PKG-VUL: Security Vulnerability Evaluation and Patch Framework for Package-Based Systems

Jong-Hyouk Lee, Seon-Gyoung Sohn, Beom-Hwan Chang, and Tai-Myoung Chung

In information security and network management, attacks based on vulnerabilities have grown in importance. Malicious attackers break into hosts using a variety of techniques. The most common method is to exploit known vulnerabilities. Although patches have long been available for vulnerabilities, system administrators have generally been reluctant to patch their hosts immediately because they perceive the patches to be annoying and complex. To solve these problems, we propose a security vulnerability evaluation and patch framework called PKG-VUL, which evaluates the software installed on hosts to decide whether the hosts are vulnerable and then applies patches to vulnerable hosts. All these operations are accomplished by the widely used simple network management protocol (SNMP). Therefore, system administrators can easily manage their vulnerable hosts through PKG-VUL included in the SNMP-based network management systems as a module. The evaluation results demonstrate the applicability of PKG-VUL and its performance in terms of devised criteria.

Keywords: Security vulnerability evaluation, patch, PKG-VLU, PKG-MIB, Ubuntu, SNMP.

Manuscript received Oct. 3, 2008; revised Aug. 7, 2009; accepted Aug. 18, 2009.

This work was supported by the IT R&D program of MKE/IITA, Rep. of Korea [2007-S022-02, The Development of Smart Monitoring and Tracing System against Cyber-attack in All-IP Network].

Jong-Hyouk Lee (jhlee@imtl.skku.ac.kr) is with Internet Management Technology Lab., Sungkyunkwan University, Suwon, Rep. of Korea, and is now with IMARA Team, INRIA, France.

Seon-Gyoung Sohn (sgsohn@etri.re.kr) and Beom-Hwan Chang (bchang@etri.re.kr) are with S/W & Content Research Laboratory, ETRI, Daejeon, Rep. of Korea.

Tai-Myoung Chung (tmchung@imtl.skku.ac.kr) is with Internet Management Technology Lab., Sungkyunkwan University, Suwon, Rep. of Korea.

doi:10.4218/etrij.09.0108.0578

I. Introduction

Attacks based on vulnerabilities are an urgent security problems faced by system administrators. In particular, remote attacks which exploit one or more vulnerabilities to seize control or break down vulnerable hosts over the Internet are dramatically increasing. Such remote attacks do not stop at one host. They use the host as a zombie host to find and attack other vulnerable hosts. Accordingly, such attacks based on vulnerabilities have a serious impact over time.

The best defense against such attacks is for system administrators to keep the latest software on their hosts and apply patches to their software as soon as possible to repair vulnerabilities. However, system administrators often do not fix vulnerabilities even though the patches are published [1], and some of system administrators do not know how to apply the patches for their vulnerable hosts. Moreover, system administrators who have many hosts to manage cannot prevent zero-day attacks [2] because they have to apply the patches to all hosts in a short time. Therefore, an efficient and convenient method is needed to evaluate and patch known vulnerabilities.

In this paper, we introduce a simple network management protocol (SNMP)-based security vulnerability evaluation and patch framework, called PKG-VUL. PKG-VUL enables evaluation of the software installed on hosts to detect known vulnerabilities and grade the vulnerability risk weight of hosts based on the Common Vulnerabilities and Exposures (CVE) [3] and information of software maintained by PKG-MIB [4]. As the result of evaluation, we determine which software is vulnerable and which host with vulnerable software in the network has the highest degree of vulnerability. After the evaluation, the vulnerable software is immediately patched with the latest corresponding software or patch code.

The main contributions of this paper are the following.

- The proposed PKG-VUL enables system administrators to identify their vulnerable software installed on hosts and its vulnerability level by a vulnerability evaluation based on the combined relations of the vulnerable software and the vulnerability information.
- The proposed PKG-VUL also enables system administrators to patch vulnerable software immediately based on the vulnerability evaluation. Thus, system administrators can keep the latest or invulnerable software on their hosts.
- The operations for the evaluation and patching of PKG-VUL are accomplished by SNMP. Therefore, PKG-VUL would be easily included in SNMP-based network management systems as a module.

The remainder of this paper is organized as follows. In section II, we present the current problems. In section III, we introduce our previous works to manage the information of software. In section IV, we present the architecture of PKG-VUL and describe how it is organized and how it works. In section V, we evaluate the applicability of PKG-VUL and provide the comparison results in which previously developed defense methods are compared with PKG-VUL. Finally, we conclude this paper in section VI.

II. Problem Statements

1. Recent Defense Methods

Recent attacks based on known vulnerabilities such as CodeRed [5], Nimda [6], and SQL Slammer [7] have motivated the development of more efficient and effective defense methods. As part of an effort to develop defense methods, Shield [8], TaintCheck [9], STEM [10], Snort [11], and Nessus [12] have been developed. Shield, developed by Microsoft Research, is a network-based vulnerability checking filter. The filter, operating on the TCP/IP stack, examines the incoming or outgoing traffic of vulnerable applications and corrects traffic that exploits vulnerabilities. A weak point of Shield is the generation of signatures and application of the signatures because Shield requires manually generated signatures derived from known vulnerabilities, and the signatures need to be applied to the TCP/IP stack before operation. Manual signature generation is clearly too slow to prevent attacks which infect hundreds of thousands of systems in a matter of hours or minutes. TaintCheck is more flexible and responsible in comparison with Shield. TaintCheck generates vulnerability-specific signatures from network traffic. The generated signatures are used to label the network traffic as tainted and to keep track of the propagation of tainted data as the program executes. TaintCheck alerts occur when tainted data is used to attack. STEM is a code instrumentation method. STEM attempts to monitor the execution of potentially vulnerable software via code instrumentation and then replaces the vulnerable software with an automatically patched version. Snort is widely used to monitor network traffic and detect attacks of known vulnerabilities as a network intrusion detection system (NIDS). However, it focuses on detection rather than attack prevention even though Snort is usually more customized by vulnerable software than firewalls. Nessus is a vulnerability scanner that checks network vulnerabilities of hosts.

Such defense methods have certain weaknesses. They introduce substantial overhead in the TCP/IP stack or in software execution time because they require the generation of vulnerability-specific signatures or monitoring of the TCP/IP stack. In particular, Shield, TaintCheck, and STEM need to generate signatures, although the patches for vulnerabilities have been published in public domains, such as CVE, CERT [13], and ISS [14]. Snort also has one of the drawbacks of NIDS, namely, a high false positive rate, which complicates the reaction process. Nessus only provides vulnerability information for hosts; therefore, system administrators need to perform tasks to remove or update vulnerable entities separately. Above all, system administrators perceive the defense methods as inconvenient and imprecise, so it is desirable to seek a more efficient and convenient alternative method to achieve the same goals.

2. Impact of Known Vulnerabilities

We are here concerned with the impact of known vulnerabilities. According to [15], more than 90% of attacks today exploit known vulnerabilities. The question we have to ask is why attacks exploiting known vulnerabilities make up a great proportion of attacks and most of such attacks are successful.

According to [16], vulnerabilities can be classified into three types: secret, published, and patched. Published and patched vulnerabilities are called known vulnerabilities. A secret vulnerability has its own time s_1 until it is either published or patched. Note that a secret vulnerability may change its status when it is published. However, it may still be a secret and may be exploited because it has not been disclosed on public domains, such as CVE, CERT, and ISS; therefore, many people would be unaware of it. A published vulnerability, which has its own time s_2, is one that has been published, but its corresponding patch is not yet available. A patched vulnerability having its own time s_3 is one that has been published, and its corresponding patch is also available. Suppose Φ is a software puerility, where time(0) $\leq \Phi < s_1$.

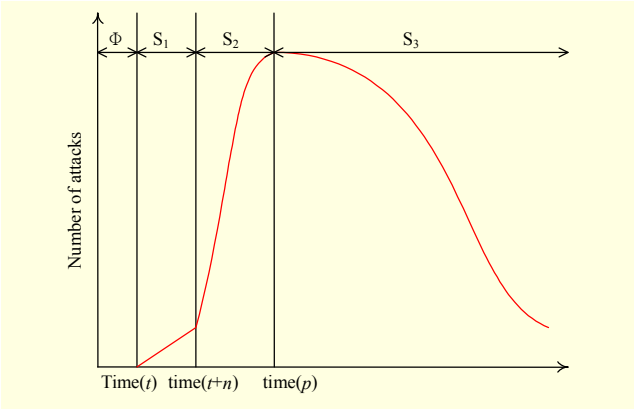

Fig. 1. Relation between vulnerability's lifetime and attack.

Table 1. Character of classified vulnerabilities.

	Secret	Published	Patched
Existing time	s_1	s_2	s_3
Transition time	time(t)	time(t+n)	time(p)
Impact	Low	Medium	High

Then, the lifetime of vulnerability is defined as $\Phi + s_1 + s_2 + s_3$. The relation between the lifetime of vulnerability and attack is depicted in Fig. 1.

In Fig. 1, time(t) is the first time of vulnerability discovery. The vulnerability is published when someone reveals details of the problem to the public domain at time($t + n$), where n is a secret time, and $n = s_1$. It is clear that, in some cases, the discoverer of a vulnerability does not disclose the problem immediately. Thus, the duration of n is decided by the discoverer. If n has a long duration, the vulnerability remains a secret for a long time. In this case, the secret vulnerability causes an unknown vulnerability attack. During the time of published vulnerability s_2, the information of vulnerability is announced: what it is, how it can be exploited, and how to patch the software having the vulnerability if a corresponding patch is available. There is a contentious ongoing debate about how software vulnerability should be made public. In s_2, the information of a published vulnerability can enable system administrators to take precautions that prevent or reduce attacks which exploit the published vulnerability. On the contrary, it can provide attackers with information on valuable software as well. The time of patched vulnerability s_3 has the longest duration in the lifetime of vulnerability since this period ends when the corresponding vulnerable software is removed or patched on the systems. In s_3, attacks slowly decrease due to the availability of the patch. However, a large proportion of system administrators are not particularly cautious and do not take adequate precautions, such as filtering ports, applying patches, and removing vulnerable software, even a few weeks after the patch is available [1], [15], [16]. Table 1 shows the summarized character of classified vulnerabilities.

In this paper, we focus on the time of patched vulnerability s_3 to reduce its impact. Note that we do not intend to reduce the time of secret vulnerability s_1; however, the proposed PKG-VUL enables system administrators to take adequate precautions for the time of published vulnerability s_2. For instance, if system administrators apply the patch to their vulnerable software as soon as possible after the time the patch is released time(p), the success rate of the attack is decreased, and thereby the number of attacks is also decreased.

III. PKG-MIB

In this section, we introduce PKG-MIB[1] as in our previous work [4]. PKG-MIB was originally developed as a private-MIB to manage the information of software installed on Linux systems by SNMP. Note that PKG-MIB is redefined for use with Microsoft Windows if it provides the information of software installed on Microsoft Windows.

To fill the information related software into PKG-MIB, we define objects that describe the behavior of the information-

Table 2. Information of *softwareStats* group.

Object	Entry	Sub-object	Syntax
statsTable(1)	-	-	Sequence of
	statsEntry(1)	-	Sequence
		indexStats(1)	CounterIndex
		name(2)	DisplayString
		version(3)	DisplayString
		desc(4)	DisplayString
		status(5)	DisplayString
infoTable(2)	-	-	Sequence of
	infoEntry(2)	-	Sequence
		indexInfo(1)	CounterIndex
		priority(2)	DisplayString
		section(3)	DisplayString
		maintainer(4)	DisplayString
		source(5)	DisplayString
		depends(6)	DisplayString
		size(7)	DisplayString

1) Internet Assigned Numbers Authority (IANA) has assigned the following private enterprise number to us: 27315.

related software. PKG-MIB only has a *softwareStats* group, which has its own object identifier (OID): 1.3.6.1.4.1.27315.1.0. Table 2 shows the *softwareStats* group. The objects of the *softwareStats* group relate to software and are used by an SNMPv2 entity acting in an SNMP agent role to describe those object resources. The SNMP agent controls the objects of the *softwareStats* group for dynamic configuration by an SNMP manager.

Table 2 lists the objects contained in the *softwareStats* group in two sub-tables. The first one is *statsTable* (1.3.6.1.4.1.27315.1.0.1), which contains essential objects to represent the information of software. Other one is *infoTable* (1.3.6.1.4.1.27315.1.0.2), which contains additional objects. These tables are read-only tables consisting of one entry for each object resource that can be dynamically configured when the software information is changed by installation, updating, removal, or patching. For instance, if any software is updated by a software management tool such as Advanced Packaging Tool (APT) [4], [17], [18], related objects are updated. In another example, if a system administrator is interested in the status information of installed software, then the OID is *softwareStats*.1.1.5 or 1.3.6.1.4.1.27315.1.0.1.1.5 is used to obtain the status information by the SNMP queries.

IV. PKG-VUL Architecture

We introduce the goals and architecture of PKG-VUL in this section. First, we present the objective and overview of PKG-VUL, and then the two main modules are described in detail.

1. Goals and Overview

There are three main objectives of PKG-VUL: agility, usability, and scalability.

Agility: The proposed architecture must be able to patch as soon as possible. Agility must be designed into PKG-VUL so that a corresponding patch for a vulnerability is applied to vulnerable software as soon as the corresponding patch is released to the public domains.

Usability: The proposed architecture must be easy to use. PKG-VUL must be designed to provide ease of use for system administrators. Since the lifetime of a vulnerability is the longest in s_3, usability must be provided to system administrators along with agility.

Scalability: The proposed architecture must be scalable for large scale networks. We must design PKG-VUL in such a way that PKG-VUL becomes a scalable tool, which has low CPU usage, low memory usage, and low traffic for deployment in large scale networks.

PKG-VUL achieves agility by applying the well-known really simple syndication (RSS) content delivery protocol. The published and patched information for vulnerabilities is posted on public domains and then the collector of PKG-VUL, which has the function of an RSS reader, aggregates the posted information. The aggregated information is used to evaluate with PKG-MIB. As previously mentioned, a large proportion of system administrators do not fix their systems even though the systems are vulnerable and the patches are also published on the public domains. The convenience of PKG-MIB brings about usability and scalability. For instance, system administrators can judge whether the managed software is vulnerable by a few clicks because PKG-MIB is a private-MIB. The information is delivered by an SNMP manager [4]. In addition, PKG-MIB could be easily included in SNMP-based network management systems as a module. Accordingly, it achieves scalability.

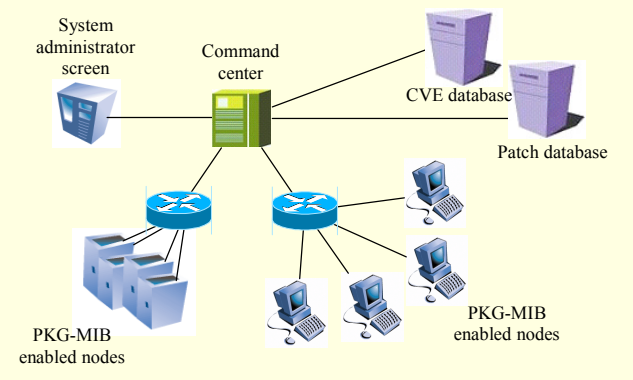

Fig. 2. Network topology managed by PKG-VUL.

Figure 2 shows the network topology managed by PKG-VUL. In the network topology, the command center has the function of an SNMP manager, and the managed hosts have the function of SNMP agents, including PKG-MIB. Thus, the command center gathers the software information from the managed hosts using the SNMP protocol. The command center also has the function of an RSS reader to obtain the CVE and patch information from the public domains. In the VUL-check module of the command center, such information is used to evaluate which node has vulnerable software and which node has the highest degree of vulnerability in the managed network. After the vulnerability evaluation, the vulnerable software is fixed by the VUL-patch module with latest patch if the corresponding patch is available. Figure 3 shows the procedures of vulnerability evaluation and patching.

A. Vulnerability Evaluation

The CVE and patch information is delivered to the collector of the command center by RSS. The collector records the information in its data structures. The two data structures are

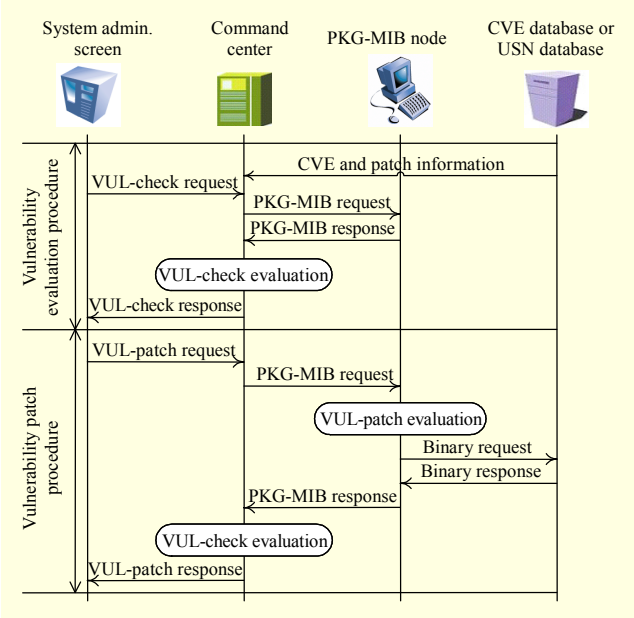

Fig. 3. Procedures of vulnerability evaluation and patching.

Table 3. Two data structures of collector.

CVE structure (published vulnerability)	
Field	Description
cve_name	Name of CVE
cve_status	Status of CVE (Entry or Candidate)
cve_desc	Description of CVE
USN structure (patched vulnerability)	
Field	Description
usn_num	Number of USN
usn_name	Name of USN
vul_name	Related names of vulnerabilities
cve	Related names of CVEs
date	Published date of USN
software	Software can be used for patch

shown in Table 3. The recorded information is then used for vulnerability evaluation. Note that the recorded information is about the published vulnerability delivered from the website of CVE and the patched vulnerability delivered from the website of Ubuntu security notices (USN)[2] [19]. The VUL-check request is executed by a system administrator, and then the function of the SNMP manager in the command center requests the information of software installed on the host. At this time, the VUL-check module evaluates the software based on the delivered information presented in Table 2 and the recorded information presented in Table 3. The evaluation procedure in the VUL-check module is introduced in detail in section IV.2. Finally, the VUL-check response, including the result of the vulnerability evaluation, is sent to the system administrator from the command center.

B. Vulnerability Patch

The system administrator decides that the vulnerable software needs to be patched based on the vulnerability evaluation. The VUL-patch request including the information of the vulnerable software is executed by the system administrator, and then a PKG-MIB request is sent from the command center to the host. At this time, the VUL-patch module in the host executes the software management tool, APT. Note that APT is a software management tools that enables easy installation, updating, patching, and removal of software via a remote patch (software) database [17], [18]. The software management tool installs the corresponding patch, and then the PKG-MIB response, including the result of the patch, is sent to the commend center. The patch procedure in the VUL-patch module is introduced in detail in section IV.2. At this time, the VUL-check module again evaluates the vulnerability of the host. Finally, the VUL-patch response is sent to the system administrator from the command center.

Thus, the system administrator can apply the patches corresponding to the vulnerabilities to prevent attacks that exploit the patched vulnerabilities which have their own time, s_3, as soon as the patches are available. Moreover, the proposed PKG-VUL enables system administrators to take adequate precautions, such as filtering addresses and ports for a published vulnerability whose time is s_2.

2. Modules

A. VUL-Check Module

The VUL-check module has two main functions. The first one is the detection of known vulnerabilities based on the information of PKG-MIB, which is the information of software installed on the host, and USN, which is the information of patched vulnerabilities for Ubuntu Linux. The second function is the evaluation of detected vulnerabilities.

The vulnerability detection function is shown in algorithm 1. In algorithm 1, *H.release* obtained from *system.sysDescr*.0 is the version of Ubuntu Linux installed on a host. The information of the *softwareStats* group provided by PKG-MIB is *P*. Set $P = \{P[1], P[2], \cdots, P[\alpha]\}$, where α is the total number of programs installed on the host. Define *P.index* as the index of software. Let *P.name* denote the name of software installed on the host, and let *P.version* denote the version of the software.

2) We have implemented PKG-VUL on Ubuntu Linux, which has a security repository to provide patches for patched vulnerabilities.

Algorithm 1. Vulnerability detection

```
1:    begin
2:    input H, P, U
3:    index ← 1
4:    for i ← 1 to α do
5:       for j ← 1 to M do
6:          if P[i].name = U[j].name then
7:             if H.release = U[j].release then
8:                if P[i].version < U[j].version then
9:                   V[index] ← (P[i].index, U[j].index)
10:                  index ← index + 1
11:               end if
12:            end if
13:         end if
14:      end for
15:   end for
16:   output V
17:   end
```

For instance, $P[i].name$ is obtained from *softwareStats*.1.1.2.*i*. Also, $P[i].version$ is the version of $P[i].name$.

The USN structure (U) is obtained by the collector which records U from the USN website [19]. Then it is defined as $U = \{U[1], U[2],\cdots, U[M]\}$, where M is the total number of entries in U, and each U entry consists of the index of entry, $U.index$; the name of the vulnerable software, $U.name$; the version of the patched software, $U.version$, which is updated for the vulnerability patch of $U.name$; and the version of Ubuntu Linux, $U.release$, which indicates the version of Ubuntu Linux having $U[any] \in U$. In this algorithm, H, P, and U are used as inputs. The output is the list of vulnerable software information V. Then, V is defined as $V = \{V[1], V[2],\cdots, V[L]\}$, where L is the last index of the list. Note that V consists of two types of binding information–the index of vulnerable software installed on the host and the index of the software to be used to patch the vulnerable software.

In algorithm 1, each program is examined by its name, which is one attribute of P. Then, U is retrieved by the name of the software which is represented by $P[i].name$. If there is an entry which has same name as $P[i].name$, the $U[j].release$ indicated in U is compared with the version of Ubuntu installed on the host $H.release$. When the result of this comparison is positive, the current version of the software installed on the host $P[i].version$ is compared with a $U[j].version$ which is known to be invulnerable to the related vulnerability. The software installed on the host is identified as vulnerable if the version is earlier than the version of the patched software. This process is iterated α times, which means that all of the software installed on the node is examined. At the end of algorithm 1, V is returned as the output. Next, we acquire the list of vulnerable software installed on the host.

Table 4. Weights of various types of software.

Section	Weight	Section	Weight
restricted/based	10	python	6
restricted/misc	10	perl	6
web	9	comm.	6
net	9	universe/misc	5
admin	8	translations	5
utils	8	x11	5
base	8	gnome	5
libs	8	graphics	5
misc	8	oldlibs	5
mail	7	math	4
shells	7	contrib/x11	4
otherosfs	7	editors	3
devel	7	doc	3
libdevel	7	sounds	2
interpreters	7	games	2
text	6		
Priority	**Weight**	**Priority**	**Weight**
required	10	optional	3
important	8	extra	2
standard	6		

・Section values are obtained from softwareStats.2.1.3
・Priority values are obtained from softwareStats.2.1.2

To evaluate the vulnerabilities, we define the weights of various types of software as shown in Table 4. The defined weights for the section and the priority value of the software types are also shown. Those values are obtained by PKG-MIB. In Table 4, the section of software indicates which software is included in which section. For instance, the Linux kernel module is included in the restricted/base section, and iptables[3] is included the net section, so the high degree section is more susceptible to attacks than the low degree section. The weight for priority is similar to the weight for section. The priority of software is assigned by the software maintainers. For instance, "required software" is necessary for the proper functioning of a system, whereas "optional software" includes all those types of software that a user might reasonably want to use, such as X window system, Latex system, and other applications.

The status of each CVE, which is determined by the CVE Editorial Board [3], is a "Candidate" or "Entry" as indicated in Table 3. The CVEs, whose status is "Candidate," are not

3) iptables is generally used for stateful and stateless packet filtering and other IP packet manipulation.

officially recognized as vulnerabilities. To be official vulnerabilities, "Candidate" CVEs have to poll a majority of positive votes by the Editorial Board members. If the members determine the CVE to be official, the status of the CVE will become "Entry." A "Candidate" CVE should be deleted from the CVE list if the "Candidate" CVE is a duplicate of another CVE, if further analysis shows that the vulnerability does not exist, or if the CVE needs to be recast. This means that a CVE whose status is "Candidate" can be rejected or deleted. Some CVEs may not be vulnerabilities. For such reasons, the weight of a "Candidate" CVE is 0.7, whereas the weight of an "Entry" CVE is 1. Equation (1) shows the formula for calculating the weight of a CVE used in the evaluation function:

$$weight\_cve = \sum_{i=1}^{\gamma} w_i, \tag{1}$$

where w_i is the weight for each CVE, and γ is the total number of CVEs which the vulnerable software contains. Let $Risk_{\text{soft}}$ be the risk weight of vulnerable software. Equation (2) shows the formula for calculating $Risk_{\text{soft}}$.

$$Risk_{\text{soft}} = \lfloor (weight\_cve \times 0.1 + 1.0) \times weight\_section \times weight\_priority) \rfloor, \tag{2}$$

where *weight_section* and *weight_priority* are assigned according to the section and priority of vulnerable software (see Table 4). For instance, software is identified as vulnerable by algorithm 1. This software's section is "utils" and the priority is "standard." In addition, the status of two related CVEs with vulnerable software is "Candidate." Then, the risk weight of vulnerable software is calculated as

$$54 = \lfloor ((0.7 + 0.7) \times 0.1 + 1.0) \times 8 \times 6 \rfloor. \tag{3}$$

The main purpose of the evaluation function is to assign the risk weight for vulnerable nodes. Suppose $Risk_{\text{node}}$ is the risk weight of a vulnerable node involving vulnerable software. Then, it is calculated as

$$Risk_{\text{node}} = \sum_{j=1}^{\beta} \left\lfloor \left(\sum_{i=1}^{\gamma} w_i \times 0.1 + 1.0 \right) \times weight\_section \times weight\_priority \right\rfloor, \tag{4}$$

where β is the total number of vulnerable programs which the vulnerable node contains. In addition, the risk weight of a managed network including vulnerable nodes $Risk_{network}$ is calculated as

$$Risk_{\text{network}} = \sum_{k=1}^{h} \sum_{j=1}^{\beta} \left\lfloor \left(\sum_{i=1}^{\gamma} w_i \times 0.1 + 1.0 \right) \times weight\_section \times weight\_priority \right\rfloor, \tag{5}$$

where h is the total number of nodes in the managed network.

Algorithm 2. Vulnerability evaluation

```
1:  begin
2:  input V
3:  node_risk ← 0
4:  for i ← 1 to β do
5:    is ← V[i].pindex
6:    weight_section ← weight of P[i].section
7:    weight_priority ← weight of P[i].priority
8:    totalCve ← 0
9:    iu ← 0
10:   for all elements of V[i].uindex do
11:     iu ← iu + 1
12:     usn_num ← V[i].uindex[iu]
13:     ic ← 0
14:     numCve ← 0
15:     for all elements of U[usn_num].C_name do
16:       ic ← ic + 1
17:       cve_name ← U[usn_num].C_name[ic]
18:       if C[cve_name].status = "Entry" then
19:         numCve ← numCve + 1
20:       else
21:         numCve ← numCve + 0.7
22:       end if
23:     end for
24:     totalCve ← totalCve + numCve
25:   end for
26:   software_risk ← (totalCve × 0.1 + 1.0)
      × weight_section × weight_priority
27:   node_risk ← node_risk + software_risk
28: end for
29: η ← node_risk
30: output η
31: end
```

An algorithm for the vulnerability evaluation function is shown in algorithm 2. In algorithm 2, the input is V obtained from algorithm 1, and the output is the weight of a vulnerable host η. Each entry of V consists of the index of vulnerable software *pindex* and the index of related USN *uindex*, where *uindex* is defined as *uindex* = (*uindex*[1], *uindex*[2],⋯, *uindex*[θ]). Let θ denote the number of related USN entries. Define *P*[*i*].*section* as the section information of the software *P*[*i*], and *P*[*i*].*priority* is the priority of *P*[*i*]. Note that both *P*[*i*].*section* and *P*[*i*].*priority* are easily obtained from PKG-MIB. Also, *U*[*i*] has the information of related CVE names *C_name*. Let *C_name* = {*C_name*[1], *C_name*[2],⋯, *C_name*[ζ]}, where ζ is the number of names (see Table 8). Thus, the list of CVE names is defined as *U*[*i*].*C_name*[*j*]. The CVE structure is defined as *C* = {*C*[1], *C*[2],⋯, *C*[κ]}, where κ is the total number of entries in the CVE structure, and each CVE structure entry has its status *C*[*i*].*status*.

The purpose of algorithm 2 is to represent the risk of

vulnerable software and the total risk of the host by numerical value. We need *P*[*is*].*section*, *P*[*is*].*priority*, and the weight of the related CVEs to calculate the risk. The section and priority information is indexed from PKG-MIB by *V*[*i*].*pindex*, which indicates the index of the vulnerable software. The section and priority information is converted into the numerical values *weight_section* and *weight_priority* as shown in Table 4. To obtain the total weight of CVEs, *totalCve*, we need to retrieve all of the related CVE entries. We know the index of related USN entries *V*[*i*].*uindex*. The related CVEs for each USN entry *U*[*usn_num*].*C_name* is retrieved from *U*. For each *C_name*, the status of the CVE entry *C*[*cve_name*].*status* is identified as either "Candidate" or "Entry." If the status is "Candidate," 0.7 is added to *totalCve*. Otherwise, 1 is added to *totalCve*. With *weight_section*, *weight_priority*, and *totalCve*, we can calculate the risk of the software by (2). This calculation is iterated β times to evaluate how vulnerable the node is. The output of algorithm 2 is η which indicates the risk weight of the host.

B. VUL-Patch Module

The main function of VUL-patch executes APT to patch vulnerable software detected by algorithm 1. An algorithm for the vulnerability patching function is shown in algorithm 3, where the input *V* is obtained from algorithm 1, and the output *S* is the result of the patch executions. Here, *S* is defined as $S = \{S[1], S[2], \cdots, S[\beta]\}$.

We know which software of the host is vulnerable from *V*, that is, the output of Algorithm 1. For each vulnerable program, patching is performed by using APT. We need the name of the vulnerable software as the parameter of APT. The name is obtained from *P* and indexed by *V*[*i*].*pindex*. Thus, the name of vulnerable software is represented as *P*[*V*[*i*].*pindex*].*name*. For each *P*[*V*[*i*].*pindex*].*name*, patching is performed. The results of each patch are stored in *S*. Patching is performed β times because the number of the vulnerable programs is β.

Algorithm 3. Vulnerability patch

```
1:    begin
2:    input V
3:    for i ← 1 to L do
4:       usname ← P[V[i].pindex].name
5:       execute APT to path usname
6:       if the result of execution is OK then
7:          S[i] ← success
8:       else
9:          s[i] ← failure
10:      end if
11:   end for
12:   output S
13:   end
```

V. Evaluation

1. Applicability of PKG-VUL

We created a prototype PKG-VUL framework on Ubuntu Linux hosts which was implemented as Java applications within Java SDK v5.0 and Tomcat. To demonstrate the applicability of PKG-VUL, we needed to evaluate how applicable PKG-VUL is to real-world vulnerabilities. For this purpose, we evaluated three different versions of Ubuntu Linux. We installed PKG-MIB and a VUL-check module on each node which has one of the versions of Ubuntu Linux installed. On another node which was assigned the commend center role, we executed a VUL-check request. The results are shown in Tables 5 and 6.

One notable result of the vulnerability evaluation of the three versions of Ubuntu Linux shown in Table 5 is that a node running Ubuntu Linux v6.10 is more vulnerable than a node running Ubuntu Linux v6.06 even though Ubuntu Linux v6.10 is the latest version of Ubuntu Linux. Any security measure should extend to keeping the latest OS secure against attacks. That is not always helpful for defense since most attackers do not attempt to exploit the OS itself. However, software running on the OS may still be vulnerable. As previously mentioned, more than 90% of attacks exploit the known vulnerabilities.

That is, the target of an attacker is generally vulnerable software, not the OS itself. Therefore, the VUL-patch module of PKG-VUL provides a valid method for defense against attacks. Also note that the vulnerabilities of libnspr4 and Firefox are detected in all Ubuntu Linux versions (see Table 6). Note that libnspr4 is a runtime library for Firefox. It shows the most recent trend of exploits. These days, attacks exploit web-based software or systems; therefore, reports of web-related

Table 5. Result of vulnerability evaluation.

	Ubuntu v5.10	Ubuntu v6.06	Ubuntu v6.10
α	1,036	1,235	1,281
β	4	3	4
γ	78	32	29
δ	0	1	1
ε	2	1	1
ζ	2	1	2
η	243	138	155

α: Number of programs installed on a host
β: Number of vulnerable programs
γ: Number of vulnerabilities
δ: Number of system vulnerabilities
ε: Number of network vulnerabilities
ζ: Number of vulnerabilities included both of sys. and net.
η: Risk weight of a node

Table 6. Details of vulnerable software running on each Ubuntu Linux.

Ubuntu Linux v5.10				
Name	Version	Sec.	USN	Risk
libnspr4	2:1.7.12-0ubuntu2	libs	USN-361-1	44
libnss3	2:1.7.12-0ubuntu2	libs	USN-361-1	44
Firefox-gnome-support	1.0.7-0ubuntu2	web	USN-354-1	66
Firefox	1.0.7-0ubuntu2	web	USN-354-1 USN-381-1 USN-398-2 USN-398-4	89
Ubuntu Linux v6.06				
Name	Version	Sec.	USN	Risk
avm-fritz-firmware	3.11+2.6.15.11-3	rest/misc	USN-346-2	30
libnspr4	2:1.firefox1.5.dfsg+1.5.0.5-0ubuntu6.06.1	libs	USN-381-1 USN-398-2	44
Firefox	1.5.dfsg+1.5.0.5 - 0ubuntu6.06.1	web	USN-351-1 USN-381-1 USN-398-2	64
Ubuntu Linux v6.10				
Name	Version	Sec.	USN	Risk
libnspr4	2:1.firefox2.0+0dfsg-0ubuntu3	libs	USN-398-1	39
Linux-image	2.6.17-10.33	base	USN-395-1	40
Firefox	2.0+0dfsg-0ubuntu3	web	USN-398-1	44
Linux-restricted-modules	2.6.17.5-11	rest/misc	USN-404-1	32

vulnerabilities are increasing. Finally, Ubuntu Linux v6.10 has serious vulnerabilities related its kernel. According to CVE-2006-6332, a remote attacker could send a specially crafted packet and execute an arbitrary code with root privileges under the vulnerability.

2. Qualitative Comparison

In this subsection, we provide the comparison results in which previously developed defense methods are compared with PKG-VUL qualitatively.

As shown in Table 7, which is based on the comparison table in [20], Snort and Shield operate in reactive mode. As network traffic passes through a network interface, Snort and Shield examine the traffic. Accordingly, they have an impact on latency and throughput. On the other hand, Nessus and PKG-VUL operate in proactive mode so that vulnerability scanning/patching is required to run at regular intervals. Therefore, Nessus and PKG-VUL only impact the system load during scanning/patching.

Table 7. Comparison of various defense methods.

	Snort [11]	Shield [8]	Nessus [12]	PKG-VUL
1	NIDS	Vulnerability checking filters	Vulnerability scanner	Vulnerability scanner and patcher
2	Reactive	Reactive	Proactive	Proactive
3	Typically only examines network level	Examines network level	Can search for application level	Can search for application level, also provide related software information
4	All network traffic is examined: impact latency and throughput	All network traffic is examined: impact latency and throughput	Vulnerability scan runs at regular intervals: impact on system load only during scanning	Vulnerability scan and patch run at regular intervals: impact on system load during scanning and patching
5	Requires regular updates and associated configuration	Requires regular updates and associated configuration manually	Requires regular updates	Requires regular updates
6	No	Yes	No	Yes
7	No	No	No	Yes
8	Complex	Complex	Simple	Simple
9	High	High	Medium	Low

1: Classification
2: Operation mode
3: Awareness of high level function
4: Impact on normal operation on host
5: Handling of attacks
6: Ability to patch
7: Ability for integrating to SNMP based network managements
8: Complexity of setup
9: complexity of maintenance

VI. Conclusion

We have presented the security vulnerability evaluation and patch framework, called PKG-VUL, which is feasible to implement with agility, usability, and manageable scalability. The results of our performance evaluation demonstrated that PKG-VUL has broad applicability to various versions of Linux. We demonstrated how easy it is for system administrators to automatically evaluate and patch their vulnerable software. We believe that it is only a matter of time before attackers start using automated vulnerability scanning tools or reported vulnerability-specific signatures to discover vulnerable hosts.

Such vulnerable hosts would have vulnerable software and could be used as zombie hosts to find other vulnerable hosts. With this paper, we hope to raise awareness and provide a valid method for system administrators faced with security problems to proactively evaluate and patch their vulnerable software.

In our future work, we will concentrate on extending PKG-MIB to Microsoft Windows, and thus enable PKG-VUL to operate on Microsoft Windows.

References

[1] E. Rescorla, "Security Holes... Who Cares?" *Proc. 12th USENIX Security Symposium*, Aug. 2003, pp. 75-90.

[2] J.R. Crandall, Z. Su, and S.F. Wu, "Intrusion Detection and Prevention: On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits," *Proc. 12th ACM Conf. Computer and Communications Security*, Nov. 2005, pp. 235-248.

[3] Website of Common Vulnerabilities and Exposures, http://cve.mitre.org (accessed Jan. 2008)

[4] J.-H. Lee et al., "PKG-MIB: Private-mib for Package-Based Linux Systems in a Large Scale Management Domain," *Lecture Notes in Computer Science*, vol. 4496, May 2007, pp. 833-840.

[5] CERT Advisory for Code Red Worm, http://www.cert.org/advisories/CA-2001-19.html (accessed June 2009).

[6] CERT Advisory for Nimda Worm, http://www.cert.org/advisories/CA-2001-26.html (accessed June 2009).

[7] CERT Advisory for MS-SQL Worm, http://www.cert.org/advisories/CA-2003-04.html (accessed June 2009).

[8] H. J. Wang et al., "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits," *Proc. ACM SIGCOMM,* Aug. 2004, pp. 193-204.

[9] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Annual Network and Distributed System Security Symposium*, Feb. 2005.

[10] S. Sidiroglou et al., "Building a Reactive Immune System for Software Services," *Proc. USENIX Annual Technical Conference*, Apr. 2005, pp. 149-161.

[11] Snort, http://www.snort.org (accessed June 2009).

[12] Nessus, http://www.nessus.org (accessed June 2009).

[13] Website of Computer Emergency Response Team (CERT), http://www.cert.org (accessed Jan. 2008).

[14] Website of IBM Internet Security Systems (ISS), http://www.iss.net (accessed June 2009).

[15] W.A. Arbaugh, W.L. Fithen, and J. McHugh, "Windows of Vulnerability: A Case Study Analysis," *IEEE Computer*, vol. 33, no. 12, Dec. 2000, pp. 52-59.

[16] A. Arora, A. Nandkumar, and R. Telang, "Does Information Security Attack Frequency Increase with Vulnerability Disclosure? An Empirical Analysis," *Information Systems Frontiers*, vol. 8, no. 5, Nov. 2006, pp. 350-362.

[17] G. Noronha Silva, "APT HOWTO," http://www.debian.org/doc/manuals/apt-howto (accessed June 2009).

[18] B. Arumugam, "Ubuntu Server Guide v6.06," https://help.ubuntu.com/ubuntu/serverguide/C (accessed June 2009).

[19] Website of Ubuntu security notices (USN), http://www.ubuntu.com/usn (accessed June 2009).

[20] R. Davies, "Firewalls, Intrusion Detection Systems and Vulnerability Assessment: A Superior Conjunction?" *Network Security*, vol. 2002, no. 9, Sept. 2002, pp. 8-11.

Jong-Hyouk Lee received his BS degree in information system engineering from Daejeon University, Daejeon, Korea, in 2004, and his MS degree in computer engineering from Sungkyunkwan University, Suwon, Korea, in 2007. He is a PhD student in electrical and computer engineering at Sungkyunkwan University. He is currently working on the development of attack scenarios and methods to counter the attacks in NEMO-based vehicle environments with the IMARA Team at INRIA, France. His research interests include mobility management, security, and performance analysis for next-generation wireless mobile networks.

Seon-Gyoung Sohn received the BS and MS degrees in computer science from Chonnam National University, Gwangju, Korea, in 1999 and 2001, respectively. Since 2001, she has been with the Information Security Research Division at ETRI, where she is a senior member of research staff. Her research interests are in the areas of network security, network traffic analysis, and security situation monitoring.

Beom-Hwan Chang received the BS, MS, and PhD degrees in electrical and computer engineering from Sungkyunkwan University, Seoul, Korea, in 1997, 1999, and 2003, respectively. Since 2003, he has been with the Information Security Research Division at ETRI, where he is a senior member of research staff. His research interests are in the areas of network security, network traffic analysis, and security situation awareness.

Tai-Myoung Chung received his first BS degree in electrical engineering from Yonsei University, Korea, in 1981, and his second BS degree in computer science from University of Illinois, Chicago, USA, in 1984. He received the MS degree in computer engineering from University of Illinois in 1987 and a PhD in computer engineering from Purdue University, W. Lafayette, USA, in 1995. He is currently a professor at Sungkyunkwan University, Suwon, Korea. He is now vice-chair of the Working Party on Information Security & Privacy, OECD, and a senior member of IEEE. He also serves as a presidential committee member of the Korean e-government, the chair of the information resource management committee of the e-government. He is an expert member of the Presidential Advisory Committee on Science and Technology of Korea and is chair of the Consortium of Computer Emergency Response Teams (CERTs). His research interests are in information security, networks, information management, and protocols of the next-generation networks such as active networks, grid networks, and mobile networks.