CERIAS Tech Report 2001-03

Sharing Vulnerability Information using a Taxonomically-correct, Web-based Cooperative Database

L. Ma, S. Mandujano, G. Song, P. Meunier
Center for Education and Research in
Information Assurance and Security
Purdue University, West Lafayette, IN 47907

Sharing Vulnerability Information using a Taxonomically-correct, Web-based Cooperative Database

L. Ma, S.Mandujano, G. Song, P.Meunier
{lingfeng, sam, songg, pmeunier}@cerias.purdue.edu
Center for Education and Research in Information Assurance and Security (CERIAS)
Purdue University
W. Lafayette, IN 47907

February 12th, 2001

ABSTRACT

Software vulnerabilities are potential attack points in computing systems that can lead to considerable losses and severe security incidents. The way in which the information describing these vulnerabilities is handled is extremely important. Vulnerability data is very sensitive and therefore should be disclosed to the right people in the right circumstances. However, information sharing is currently mostly unidirectional; the present paper discusses a new approach for handling software vulnerability information: a cooperative system supported by a vulnerability classification. The system is composed by internal protocols that determine state transitions through which new vulnerability information is submitted, classified, verified, and made available via a Web Interface. Based on features like effects and nature, vulnerabilities in the collection can also be assigned a type. The proposed type system is a set of sub-classes that contain features of well-known vulnerability groups. Vulnerabilities can be linked together through these types and can be referenced as a group when retrieving or storing entries, hereby, speeding up the process. A voting mechanism allows a set of cooperating arbiters to review the information submitted from different sources. Approved descriptions of vulnerabilities can then be made available to the members of the cooperative system. The data model storing the vulnerability information is composed of a comprehensive set of features whose values are selected through decision trees. The leaves of the trees represent the most detailed qualities of a vulnerability.

1 Introduction

The prevention and removal of vulnerabilities constitute one of the main challenges in software development and computer security. Vulnerabilities may appear during the conception, the implementation or the deployment of software packages, and even through the interaction of separately designed packages. The growing complexity of software and the increasing number of interacting subsystems promises an unending discovery of new vulnerabilities. The exploitation of vulnerabilities may result in considerable losses [CSI/FBI survey].

Multiple vulnerability data banks and bug-tracking systems already exist. They provide information that helps identify, work around or fix vulnerabilities. The public ones have limited information that is usually selected in such a way as to strike a balance between helping system administrators and avoiding giving useful information to hostile entities. An example of this is the absence of exploits in information released by CERT (Computer Emergency Response Team, http://www.cert.org). The private ones that are known do not have mechanisms for easy sharing of information. Moreover, the sharing of vulnerability information with the maintainers of databases is often unidirectional.

During the 2nd Workshop on Research with Vulnerability Databases [Meunier1], security experts from different sectors discussed the requirements and obstacles of developing and maintaining vulnerability databases. The workshop supported the idea of carefully sharing vulnerability information so that it reaches the right people in the right circumstances without giving attackers more information than users have to protect and fix their own systems.

Multiple ideas were collected in the workshop from the various groups analyzing the different data sharing models that could be used with vulnerability information. From all the technical, consequential, and motivational challenges discussed, we identified a particular need regarding the systems that could successfully store and control this particular kind of information. A system capable of receiving vulnerability data submitted by multiple users was necessary. It was also clear that a classification of software vulnerabilities was mandatory in order to have a strong structure that allowed properly storing, querying, and expanding a vulnerability data collection. The quality of the information being stored was another important concern. Information accepted by an open system could be misleading or false if no screening process is applied. The problems would outweigh the benefits, therefore a rigorous control over the data needs to be enforced. There should be a process that provides confidence in the quality of the vulnerability information. A successful approach in information quality assurance is the appointment of a reviewer.

The design and construction of a vulnerability database system are not trivial. Besides the choice of taxonomy, access control, security, user interface issues and a review process, there is the question of sharing. Harm may come of not revealing it to those who need to know, and from revealing it to hostile agents. We suggest that the most important objective of vulnerability management is minimizing the "window of vulnerability", the interval between the times a vulnerability is discovered and the time at which it is fixed or the time a vulnerability is discovered and the time at which it is fixed or worked around. By letting manufacturers of implicated software know as soon as possible, the CERT helps initiate the process of closing the window. By doing so, the value of the information to hostile agents diminishes with time. CERT publicly releases limited vulnerability information after approximately 45 days. This implies a judgment that at that time, it is or should be more valuable to legitimate users and system administrators than to hostile agents. The situation is different depending on whether or not hostile agents are aware of that information; if the information is widely known by hostile agents, little harm may be done by publicly releasing it, and it may help legitimate users.

The CERT model does not distinguish between legitimate and hostile agents, excepted for the producer of the software; the information is either publicly released or it is withheld. An alternative model (the "federated model") proposes to share the information as soon as possible between a subset of legitimate, cooperating entities or individuals [Meunier1].

We present the design of a structured system that takes care of these concerns while adding some significant advantages. Within this vulnerability database system, users are able to cooperate by sharing the information they have. This builds a richer and more useful system whose added value resides not only on collecting information from many different sources and having a well-defined cooperative review process, but also on the classification model supporting the database schema. Taxonomies are key to the successful control and storage of vulnerability information. Having a good way of distinguishing and categorizing vulnerabilities makes the system much more structured, consistent, and easy to use and maintain. It also helps eliminate possible ambiguities in the description of vulnerabilities that could be similar. A vulnerability type subsystem is one of the important contributions of this project. According to the multiple attributes of vulnerabilities, we are able to construct classes or types by grouping features that belong to a same vulnerability nature.

The ideas from the workshop were just part of the motivation behind this project. CERIAS (previously known as COAST Lab) developed a Vulnerability Database System [Song1] that allows local researchers to access vulnerability information and to update it at will. This system, however, keeps this information confidential for internal use only. The system uses a classification proposed by Krsul in [Krsul1]. We utilize that classification to build a more robust system using a relational database engine that allows for efficient storage and retrieval of data. This system is called the CERIAS Cooperative Web Vulnerability Database (WebVDB). The design of the system includes a relational database schema that hosts all the information from the original system and supports a voting mechanism where a group of reviewers verifies the information submitted by multiple users. They submit votes indicating the level of confidence they have on the validity of the information. Votes are then combined in order to decide whether the vulnerability can be made public, or if any corrections are necessary.

The remainder of this paper is organized as follows. Section 2 describes the classification model and the structure of the user interface. Section 3 outlines the privilege system, the state transition mechanisms, and the review process. In Section 4 we describe some technical aspects of the system such as the use of cookies, and the integration of PHP/HTML with the MySQL relational database management system. Our concluding remarks are in Section 5.

2 Classification

The system is built on the taxonomy proposed by Krsul [Krsul1]. The classification of vulnerabilities involves assigning correct values to a number of features; this often requires following decision trees. An initial implementation of a vulnerability database using this classification did not easily allow cooperation between remote parties, did not utilize an SQL-compatible relational database, did not include submission and review processes, had a poor user interface, and did not clearly allow several of the one-to-many or many-to-many relationships presented here. We adapted the classification to a normalized database schema and grouped classifiers into functional categories and entities that explicitly provide the needed one-to-many or many-to-many relationships.

2.1 Classification features

We assembled classifiers into related functional groups that provide the foundation for a sensible user interface, using a different page or "tab" for each group (Table 1). The first group is named "Identity". In addition to a title and a short description, vulnerabilities are assigned a unique key. Because vulnerabilities in the database might not yet have been assigned CVE numbers (Common Vulnerabilities and Exposures, http://www.cve.mitre.org), these could not be used as a key. A simple integer was used instead. Nevertheless, CVE numbers may be recorded when available, because they are helpful in identifying vulnerabilities.

The vulnerability is described in depth in the Analysis section. The impact, access required, core vulnerability, detection method, and detailed analysis are provided in this section, as defined by [Krsul1]. A collection of exploit programs is also maintained. An exploit program can be linked to several vulnerabilities (some hacking tools, for instance, can exploit more than one vulnerability on a system). In addition, vulnerabilities may be linked to several exploit programs, thus defining a many-to-many relationship. The level of expertise needed to run the exploit and its complexity are also stored with the exploit. Other groupings are the Nature, Sources, Fixes, Policies, Operating system, Environmental features, Assumptions, Software, and other features [Krsul1].

Group	Features
Identity	Title Description CVE
Analysis	Direct Impact Access Required Analysis Core Vulnerability Detection
Exploits	Exploit programs Description Exploit code Ease of exploit Access required Complexity of exploit
Nature	Object Effect Method Input
Sources	CERT Detail Advisories Information References Related Documents
Fixes	Main Fix Patches Workaround Test
Policies	Policies set
Operating system	Name Vendor Type Variant Version
Environmental features	Environmental features
Other features	Other features
Assumptions	Assumptions set
Software	Name Vendor Version

Table 1. Functional WebVDB groups

The nature of the vulnerability is a one-to-many relationship because many objects may be involved. For example, a stack buffer overflow involves overwriting the stack data, the return address, and the stack is executed. This generates 3 Nature entries to link to a vulnerability. Therefore, the database schema storing this information is an entity-relationship structure composed by multiple relations (Figure 1).

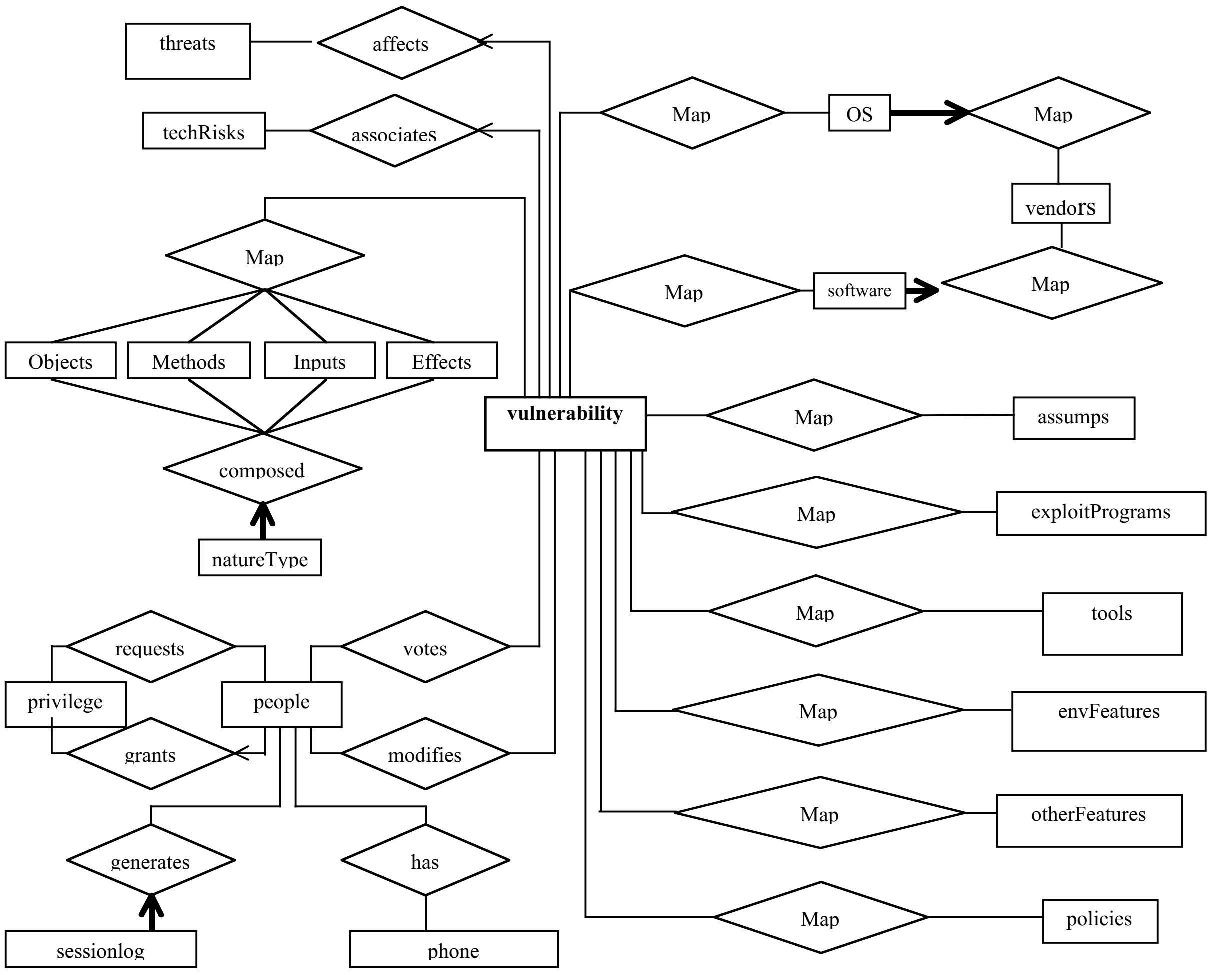


Figure 1. Overview of database schema (Entity/relationship diagram)

In the database there is a total of 46 tables; 29 of them are entities and 17 are relationships. The three major sets of tables are types, people (system users), and features. Types will be discussed below (2.2).

2.2 Types

A vulnerability type is a meaningful combination of values for several related features. It is a known fact that several vulnerabilities may share the same characteristics, represented by the unique combination of feature values. Type is a useful theoretical construct to identify the shared characteristics in the taxonomy. By this construct, it is easy to address a group of vulnerabilities at the same time. In practical terms, a type also helps in data input because data can be directly copied from predefined types. For instance, Figure 2 shows three types. Nature Type describes the nature of the vulnerability in terms of four features: object, method, effect, and input.

Technical Risk Type describes the kind of risk vulnerability implies. Threat Type describes the kind of threat vulnerability has regarding the availability and functionality of the information.

Nature TYPE	Object Method Effect Input
Technical Risk TYPE	Consequence Access Type Privilege Type
Threat TYPE	Observe Destroy Modify Creation Availability Disclose Execution Misrepresent Repudiate Integrity Confidentiality

Figure 2. Sample types

These types group features that identify a certain vulnerability class. For example, because of their nature, vulnerabilities that can affect heap memory through the use of memory copy routines (represented by the "memcpy" keyword in [Krsul1]), can be classified as a buffer overflow.

<u>Example of Nature Type</u>

Name:	Heap Buffer Overflow
Nature object:	heap_data
Nature effect:	replaced
Nature input:	any
Nature method:	memcpy

<u>Example of Technical Risk Type</u>

Name:	User access
Description:	Attacker is able to access a user account other than root.
Priority:	90
Consequence:	Account access
Access Type:	List files, read files, write files, execute files
Privilege Type:	Normal

<u>Example of a Threat Type</u>

Name:	Denial of Service
Observe:	No
Destroy:	No
Modify:	No

Creation:	No
Availability:	Yes
Disclose:	No
Execution:	No
Misrepresent:	Yes
Repudiate:	No
Integrity:	No
Confidentiality:	No

2.3 Structure of the user interface

Because the number of classifiers is large, the user interface needs to be designed to avoid confusion, and must allow the saving of partial entries (increments of work). Simply putting all the features on one web page is a known bad idea. Instead, a method known as "feature grouping" was used (Table 1). Each group represents a more general and abstract concept when compared to individual features. For example, the Exploit group contains all the features related to how the vulnerability can be exploited on the flawed system. The grouping method effectively creates a Hierarchical structure with groups at higher level and features at lower level. The hierarchy is consistent with the categorized nature of the organization of information on the Web. Therefore, both a logical organization of the vulnerability data and a highly usable interface are made possible.

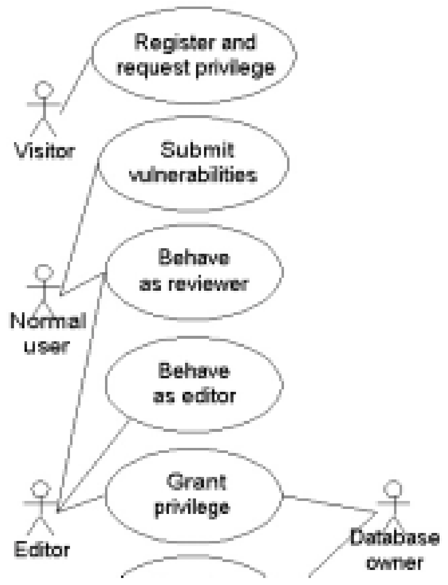
To accommodate the various activities and options for users with different privilege level (see later), a matrix representation is used as the menu interface. Each row of the matrix corresponds to a privilege level (normal and editor), while each column corresponds to a category of activities (people-related or vulnerability-related). It is easy to identify a menu option by following a row and a column up to the cross point.

3 Mechanisms

The working environment supporting cooperative efforts for analyzing and publishing computer vulnerabilities uses the following mechanisms:

- Privilege system and access control
- Vulnerability state transitions

Different people with various levels of knowledge and skill are meant to share information and



work together on vulnerabilities. The privilege system along with the vulnerability state control, ensures proper cooperation among users. An editor-reviewer protocol was adopted to reflect the interactions between privileged users and the transitions of state for vulnerabilities when they are being processed in the system.

Figure 3. Privilege levels

3.1 Privilege system and access control

Because of the collaborative nature of the system, a visitor must register first to be approved as a user. Users are expected to meet a minimum level of trust and knowledge for working on vulnerabilities. The system contains two privilege levels: normal user and editor (Figure 3). The normal user privilege means that the user can submit and edit his submitted vulnerabilities, and review those assigned by editors. Editor privileges comprise normal user privileges plus the privilege of processing submitted vulnerabilities. Editors can also have the option, based on their evaluation, to grant or deny a privilege request from a visitor. The database owner has the highest level of privilege which allows looking at the table contents, modifying their structure, and even deleting information. A use-case diagram in Figure 3 illustrates the privilege levels.

3.2 Vulnerability state transitions

This is an 8-state process from submission to availability. They are:

1. Being Entered. A user is inputting/editing the vulnerability record before submission.
2. Submitted. The submitted vulnerability is waiting to be processed by editors.
3. Checking (Checks). An editor takes (grabs) the vulnerability for initial review.
4. Assigning (Assignment). The editor assigns the vulnerability to users for review.
5. Voting. Indicates the voting mechanism has started.
6. Rejected. The vulnerability has been rejected either by the editor or by the reviewers.
7. Available. The vulnerability has passed the review and becomes available in read-only mode to all users.
8. Modify. The submitter can modify a rejected vulnerability for re-submission.

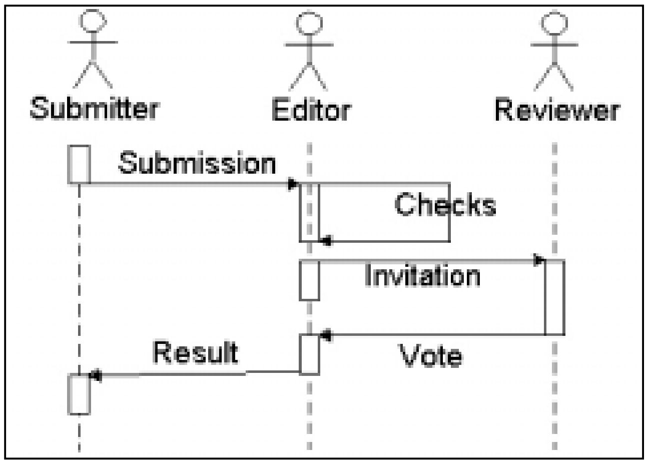


Figure 4. Editor-reviewer protocol

Upon submission of a vulnerability, an editor must take ownership of the review process. The editor will lead and organize the collaborative efforts on a particular vulnerability. An editor-reviewer protocol is implemented to evaluate the vulnerabilities. The result is collected using a voting mechanism.

First, an invitation is send by the editor to reviewers selected for that vulnerability. The selection is based on the personal judgment of the editors. Currently, the number of reviewers for a vulnerability is three. Second, the reviewers, upon receiving the automated email invitation, accept or decline the invitation. If a reviewer refuses to join the collaboration, the editor needs to find a substitute. Third, the reviewers will evaluate the content of the vulnerability based on their knowledge in the field. A vote is expected from each reviewer of the vulnerability. This vote ranges from -3 to 3. The sum of the votes from all reviewers is the final score given by the evaluation mechanism. The comments of the reviewers with regard to their votes will also be recorded. If the vulnerability is judged as acceptable for evaluation, the editor will invite 3 other users, either editors or normal users, to be the reviewers of the vulnerability. Thus the editor-reviewer protocol is invoked, which will give a voting score for the vulnerability.

The voting score determines whether or not the vulnerability is accepted. A minimum score of 3 is necessary for a vulnerability to be accepted. A negative score leads to its rejection. A score between 0 and 2 means that the opinions of the reviewers are either neutral or divided. In this case, the vulnerability will be sent back to its submitter for improvement.

Acceptance or rejection of a vulnerability marks the end of the collaboration process. The submitter of the vulnerability will be notified in both cases. However, if a vulnerability is sent back for modification, the evaluation process will continue while the modifications are done and will end when a decision is finally made on it. Vulnerabilities will be made available when they are accepted.

The above description explains the way the users of the system interact. They move a vulnerability to one state into another to ensure information quality. A submission may be subjected to the following transitions (Fig. 5):

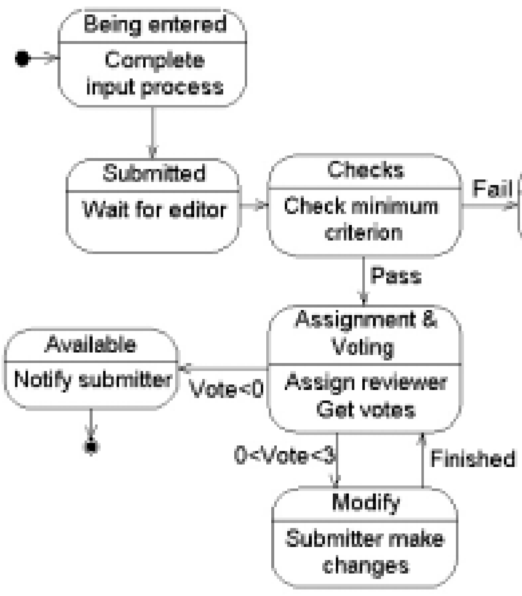


Figure 5. State transitions

Being entered → Submitted. A user submits a vulnerability. Entering data for a submission may take time, so the submitter is allowed to save partial entries for future edition and submission. In this stage, the state of the vulnerability is "Being entered". Once the submitter finishes the input process, the state of the vulnerability changes to "Submitted", and the vulnerability now waits to be processed by an editor.

Submitted → Checks. Once a user has submitted a vulnerability, it becomes listed in the menu for editors. If an editor takes ownership of the review process, the state changes to "Checks". At this stage, the editor studies the vulnerability, then

Checks → Rejected/Assignment. The editor studies the submitted vulnerability against minimum requirements of understandability and completeness. The editor decides whether to reject the vulnerability or to assign it to three users for review. While looking for reviewers, the state of the vulnerability is "Assignment".

Assignment → Voting. Users receive a notifying email from the system inviting them to be reviewers. From the moment any reviewer sends a vote, the state of the vulnerability changes to "Voting" until all three reviewers have voted for the vulnerability.

Voting → Available/Rejected. In the "Voting" stage, the system automatically records the voting results and comments from all the reviewers. Based on the results, the vulnerability can be made available for publication and then the vulnerability's state changes to "Available". The editor, the submitter and the reviewers of this vulnerability then receive a notification email from the system, and the vulnerability is open to all users in read only mode. Otherwise, the system rejects the vulnerability and the vulnerability's state changes to "Rejected". The editor, the submitter and the reviewers of this vulnerability then receive a notification email with the voting results and comments.

Rejected → Modify. A rejected submission may be deleted or modified for re-submission taking into account the comments received from the reviewers. In the later case, the vulnerability's state changes to "Modify". This stage is similar to "Being entered" in the sense that edition is allowed, but it allows knowing whether a vulnerability has been modified and re-submitted.

4 Technical Aspects

4.1 Cookies

MySQL was selected as the database engine because it is free, easy to maintain, and well-supported by web-scripting languages. All the code is written in PHP and HTML, and only secure connections (SSL 3.0 or TLS) are used. We created small modules that can be easily analyzed and the comments throughout the code help understand the functionality and the input/output values of each program. A library of utility functions were put together as include files.

A well-known potential problem of web-based systems is the submission of html code in input fields, that would later subvert the system during a subsequent attempt to display the data to other users. In addition, quotes might be used to affect SQL statements. In order to prevent the introduction of incorrect or dangerous data into the database, all input was run through “sanitization” routines. This guarantees that no improper operations will be performed once the data is stored onto the database, and that the functionality of the system will not be affected by values being read from the tables. For example, special html characters are eliminated or replaced by its corresponding "printable" representation before being sent to the database for storage (e.g., a </TABLE> string entered into an input field would be interpreted as the closing tag of a table if some special characters were not substituted properly).

HTML being stateless, the credentials of users are verified at the beginning of every script. A cookie stores a large number (nonce) that is used once only (used nonces are remembered and marked as invalid upon logging out of the system) and acts as a key into a table that stores all session information. No information is stored on clients, because it could not be trusted. A utility routine helps determine the privilege of a user so that he can only do what she is entitled to. Multiple items are displayed dynamically on the screen depending of this privilege that grants read-only, add-only, or total-access to a vulnerability. Code review sessions were held to verify the quality of the program.

5 Conclusions and applications

As designed, the cooperative vulnerability database can be used within or across companies or groups. It provides unique capabilities for remote cooperation and quality assurance. The framework may be duplicated and installed as needed, with different participants. Remote access to installations of the cooperative vulnerability database would greatly help community efforts such as the CVE.

The database could easily be adapted for other purposes because the source is available and because its capabilities are extended by simply adding a new script. For instance, it could be integrated into an incident response system. Another possibility is using it to control the disclosure of vulnerability information. Typically, users need to be informed of possible flaws present in the software they use and vendors should be contacted whenever a vulnerability is found so that they can produce a patch for the software they produce. Releasing vulnerability information too early would give attackers the chance to study the vulnerability and exploit it before the vendor actually makes the corresponding fix available. The cooperative database could be used by a trusted 3rd party to control the disclosure of the information and prove that vendors were notified in time.

6 Bibliographic references

[Buck1] D. Buck, D. Stucki. "Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development". Proceedings of the thirty-first SIGCSE technical symposium on Computer Science education, 2000, Pages 75 - 79.

[CSI/FBI] "Issues and Trends: 2000 CSI/FBI Computer Crime and Security Survey". Computer Security Institute, San Francisco (http://www.gocsi.com/)

[Krsul1] I. Krul. "Software Vulnerability Analysis", PhD Thesis. Department of Computer Sciences, Purdue University. COAST TR 98-09, 1998.

[Krsul2] I. Krsul, E. Spafford, M. Tripunitara. "Computer Vulnerability Analysis". Department of Computer Sciences, Purdue University. COAST TR 98-07, 1998.

[Landwehr1] C. Landwehr, A. Bull, J. McDermott, W. Choi. "A taxonomy of computer program security flaws". ACM Comput. Surv. 26, 3 (Sep. 1994), Pages 211 - 254.

[Meadows1] C. Meadows. "An outline of a taxonomy of computer security research and development". Proceedings on the 1992-1993 ACM SIGSAC on New security paradigms workshop, 1993, Pages 33 - 35.

[Meunier1] P. Meunier, E. Spafford. "Final Report of the 2nd Workshop on Research with Security Vulnerability Databases". CERIAS, Department of Computer Sciences, Purdue University. TR 99-06, 1999.

[Meunier2] P. Meunier. « The Incident Response Database ». https://www.cerias.purdue.edu/irdb. CERIAS-Purdue University, 2000.

[Oman1] P. Oman, C. Cook. "A taxonomy for programming style". Proceedings of the 1990 ACM annual conference on Cooperation, 1990, Pages244 - 250.

[Song1] G. Song, S. Mandujano, P. Meunier. "CERIAS Classic Vulnerability Database User Manual". CERIAS, Purdue University. CERIAS TR 2000-17, 2000.