Foundations for Intrusion Prevention

Shai Rubin, Ian D. Alderman, David W. Parter, and Mary K. Vernon
University of Wisconsin, Madison

Abstract: We propose an infrastructure that helps a system administrator to identify a newly published vulnerability on the site hosts and to evaluate the vulnerability's threat with respect to the administrator's security priorities. The infrastructure foundation is the vulnerability semantics, a small set of attributes for vulnerability definition. We demonstrate that with a few attributes it is possible to define the majority of the known vulnerabilities in a way that (i) facilitates their accurate identification, and (ii) enables the administrator to rank the vulnerabilities found according to the organization's security priorities. A large scale experiment demonstrates that our infrastructure can find significant vulnerabilities even in a site with a high security awareness.

1 Introduction

To date, worms and other widespread network-based attacks have gained unauthorized access to many organizations' hosts by exploiting known vulnerabilities. For example, the 'Code Red' worm—which spread in July and August 2001—exploited a known buffer overflow vulnerability on thousands of hosts that were running Microsoft servers [CCZ02, SPW02]. The damage could have been avoided if the host administrators had installed the software patch that was announced by Microsoft approximately one month before the worm was released.

While intrusion detection systems (e.g., [Ro99]) can detect an attack such as the Code Red, they usually cannot prevent an attack from occurring. To prevent attacks, we envision an *Intrusion Prevention Infrastructure* (IPI), depicted in Figure 1. As soon as a new vulnerability is published, the IPI detects the vulnerability on every system host, estimates the vulnerability threat with respect to the site security priorities, and aids the administrator in repairing the vulnerability. The infrastructure core is a vulnerability database which provides a vulnerability definition that facilitates and integrates the other infrastructure components: an accurate audit tool, a site-customizable threat analyzer, and a repair tool that derives repair options from the vulnerability definition.

This paper focuses on two IPI components: the vulnerability database, with its language for defining vulnerabilities, and the threat analyzer. More particularly, we make the following contributions:

Vulnerability semantics (Section 4). We propose a formal language for vulnerability definition. The language core comprises two sets of attributes: the *presence* attributes define the characteristics of a vulnerable host and the *threat* attributes define the severity of the

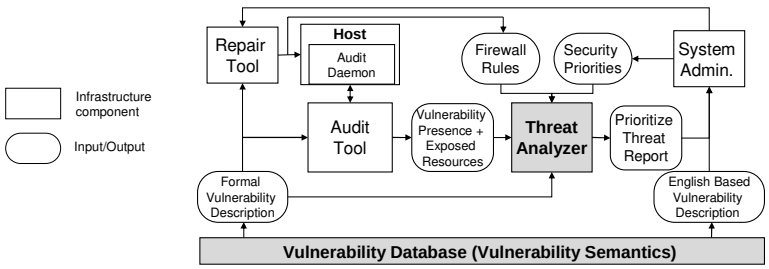

Figure 1: Intrusion Prevention Infrastructure. In this paper, we develop the threat analyzer and the vulnerability semantics.

vulnerability and the difficulty of exploiting it. Since our language is formal, it facilitates unambiguous vulnerability definition and accurate audit; since it is simple, repair options can be derived from the vulnerability description by nullifying the presence or threat conditions. Unlike other languages (e.g., [MIb]), our attributes do not rely on specific host or operating-system features, so they support an efficient audit. At the same time, our language enables threat definition that depends on the host configuration. For example, one can define a conditional threat: "if file F exists, the threat is high, else the threat is low".

Threat analyzer (Section 5). We develop a *site-customizable* threat analyzer that ranks vulnerabilities according to the site security preferences. Our first threat measure is the *attack severity*, a quantitative measure of the resources a vulnerability exposes. To identify these resources, the analyzer uses both the vulnerability semantics and the audit results; to quantify them, it uses a resource ranking which is customized according to the site security preferences. The second threat measure is the *attack difficulty*, an estimate of the attacker ability to launch a successful attack. We split the difficulty definition into two components. The *inherent* component represents the vulnerability difficulty by comparing the vulnerability to others that are similar. The *site-specific* difficulty reflects the difficulty of exploiting the vulnerability on a particular host.

IPI case study (Section 6). We study the IPI feasibility, its necessity, and its impact on security. We partially implemented the IPI audit and threat tools, and we used the tools to conduct a long term experiment on a real site. Our site has 1500 hosts, maintains strong configuration management (only software that is authorized by the site administrator can be installed), and has a security officer who devotes his entire time to ensuring the site security. The results demonstrate the IPI's capability of finding long-lived vulnerabilities, even for a site with such a high security awareness. The study demonstrates the value of a site-customizable threat analyzer for consistently measuring threat levels over a long period of time. The threat analyzer measures both the impact of repairing vulnerabilities and the threat levels of vulnerabilities that were not repaired because they are allowed by the site security policy.

2 Related Work

Vulnerability databases. There are several vulnerability databases that contain hundreds [CE, De] to thousands [Th, Se, In] of vulnerability descriptions. These databases primarily use a natural language (i.e., English) to describe vulnerabilities. As we illustrate in the next section, such descriptions are subject to different interpretations by different people. Furthermore, a natural language description is difficult to use by programs that can automate the audit, repair, and threat analysis processes. In contrast, our language is formal, so it enables unambiguous vulnerability definition and facilitates the use of automatic tools.

Vulnerability definitions and languages. Cuppens et al. [CO00] propose a formal language to describe vulnerabilities. However, they focus on vulnerability composition to detect a series of exploits that form a significant attack [CM02]. Their language does not include semantics that support accurate audit or repair of a vulnerability, and is limited in the threat characterization for each vulnerability.

Parallel to our work, MITRE corporation is developing the Open Vulnerability Assessment Language (OVAL) [MIb]. OVAL uses SQL to define a host-dependent test that determines whether or not a host is vulnerable. Unlike OVAL, our higher-level definition supports threat analysis and vulnerability repair in addition to testing for the presence of the vulnerability. Furthermore, our presence attributes facilitate a network-based audit technique that is host independent. Network-based auditing is identical for all hosts and operating systems, and is needed for efficient testing.

Quantitative threat analysis. Existing databases and audit tools specify, for each vulnerability, a *fixed* quantitative threat level like 'high', or 'low' (the CERT database uses fixed numeric ratings). As far as we can tell, the meaning of these different levels is not precisely defined in any of these tools and there are inconsistencies among the tools (e.g., the threat of CVE-2000-0614 is 'high' by ICAT [Th] but 'medium' by ISS [In]). In our threat model, the threat level depends on the identity of the resources the vulnerability exposes rather than on a fixed value given by the vulnerability definer. This means that (i) a vulnerability can yield different threat levels when found on different vulnerable hosts (e.g., web server vs. user workstation), and (ii) different vulnerabilities that expose the same resources have the same threat level.

Ortalo et al. [ODK99] define a vulnerability threat based on the difficulty of exploiting the vulnerability; they assign a fixed difficulty level to each vulnerability. Our threat definition is broader: it is based on the resources the vulnerability exposes in addition to the vulnerability difficulty. Furthermore, our difficulty definition is customizable: it takes into account site-dependent factors such as host configuration and firewall protection.

Audit and testing technologies. Two types of audit technologies have been developed: (i) host configuration checkers that search for host configuration flaws that might be exploited by attackers [FPA98, FS90, KS94, Or99, SSH93, ZL96], and (ii) remote network-based vulnerability scanners which test for vulnerabilities exclusively using the network protocols [De, Mu95, In, To]. Since neither of these techniques alone can conclusively identify all vulnerabilities, these tools produce many false positives. To increase accuracy, the IPI audit methodology is based on precise semantics, and uses information from both host and

network audit techniques. However, we leave the detailed implementation of this audit methodology to the future.

3 Motivation: Imprecise Vulnerability Definition

We demonstrate that a natural-language vulnerability definition—used by all contemporary vulnerability databases—lacks the ability to define a vulnerable host and the vulnerability threat. Consider the natural language description from the ICAT database [Th]:

***GuestBook* vulnerability (CAN-1999-1053 [MIa]).** The *guestbook.pl* is a CGI script that enables visitors to sign an online guest book. A security hole, when guestbook.pl is run on Apache 1.3.9 and possibly other versions, enables users to insert, instead of their names, shell commands—called Server-Side Includes (SSI). Consequently, attackers can execute arbitrary commands on the host. However, such commands will be executed only if the Apache web server enables SSI, and guestbook.pl is configured to accept HTML tags.

This description leads to three possible definitions of a vulnerable host. A *first* definition could be: (i) the Apache web server is running, (ii) the Apache web server is configured to enable SSI, (iii) at least one user is using the guestbook.pl script, and (iv) HTML tags are enabled in that guestbook.pl script. A *second,* broader definition can ignore the third and the fourth conditions. After all, since any user at any time can install the guestbook.pl script, a host can be viewed as vulnerable even if the script is not currently installed. Lastly, one might claim that the *GuestBook* vulnerability represents a family of vulnerabilities which can be described by a *third* definition broader than the first and narrower than the second—conditions (i), (ii), and (iii'): there exists a script accessible by Apache that does not sanitize SSI directives in its input.

Natural language description is not only ambiguous with respect to the presence of the vulnerability, but also imprecise with respect to the vulnerability threat. First, the commands the attacker inserts are executed under the privileges of the web server account. If Apache is running with 'root' privileges, the threat is higher than if it is running with 'nobody' privileges. Second, the threat depends on the conditions that define the presence of the vulnerability. *GuestBook* poses an immediate threat according to the *first* and *third* definitions above, but only a potential threat according to the *second* definition.

The *GuestBook* description lacks two other important properties a vulnerability definition must possess. First, a definition must facilitate accurate audit. We cannot use a vague phrase like 'possible other versions' to implement an accurate audit procedure. Second, to facilitate rapid repair, the definition must include simple and accurate repair options.

4 Vulnerability Semantics

A system administrator considers a vulnerability in the context of a system; an administrator may choose not to repair the *GuestBook* vulnerability, because the vulnerable host

is behind a firewall. The goal of the vulnerability semantics is to specify how various attributes influence the meaning of a vulnerability, and enable the audit tool and threat analyzer to fill in the attributes with values. For example, the semantics specifies the vulnerability protocol, and the audit tool fills in whether this protocol is blocked by a firewall for a given host, or, in the *GuestBook* case, the semantics specifies that the threat depends on the account that runs the Apache process, while the audit tool determines who the account owner is (root vs. nobody).

To achieve flexibility in definition, the vulnerability semantics contains four attribute sets. The *identification* attributes specify key characteristics of the vulnerability in a human-readable form. The *presence* attributes define sufficient and necessary conditions to determine whether a host is vulnerable. The *threat* attributes define system resources compromised by the vulnerability and the difficulty of exploiting it. Lastly, the *repair* attributes define how to immunize a vulnerable host; this set is beyond the scope of this paper.

4.1 Vulnerability Identification

This is a set of attributes that provides a human-readable description of the vulnerability; it is not a complete set of properties that characterize a vulnerable host. The attributes in this set are (Table 1): (i) *Name*: provides a vulnerability-unique identifier (e.g., CVE name [MIa]) (ii) *Operating system*: provides a list of operating systems that are known to be vulnerable; (iii) *Vulnerable unit*: provides a list of the software components that should be repaired; (iv) *Configuration*: provides an informal description of configuration

Identification	Name	Apache GuestBook (CAN-1999-1053)	Telnet cleartext passwords (CAN-1999-0619)
	Operating system	ANY	ANY
	Vulnerable unit	Apache version 1.3.9, guestbook.pl.	Telnet
	Configuration	Server Sides Include (SSI) on.	
	Protocol,Port	RFC: 2616/HTTP,80+'any'	RFC: 854/TELNET,23
Presence	Condition Set	P_1: *serviceRunning()* P_2: $package{=}Apache$ P_3: *content*(*config_file*, $[Includes\|XBitHack]$)	P_1: *serviceRunning()*
	Verification Hints	$H_{3,UNIX}$: (*config_file*= '/etc/httpd/conf/httpd.conf')	
Threat	Exposed Resources	if (version=1.3.9) then <*CIA,SA*> else UNKNOWN	<*CIA,NPA*>

	Expected Time to Exposure (days)	if (access(guestbook.pl) or *content(guestbook.pl, 'html=1'))* then 0;
		else
		TimeUntil(guestbook.pl installed)=30
			TimeUntil(a user uses TELNET from sniffed network)=7.

	Expected Time to Attack (days)	7	0.5

Table 1: Examples of vulnerabilities definitions.

settings of the vulnerable units, like Apache with SSI turned on; and (v) *Protocol/Port*: specifies the name and RFC number of the vulnerability's application-level protocol (e.g., HTTP/2616) and the port number the vulnerability uses. The port is either an integer (e.g., 21 for FTP), or the word "any" if the protocol standard does not dictate a specific port. In the latter case, we also specify the default port (e.g., 80 for HTTP).

4.2 Presence Semantics

The vulnerability presence semantics has two components. The first is a set of predicates, the *condition set*, that specifies necessary, sufficient, and verifiable conditions that must hold in order for a host to be vulnerable. The second is a set of *verification hints* the audit tool can use to verify these conditions.

4.2.1 Condition Sets.

Out of the hundreds of vulnerabilities we reviewed, 90% of them can be defined using the predicates in Table 2. We use predicates that assert configurational properties only, functional properties only, or both.

Since audit tools usually use a functional test—a test that uses the means the attacker uses [De]—to verify functional predicates, using functional predicates in the vulnerability presence definition is preferable. When this is not feasible, because a functional test is either unsafe (e.g., requires buffer overflow exploitation) or inefficient, configurational predicates can be used. For example, the most efficient way to verify that the SSH service enables Kerberos authentication is by checking the SSH configuration file (P_4 in *SSH* in Table 5). To facilitate flexible and efficient audit, we permit two condition sets; in the *WebSitePro* vulnerability (Table 5) we specified two (logically equivalent) sets, and the audit tool can select the set that it believes is more efficient to check.

The condition set relies on the assumption that the site uses an official software release. If

Predicate	**Evaluated to true if and only if**	**Type**
serviceRunning()	the service specified by the Protocol attribute is running.	functional
access(application, p)	*application* is accessible using the service specified by the Protocol attribute **given** the predicate p is true.	
$Login(user, password)$	login with the pair (*user,password*) succeeded.	
content(file, regExp)	regular expression *regExp* is found in $file$.	config.
version() < v	version of the network service package is smaller than v. Other binary operators (e.g., $=, >, \leq$) are supported.	
package() = name	the name of the software package that provides the network service is $name$.	
hostResponse(command,regExp)	after executing command com the respond contains the regular expression $regExp$.	config. + functional
netResponse(message, regExp)	after sending the message msg the respond contains the regular expression $regExp$.	

Table 2: Predefined basic and compound predicates for presence and threat semantics.

the site changed the software (e.g., by modifying the version number, service banner, etc.), we classify such changes as user software that is not widely distributed and is not covered by the standard IPI audit.

4.2.2 Verification Hints.

To help the audit tool verify the predicates in the condition set, we add verification hints to the vulnerability presence definition. We define two categories of verification hints: *informational* and *logical*.

An informational hint suggests the location of a resource that the audit tool can use to verify predicates. For example, $H_{3,UNIX}$ in the *GuestBook* definition (Table 1) points to the Apache configuration file required by predicate P_3 (an additional hint can be provided for a Windows machine). A logical hint specifies either a *netResponse* or a *hostResponse* predicate that can be used to verify the predicates in the condition set. For example, to verify the predicates P_1 and P_2 in *Perl-In-CGI* (Table 5), the audit tool can use $H_{1,2}$ that specifies a *netResponse* predicate that if it holds, P_1 and P_2 hold too. Note that $H_{1,2}$ is a *sufficient* hint only: if it fails, P_1 and P_2 may still be true. We can specify *necessary and sufficient* hints; for example, $H_{2,3,UNIX}$ in the *LPRng* example (Table 5) specifies a *hostResponse* predicate which holds if and only if both P_2 and P_3 hold. To increase the audit flexibility, it is possible to specify more than one hint for predicates. For example, $H_{2,3,UNIX}$ (a host dependent hint) and $H_{2,3}$ (a host independent hint) can be used to verify P_2 and P_3 in the *SSH* example (Table 5).

There is no clear border between necessary-and-sufficient hints and predicates in the condition set; the vulnerability definer has the power to 'upgrade' such hints into predicates. However, our guideline is to leave the condition set, to the extent possible, host independent.

4.3 Threat Semantics

The threat semantics has two components: the *vulnerability severity* and the *vulnerability difficulty*. A novel feature of our semantics is the use of the same predicates that define presence to define the threat attributes.

4.3.1 Vulnerability Severity.

The vulnerability severity is the extent to which an attacker gains unauthorized privileges on various system resources. Hence, the *Exposed Resources* attribute specifies privileges on resources that the attacker gains. In the *Telnet* example (Table 1), the attacker gains Confidentiality, Integrity and Availability (CIA) privileges (i.e., reading, modifying, and blocking access, respectively) to all resources associated with a particular non-privileged user. In *WebSitePro* (Table 5), the attacker gains only confidentiality privileges to the directory names of the web server.

Reg. Expression	Resource Definition
[*'PA'* | *'NPA'* | *'SA'* | *UserName*]	Denotes resources associated with user accounts. The strings '*PA*' and '*NPA*' distinguish between privileged accounts (e.g., root or administrator) and regular user accounts. '*SA*' denotes an account of the owner of the vulnerability network service (e.g., *SA=root* if the web server is running with the root privileges). *UserName* is used to specify a specific user account (e.g., 'john' or 'guest').
File	Specifies a regular expression for a set of files or directories.
HostIp(.*RFC*)	Denotes the physical device associated with an IP address *HostIp* (e.g., a workstation, a printer). *RFC* is optional and specifies the protocol of the compromised service (e.g., 2616 for HTTP).
root_directoy	The root directory associated with the vulnerability network service (e.g., root directory of a web server).
[*'user_data'*| *'file_data'*| *'directory_data'*]	Denotes compromised *data* about users' files, or directories.
config_file	Denotes the configuration file of a vulnerable network service.

Table 3: Predefined resources.

We used the resources listed in Table 3 to define hundreds of vulnerabilities. Besides facilitating the customizable threat model, our 'exposed resources' approach has two other advantages. First, it enables us to specify resources in a fine-grained manner. For example, <*CIA,john*> specifies *CIA* privileges of the account with the name *john*, while <*CIA,/home/john/**> specifies *CIA* privileges to all files under the directory '*/home/john*'. Second, using presence predicates to express conditions on the exposed resources attribute further increases our definition expressiveness. In the *GuestBook* example, it is not clear whether Apache versions other than 1.3.9 are vulnerable too (Section 3). So, we put the *version* predicate in the exposed resources attribute rather than in the condition set (Table 1). The result is a vulnerability definition that is not limited to a particular version when it is unknown whether other versions are vulnerable too (as more knowledge becomes available, the definition may change).

4.3.2 Attack Difficulty.

Our difficulty definition is based on two observations. First, some vulnerabilities are intrinsically more difficult to exploit than others. The *GuestBook* can be exploited by any user familiar with a few UNIX commands, whereas exploiting a buffer overflow on an SNMP daemon (e.g., CAN-2002-0017) requires programming and network expertise. Second, the vulnerability difficulty is also host dependent. For example, it is very difficult to exploit a host behind a firewall. We use two attributes to define the intrinsic and host-specific difficulty.

The *Expected Time to Exposure* (ETE)—which captures the host-dependent difficulty—is the time it takes for a host to be vulnerable. In most cases, the ETE is 0 if all the presence predicates hold (e.g., *LPRng* in Table 5). In cases where additional user activity is required before a host becomes vulnerable, the ETE gets a non-zero value. For example, a user must install the *guestbook.pl* script before the *GuestBook* attack can take place. In such cases, the attribute value is '*timeUntil*(user-activity-description)=default' (Table 1). The

IPI database contains the default value for the *timeUntil* predicate, but the administrator can customize the value according to her familiarity with her system.

The *Expected Time to Attack* (ETA)—which captures the intrinsic vulnerability difficulty—is the expected time until an attacker gains unauthorized privileges, given that the host is vulnerable (i.e., no more user activity is required for the host to become vulnerable). In the case of *LPRng* (Table 5), the ETA is the time it takes to develop code that exploits the buffer overflow. For password sniffing (*Telnet* in Table 1), it is the time it takes to login *after* a user/password pair is known, which is essentially zero. For *GuestBook* (Table 1), it is the time until the guest book owner will upload the guest book and the attacker code will be executed (estimated as a week). We give examples of assigning the ETA value to the vulnerabilities we found in Section 6.1.

The vulnerability definer assigns both the ETA and ETE values (e.g., 1, 7 days). These values can be conservative, and can be refined from measures of exploit activity on the Internet (e.g., [BAMF01]). However, it is important to be consistent, and to assign similar values to vulnerabilities with similar difficulties.

4.4 Repair Semantics

The repair semantics is beyond the scope of this paper. However, since the presence semantics defines a set of necessary and sufficient conditions a vulnerable host must possess, to repair the host is to nullify at least one of the conditions. For example, software upgrade usually changes the software version number, so it nullifies the version predicate. This observation makes it easier to present to the system administrator the repair options, as we illustrate in Section 6.2.

5 Threat Analysis

In the previous section we presented the threat semantics for defining the vulnerability threat. Here, we discuss how the threat analyzer uses the semantics to estimate the vulnerability threat according to the site security priorities.

5.1 Ranking Severity

We assign to every resource in the system (e.g., file, host, printer) an 'exposure cost' that represents the damage to the organization if the resource is compromised (e.g., the cost for the company if its front web page is hacked). We assume that the audit tool reveals the exposed resources of each vulnerability, so we can map each exposed resource to its cost. We define the total severity of a vulnerability as the total cost of the resources the vulnerability exposes. More formally, the total severity of vulnerability v, TS_v, is defined as:

$$TS_v = \sum_{h \text{ vulnerable to } v} EC(h, <P, R>)$$

where h is a system host (e.g., server, printer), $<P, R>$ is a privileges-resource pair (Section 4.3.1) the attacker gains after exploiting v on h, and EC is the site *Exposure-Cost* function which maps the exposed resource $<P, R>$ on the host h to its exposure cost. The following properties make TS_v a good candidate for measuring severity:

1. The EC map counts for both the type of the vulnerable host and the resources the vulnerability exposes. This enables threat definition that is relative to both the vulnerability characteristics and the type of the vulnerable host. For example, EC can assign a higher cost to resources on an AFS server than to resources on a user workstation, so the threat of a given vulnerability is higher on the AFS server.
2. Usually, an administrator repairs a vulnerability by applying a patch to *all* vulnerable hosts. Since TS_v sums the costs over all hosts, it concisely shows to the administrator the 'security benefit' from his actions.
3. TS_v ensures that a resource is protected only if it is not exposed from any host. If v exposes the same resource from two different hosts, a situation that occurs in the presence of a shared file system, TS_v counts the resource twice. One might claim that this 'overestimates' v's severity, but it still reflects that removing the vulnerability necessitates repairing more than one host.
4. TS_v facilitates a simple and intuitive comparison between vulnerabilities. If v_1 and v_2 are vulnerabilities that expose the same resources on the same hosts, the definition ensures that TS_{v_1}=TS_{v_2}. Furthermore, if v_1 and v_2 expose different resources but still TS_{v_1}=TS_{v_2}, then even though v_1 and v_2 are different, their severity is still the same (with respect to the EC chosen). Similarly, because TS_v is additive, the administrator can easily understand that if TS_{v_1}=$100TS_{v_2}$ then v_1 is 100 times more severe than v_2.

5.2 Building an Exposure Cost Map.

We build the EC map by (i) qualitatively ranking sets of hosts according to the site security priorities, (ii) qualitatively ranking sets of exposed resources according to the site security priorities, and (iii) combining and quantifying the two rankings in a consistent way.

The IPI has a repository of rankings for hosts and exposed resources; any host ranking can be combined with any resource ranking. So, the administrator can select the rankings that come close to her system characteristics, refine them, and quickly build an EC function according to her security preferences. This process is simple enough to be done from scratch by the administrator, if desired.

To rank hosts, we create the $Hosts$ partial order which *qualitatively* ranks sets of hosts according to their exposure cost as defined by the site security priorities. Such a simple qualitative ranking—which fits many organizations—is illustrated in Figure 2(a). The ranking contains three levels, where the organizations' servers have the highest exposure cost and the employees' workstations the lowest.

To rank resources, we create the *Exposed Resources* (ERs) partial order which ranks sets

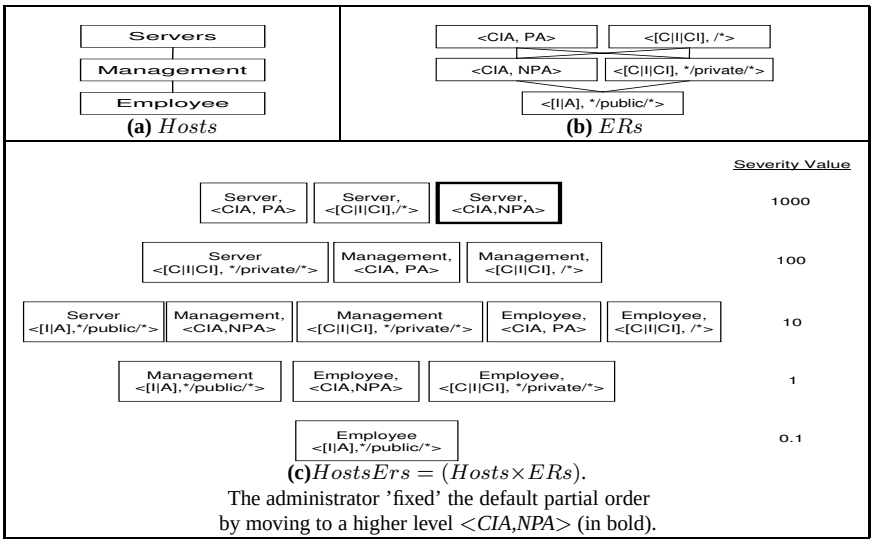

Figure 2: Building severity cost function.

of exposed resources according to the site security priorities, without considering the resource host component. For example, a site that emphasizes confidentiality and integrity can rank exposed resources using the ERs from Figure 2(b). At the lowest cost level, the site ranks availability and integrity of files (i.e., vulnerabilities that expose those files) in *public* directories. Next, the site ranks vulnerabilities that compromise non-privileged accounts. Since, according to the site policy, compromising user's private files is equivalent to compromising the user's account, the site ranks also into the second level vulnerabilities that expose either confidentiality or integrity (but not availability) of files in private directories. Last, the site assigns the highest exposure cost to vulnerabilities that expose privileged accounts, or expose the confidentiality or integrity of files which do not fall under the previous categories[1].

Both *Hosts* and ERs are neither unique nor as detailed as possible. As the system administrator becomes more familiar with the process, he can further partition the $Hosts$ or ERs. For example, he can add more server levels to $Hosts$, or move a specific exposed resource, like the availability of the company's main web page, to a higher level in ERs.

The next step is to build the product partial order: $Hosts \times ERs = HostsERs$. $HostsERs$ preserves the consistency of both $Hosts$ and ERs. But, it may not reflect exposure rankings that depend on a combination of hosts and resources. For example, servers are considered more valuable than user workstations, and in many cases compromising a non-privileged account on a server may cause the same damage as compromising a privileged account on the server. Hence, the administrator can move the exposure level of non-privileged accounts on servers to a higher level in the $HostsERs$ order (Figure 2(c)).

[1]To keep the example simple, we assume that all users' files are either under *public* or *private* directories. The example can be easily refined to include files that are not under one of these directories (e.g., by using the resource */home/*).

Note that the modified $HostsERs$ (Figure 2(c)) 'breaks' the consistency of the ERs partial order. That is, it is no longer true that the cost of any privileged account vulnerability is higher than any non-privileged account vulnerability. However, the $HostsERs$ helps to maintain the consistency of both $Hosts$ and ERs in the majority of the cases. The administrator will override the consistency only when security priorities dictate the inconsistency. In a realistic case, where $Hosts$ and ERs contain many more sets of hosts and resources, it is difficult to build an EC function which is consistent with both. In such cases, the importance of the $HostsERs$ order and the methodology to consistently build it from a product of two partial orders increases.

Since every level in both $Hosts$ and ERs represents the same exposure cost, it makes sense to assign to each level in $HostsERs$ the same cost too. A consistent way to do so is to assign higher exposure costs to higher levels in $HostsERs$. For example, we assign exposure costs with different orders of magnitude to each level in the $HostsERs$ from Figure 2(c). As we demonstrate in Section 6, such a simple scheme is very effective in pinpointing vulnerabilities with high severity.

5.2.1 Ranking Difficulties.

To quantify attack difficulty, we use a sum of the ETE and ETA values. Other (e.g., non-linear [ODK99]) measures are left for future work.

To account for visibility (i.e., firewall), the vulnerabilities could again be partitioned (within each difficulty ranking) by their visibility ranking. If there are many firewalls, the system administrator could assign a "trustworthiness"rating to the users of each firewalled subnet, where this rating is used as a multiplier for the expected time to attack (ETA) measure defined for the wide area Internet.

6 Pilot Study

We investigate two questions regarding the IPI. First, to what extent the IPI is needed. Second, to what extent the threat model can express the site security priorities, and whether such a model is useful as a quantitative security measure. To answer the first question, we test whether a 'repeated audit' approach is capable of identifying security holes in a site with relatively high security awareness. To answer the second question, we perform a field test of our threat model.

6.1 Experimental Methodology

We simulated the IPI, over a period of six months, on a site that contains more than 60 dedicated servers, and almost 1500 hosts running a variety of operating systems. The site administrators monitor mailing lists and web sites for security issues, and apply secu-

rity patches as quickly as possible. Most importantly, the site security practices emphasize *strong configuration management* (SCM)—all software installations are identical and users cannot install their own software on their workstations. Under SCM settings, the administration can focus security efforts on a (relatively) small set of software packages, so it is relatively easy to identify and patch vulnerabilities. Finding long-lived vulnerabilities in such a situation points to the necessity of the IPI.

We used Nessus [De] to audit the system over a period of six months. During the first five months we did not share the audit results with the administrators, but we modeled the vulnerabilities using our presence semantics and semi-automatically applied our threat analysis:

1. We analyzed each vulnerability Nessus reported and expressed its presence using our predicates (Section 4.2). Then, we analyzed the Nessus script that attempts to identify the vulnerability. If the script did not verify all the predicates necessary for the vulnerability identification, we classified this Nessus alarm as a false positive and removed the alarm from the audit results. For each vulnerability, we also specified the repair options we derived from the predicates (Section 4.4).

2. We assigned exposed resources (Section 4.3.1) to each vulnerability. According to our difficulty definition, we assigned an ETA value (intrinsic difficulty) to each vulnerability: 30 days to exploits that require large computational effort (e.g., dictionary attack), 7 days to vulnerabilities that require considerable programming effort (e.g., buffer overflows without a known exploit), and 1 day to exploits that are easily found on the web (e.g., buffer overflow with a known exploit) [2].

3. We removed from the audit results all vulnerabilities that cannot be exploited due to site firewall protection.

4. Together with the system administrators we built the $Hosts$, the ERs, and the final $HostsERs$ partial orders; an order that is similar in nature to the one illustrated in Figure 2(c). Then, we calculated the total threat (TS_v) of each true positive vulnerability.

The prioritized audit report for the fifth month was presented to the system administrators. To measure the impact of the audit results on the site security, we performed one additional audit a month later.

6.2 Results

Per month, Nessus reported about 190 different vulnerabilities on hundreds of different hosts. After we analyzed the reported vulnerabilities, we conclude that 75% of the alarms are false positives caused by one of the following reasons:

[2] We do not claim that these are the difficulties the IPI should use—future work should consider such issues. But, these values illustrate to the system administrator the differences in the inherent difficulty of the vulnerabilities found.

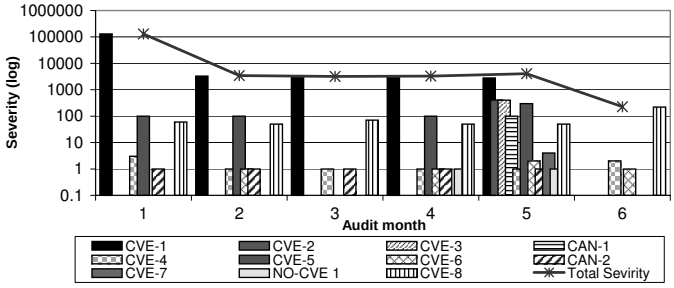

Figure 3: Severity of persistent vulnerabilities. Severity calculated by TS_v.

1. Inaccurate test. A presence predicate is not verified properly. For example, Nessus identifies the *TELNET* vulnerability (Table 1) by checking for the existence of the Telnet banner. However, the existence of a banner does not mean that Telnet login is possible. Indeed, the site administrators disabled Telnet, but to notify users about this, left a modified banner in place.
2. Incomplete test. One of the predicates in the condition set is not verified at all. For example, Nessus cannot verify predicates that require host configuration information as P_4 in the *SSH* vulnerability (Table 5).
3. Firewall protection. Nessus does not filter out vulnerabilities that cannot be exploited due to firewall protection[3].

We believe that the IPI eliminates these types of false positives. First, inaccurate tests are less likely to happen because the IPI audit stems from a precise semantics (Table 2). Second, configuration predicates, that are verified using a host-based audit daemon, eliminate false positives that require host configuration information. Lastly, we indent to incorporate firewall information into the threat analyzer.

Figure 3 presents the severe vulnerabilities found during our six month audit. To maintain the site confidentiality, we do not specify the vulnerability names or the $HostsERs$ partial order. Three observations should be noted:

1. Vulnerabilities of which the system administrator was unaware did pop up, even though the system used SCM. CVE-2, CVE-3, CVE-7, and CAN-1 appeared in the fifth month. During the experiment period CVE-6 appeared, was fixed (without the use of the audit results) and reappeared.
2. High severity vulnerabilities could pass undetected even in a site that used SCM. Between the first and second months, the administrator removed a severe vulnerability (CVE-1) from almost all hosts. This system repair was done without any audit information and reduced the severity level of the vulnerabilities in the system by two orders of magnitude. Nevertheless, two servers and three user workstations were inadvertently left exposed to CVE-1. During this time, despite CVE-1 existence on

[3]Nessus can identify these by performing two audits: one inside the firewall and one from outside the firewall. Clearly, this approach doubles the audit effort.

Name	Service	Threat Definition			Threat Analysis			Repair			Comments
		Exposed Resources	ETA (days)	ETE (days)	Servers	Non-Servers	Severity	Update	Reconfig	Block	
CVE-2	ftp	<*CIA,SA*>	1	0	7	11	410	√	-	√	buffer overflow
CVE-5	ftp	<*CIA,PA*>	1	0	-	3	300	√	-	√	buffer overflow
CVE-4	oracle tnslsnr	<*CIA,SA*>	1	0	-	1	1	√	√	-	easy password
NO-CVE 1	HTTP	<*user_name,C*>	1	0	-	1	1	-	-	√	misconfiguration
CVE-1	ssh	<*CIA,PA*>	7	0	2	8	2800	√	√	√	buffer overflow
CVE-3	ftp	<*CIA,NPA*>	7	0	3	-	400	√	-	√	buffer overflow
CAN-1	finger	<*CIA,PA*>	30	0	-	1	100	√	-	√	easy password
CVE-8	X11	<*CIA,NPA*>	30	0	-	5	50	-	√	√	easy password
CVE-7	HTTP	<*HostIp.2616A*>	30	0	-	4	4	√	-	√	buffer overflow
CVE-6	ldap	<*Host,A*>	30	0	-	2	2	√	-	√	buffer overflow
CAN-2	lpd	<*Host,A*>	30	0	-	1	1	-	√	√	easy password

Table 4: The threat analyzer report after the audit of the fifth month. Vulnerabilities are prioritizes first by their difficulty and then by their severity.

only a few machines, it imposed a risk that was an order of magnitude higher than all the other vulnerabilities combined. CVE-1 and other undetected vulnerabilities (e.g., CAN-2, CVE-5) were fixed only after the administrator got the audit results.

3. Our threat analysis accurately captured the security notion of the system administrator. After the administrator fixed the majority of the vulnerabilities in the audit report (Table 4), the total severity level dropped one order of magnitude (between the fifth and the sixth months in Figure 3), a decrease that was confirmed by the system administrator. The administrator reviewed the raw Nessus report and assured us that the vulnerabilities in Table 4 were the most severe ones.

   Other state of the art audit tools do not capture this major change in the threat level. For example, the Nessus report after the fifth month included 'high severity' vulnerabilities on 45 hosts and the sixth month report included 44. Due to vulnerable hosts that popped up before the sixth month audit (CVE-4 and CVE-8), the number of vulnerable hosts remained almost the same. However, the threat level was reduced considerably because the administrator considered the vulnerabilities that were fixed (i.e., CVE-1) much more severe than the vulnerabilities that popped up.

Table 4 presents the prioritized audit report (of the fifth month) our threat analyzer produces from the audit results. A few observations should be noted.

1. The severity of privileged-account vulnerabilities is not always higher than that of non-privileged-account ones. The severity of CVE-3, a non-privileged-account vulnerability found on three sensitive servers, is higher than CVE-5, a privileged-account vulnerability found only on user workstations.
2. Not all vulnerabilities with the same exposed resources (e.g., privileged account) have the same severity. Although CVE-1, CVE-5, and CAN-1 have the same exposed resources, their severity levels are considerably different.

3. Our audit report includes precise repair options that are derived from the vulnerability presence definition.

The above two sets of observations reveal the advantages of the proposed IPI. First, the IPI is based on the 'frequent audit' approach which is necessary to identify severe vulnerabilities even in a site that uses strong configuration management. Second, because our threat definition is based on exposed resources and because our threat analysis is sensitive to the number and type of vulnerable hosts, we can accurately capture the administrator security preferences. The analysis not only enables us to differentiate between vulnerabilities that traditionally had the same severity level (e.g., CVE-1 and CVE-5), but it also demonstrates that sometimes the severity of 'highly severe' vulnerabilities is actually less severe than that of 'less severe' ones.

7 Conclusion

Much remains to be done: automating the audit and repair processes, extending the threat model to fully support difficulty and visibility site-dependent parameters, and developing the vulnerability semantics to support all these activities. Building a robust IPI is a significant challenge, and we hope that the vision we have outlined here will inspire both academics and industry professionals to continue this work.

References

[BAMF01] Browne, H. K., Arbaugh, W. A., McHugh, J., and Fithen, W. L.: A trend analysis of exploitations. In: *IEEE Symposium on Security and Privacy*. Oakland, CA. May 2001.

[CCZ02] Cliff Changchun Zou, D. T., Weibo Gong: Code red worm propagation modeling and analysis. In: *ACM Conference on Computer and Communications Security*. Washington DC. November 2002.

[CE] CERT Coordination Center. Vulnerabilities, Incidents & Fixes. Available at `http://www.cert.org/`.

[CM02] Cuppens, F. and Miege, A.: Alert correlation in a cooperative intrusion detection framework. In: *IEEE Symposium on Security and Privacy*. Oakland, CA. May 2002.

[CO00] Cuppens, F. and Ortalo, R.: LAMBDA: A language to model a database for detection of attacks. In: *International Symposium on Recent Advances in Intrusion Detection (RAID)*. Toulouse, France. October 2000.

[De] Deraison, R. Nessus Security Scanner. Available at `http://www.nessus.org/`.

[FPA98] Farmer, D., Powell, B., and Archibald, M.: TITAN. In: *Large Installation System Administrator's Conference (LISA)*. Boston, MA. December 1998.

[FS90] Farmer, D. and Spafford, E. H.: The COPS security checker system. In: *USENIX Technical Conference*. Anaheim, CA. June 1990.

[In] Internet Security Systems. The Internet Security Scanner. Available at `http://www.iss.net/`.

[KS94] Kim, G. H. and Spafford, E. H.: Experiences with tripwire: Using integrity checkers for intrusion detection. In: *USENIX Applications Development Symposium*. Toronto, Canada. April 1994.

[MIa] MITRE Corporation. CVE: Common Vulnerabilities and Exposures. Available at `http://www.cve.mitre.org/`.

[MIb] MITRE Corporation. OVAL: Open Vulnerability Assessment Language. Available at `http://oval.mitre.org/`.

[Mu95] Muffett, A.: WAN – hacking with AutoHack – Auditing security behind the firewall. In: *USENIX Security Symposium*. Salt Lake City, UT. June 1995.

[ODK99] Ortalo, R., Deswarte, Y., and Kaâniche, M.: Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*. 25(5). September/October 1999.

[Or99] Ortalo, R.: Implementation of a distributed security monitoring tool: ESOPE. Technical Report 99506. Laboratory for Analysis and Architecture of Systems (LAAS). December 1999.

[Ro99] Roesch, M.: Snort – lightweight intrusion detection for networks. In: *Large Installation System Administrator's Conference (LISA)*. Seattle, WA. November 1999.

[Se] SecurityFocus Inc. Vulnerabilities database. Available at `http://www.securityfocus.com/bid/`.

[SPW02] Staniford, S., Paxson, V., and Weaver, N.: How to own the internet in your spare time. In: *USENIX Security Symposium*. San Francisco, California. August 2002.

[SSH93] Safford, D. R., Schales, D. L., and Hess, D. K.: The TAMU security package: An ongoing response to Internet intruders in an academic environment. In: *USENIX Security Symposium*. Santa Clara, CA. October 1993.

[Th] The National Institute of Standards and Technology (NIST). The ICAT Metabase. Avaliable at `http://icat.nist.gov/`.

[To] Todd, B. Sara: The security auditor's research assistant. Available at `http://www-arc.com/sara/`.

[ZL96] Zerkle, D. and Levitt, K.: NetKuang–A multi-host configuration vulnerability checker. In: *USENIX Security Symposium*. San Jose, CA. July 1996.

Identification	Name	Perl in CGI directory (CAN-1999-0509)	LPRng buffer-overflow (CVE-2000-0917)	SSH buffer-overflow (CAN-2002-0575)	Windows FTP backdoor (CAN-1999-0200)	WebSitePro root directory (CAN-2000-0066)
	Operating system	ANY	<RedHat,Linux,7.0>	<OpenBSD,OpenBSD, 3.1>	<MS,Windows,*>	<MS,Windows,*>
	Vulnerable unit	Perl	<Patrick_Powell,LPRng, <3.6.24>		Windows FTP	<OReilly Software, WebSitePro, <2.4.9>
	Configuration			Kerberos Authentication On	There exist an account with the name 'guest'.	
	Protocol Port	RFC: 2616/HTTP 80+'any'	RFC: 1179/LPDP 515+'any'	[a]/SSH 22+'any'	RFC: 959/FTP 21	RFC: 2616/HTTP 80+'any'
Presence	Condition Set	$P_1 : serviceRunning()$ $P_2 : access(perl)$	$P_1 : serviceRunning()$ $P_2 : version < 3.6.24$ $P_3 : package = LPRng$	$P_1 : serviceRunning()$ P_2:*version* < 3.2 P_3:*package* =*OpenSSH* P_4:*content*(*config_file*, AFSTokenPassing)	P_1 :*Login(USER=*, PASSWARD=*)*	Set 1: $P_1 : serviceRunning()$ $P_2 : version < 2.4.9$ $P_3 : package =$ *WebSitePro* Set 2: P_1:*netResponse(* ('GET /HTTP1.0 ', 'WebSitePro*404*: ||*windows_dir_name*'[b]
	Verification Hints	$H_{1,2,UNIX}$: *netResponse*('/cgi-bin/perl?-v', 'version') $\rightarrow (P_1 \wedge P_2)$	$H_{2,3,UNIX}$: *hostResponse*('lp -V', 'LPRng-'|| <'3.6.24')[c] $\leftrightarrow (P_2 \wedge P_3)$	$H_{2,3,UNIX}$: *hostResponse*('sshd -V', 'OpenSSH_'|| <'3.2') $\leftrightarrow (P_2 \wedge P_3)$ $H_{4,UNIX}$: *config_file*= '/etc/ssh/sshd_config') $H_{2,3}$: *netResponse*(*serviceRunning()*, 'OpenSSH_'|| <'3.2') $\leftrightarrow (P_2 \wedge P_3)$		
Threat	Exposed Resources	<*CIA,SA*>	<*CIA,PA*>	<*CIA,NPA*>	<*CIA,ftpguest*>	<*C,directory_data*>
	Expected Time to Exposure (days)	0	0	0	0	0
	Expected Time to Attack (Days)	1	1	1	0	1

[a] No RFC number yet for SSH.
[b] Put here a regular expression for a windows directory name.
[c] To conserve space we use the shorthand < to denote 'all versions smaller than', alternatively in reality all versions can be specified.

Table 5: Examples of vulnerability definitions.