia, L. Keuthan, A. Nguyen: Combinatorial Analysis of Network Security. Proceedings of the 16th Annual International Symposium on Aerospace/Defence Sensing, Simu-lation, and Controls, Orlando, Florida, April 2002.
8. J. R. Lipton, L. Snyder: A Linear Time Algorithm for Deciding Security. Proc 17th Annual Symp. on the Foundations of Computer Science (Oct. 1976), 33-41.
9. M. Bishop: Hierarchical Take-Grant Protection Systems. Proc. 8th Symp. on Operating Systems Principals (Dec. 1981), 107-123.
10. A. Jones: Protection Mechanism Models: Their Usefulness. in Foundations of Secure Computing, Academic Press, New York City, NY (1978), 237-254
11. L. Snyder: On the Synthesis and Analysis of Protection Systems. Proc. Sixth Symp. on Operating Systems Principals (Nov. 1977), 141-150.
12. M. Wu: Hierarchical Protection Systems. Proc. 1981 Symp. On Security and Privacy (Apr. 1981), 113-123.
13. M. Bishop: Conspiracy and Information Flow in the Take-Grant Protection Model. Journal of Computer Security, vol 4(4), 1996, pp 331-360.
14. J. Frank, M. Bishop: Extending The Take-Grant Protection System. Technical Report, Department of Computer Science, University of California at Davis, 1996.

15. R. Derasion, [online]: The Nessus Attack Scripting Language Reference Guide. 2000. Available from: http://www.nessus.org.
16. O. Sheyner, J. Haines, S. Jha, R. Lippmann, J.Wing: Automated Generation and Analysis of Attack Graphs. Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, 2002.
17. L. Swiler, C. Phillips, D. Ellis, S. Chakerian: Computer Attack Graph Generation Tool. Proceedings of DARPA Information Survivability Conference & Exposition II, June 2001.
18. P. Ryan, S. Schneider: Modeling and Analysis of Security Protocols - A CSP Approach. Addison-Wesley, 2001.
19. G. Rohrmair, G. Lowe: Using Data-Independence in the Analysis of Intrusion Detection Systems. Workshop on Issues in the Theory of Security (WITS'03), Warsaw, Poland, April 2003.
20. S. Jajodia, S. Noel, B. O'Berry: Topological Analysis of Network Attack Vulnerability. Managing Cyber Threats: Issues, Approaches and Challenges. V. Kumar, J. Srivastava, A. Lazarevic (eds.), Kluwer Academic Publisher, 2003.
21. S. Noel, S. Jajodia: Managing Attack Graph Complexity through Visual Hierarchical Aggregation. Proceedings of the ACM CCS Workshop on Visualization and Data Mining for Computer Security, Fairfax, Virginia, October 2004.
22. J. S. Shapiro: The Practical Application of a Decidable Access Control Model. Technical Report SRL-2003-04, John Hopkins University, 2003.
23. SANS Research Center, [online]: The SANS Top 20 Internet Security Vulnerabilities. Available from: http://www.sans.org/top20/.
24. J. R. Lipton, and L. Snyder: A Linear Time Algorithm for Deciding Subject Security. J. ACM. 24, 3 (Jul. 1977), 455-464.