A Machine-Oriented Integrated Vulnerability Database for Automated Vulnerability Detection and Processing

Sufatrio – Temasek Laboratories, National University of Singapore
Roland H. C. Yap – School of Computing, National University of Singapore
Liming Zhong – Quantiq International

ABSTRACT

The number of security vulnerabilities discovered in computer systems has increased explosively. Currently, in order to keep track of security alerts, system administrators rely on vulnerability databases such as: CERT Coordination Centre, Securityfocus BugTraq and Sans Vulnerabilities Notes Database. Such databases are designed primarily to be read and understood by humans. Given the speed at which an exploit becomes available once a vulnerability is known, and the frequency of occurrence of such vulnerabilities, manual human intervention is too slow, time-consuming and may not be effective. We propose the design of a new vulnerability database which is oriented to be machine readable and processable rather than human oriented. This allows automated response to a vulnerability alert rather than relying on manual intervention of system administrators. With this approach, many kinds of automatic processing of alerts become feasible. We show the value of such a database by constructing a prototype sample scanner for Unix systems tailored for Linux RedHat and FreeBSD. We envisage that our work can help spur a development of far more effective vulnerability databases to benefit a wide-ranging user community.

Introduction

A worrying trend in the age of the Internet is the increasing incidence of cyber attacks. CERT statistics [1] quotes 114,855 reported incidents (an incident may involve an arbitrary number of sites, even thousands) in the first nine months of 2003 alone. This is a large jump from 21,756 incidents in 2000.

One of the objectives of computer security emergency centers like CERT is to help disseminate vulnerability alerts and relevant advisory notes to the user community in a timely fashion. However, the speed of cyber attacks together with the complexity of administrating computer and network infrastructures today, makes it difficult for many system administrators to cope with such attacks. While automatic tools may be available, there is still a need to routinely inspect any security/vulnerability alerts in order to take the necessary corrective measures.

Current sources of such alerts are designed primarily for human consumption and contain large amounts of information in natural language format. In this paper, we will call such sources, *vulnerability databases*, because they deal with collections of data and not whether they are actually kept in a database form or not. While a human oriented format is useful for disseminating the full details of an alert, it also requires a human in the chain to make use of it. This problem is acknowledged in a CERT document [2]. Given the 5500 vulnerabilities reported in 2002, it is estimated that a system administrator would need 229 days just to digest the information. Furthermore, usually multiple vulnerability databases need to be consulted to fully deal with a vulnerability, i.e., just the CERT entry is not sufficient. Thus, the deck is stacked on the side of the hackers rather than the system administrators.

Clearly, the solution would be to move away from direct human processing towards automatic security alert response processing. This paper proposes an initiative to redesign vulnerability databases to be machine oriented and amenable to automatic processing. In practice, such a database would also need to integrate vulnerabilities disclosed from multiple sources. The dissemination of machine processable alerts allows for automated tools to operate on an alert immediately without requiring humans in the loop. This would cut down the long time interval between release of a vulnerability/advisory note and corrective action being taken. Other automated tools do exist, e.g., Microsoft Windows systems have Software Update Services, however there is little which is general purpose, publicly accessible, and open to public or third party scrutiny and verification. We have developed a proof-of-concept machine oriented database schema using a vulnerability expression language for describing the targets and effects of vulnerabilities. To illustrate the use of this database, we have developed a prototype vulnerability scanning robot which can determine existing and potential vulnerabilities based on the database.

The creation of an effective machine oriented vulnerability database would require the cooperation of many parties such as CERT, BugTraq, vendors, software developers, etc. As such, this paper is not meant to be a standalone definitive solution. Rather the prototype database and scanner is intended to spur the development of machine oriented databases by the parties concerned. We believe that our proof of concept presents the key elements for further development of machine oriented vulnerability databases. The use of a simple vulnerability expression abstraction also simplifies the integration of data from multiple sources.

Motivation and Design Goals

Figure 1 reproduced from the following CERT report [3] describes the vulnerability exploit cycle. The Y-axis represents the number of incidents for a given vulnerability.

The graph illustrates the time lag between the release of a vulnerability/advisory report and the decrease of incidents following corrective measures by users. We argue that current vulnerability databases, such as CERT, Bugtraq, CVE, in their present format are not designed to facilitate a speedy user response because they suffer from the following limitations:

1. Much of the information in these databases, particularly the portions which relate to dealing with a vulnerability, is only in a human readable free-text format. While this may be necessary to convey the full information content, it also means that a human needs to interpret the database entry. This makes it difficult to have any form of automated machine processing of this information. While it is possible to analyse the natural language text, this may introduce more problems due to ambiguities in natural language.
2. Different response centers use different terminologies and conventions in describing one vulnerability, which may confuse the users of the information. For instance, some databases put the affected systems according to vendor's version (e.g., RedHat Linux 8.0). A different vulnerability might refer to the Linux kernel version instead.
3. There is a conflicting and fluctuating standard among response centers which actively promote their own methods and standards, thus causing frequent shifting and switching among different proposed standards.

Our philosophy is that vulnerabilities should be expressible in an explicit form in terms of data (or a description) rather than an implicit form like code to process a vulnerability. Hence the data can be stored in a database (or any data description language, i.e., XML). Our database is designed with the following goals:

- The database is designed so that it can be consolidated from multiple sources, in which each vulnerability entry includes the origin of information, environment of which it can cause an impact, its consequence, as well as additional useful information.
- The pre-requisites and the consequence of a vulnerability are described using an abstraction which we call a vulnerability expression. The vulnerability expression allows a precise formulation of the nature of the vulnerability and is machine processable. The vulnerability expressions are not specific to a particular system but rather tailorable to the specific system using another mapping, e.g., a configuration file may be mapped separately to its pathname for the particular system being tested.
- The structure of the database should allow easy retrieval both by user and automated-tools via SQL.

We also want to have an automatic scanner which can use the database to do the following:

- Check whether a given vulnerability exists on a local system;

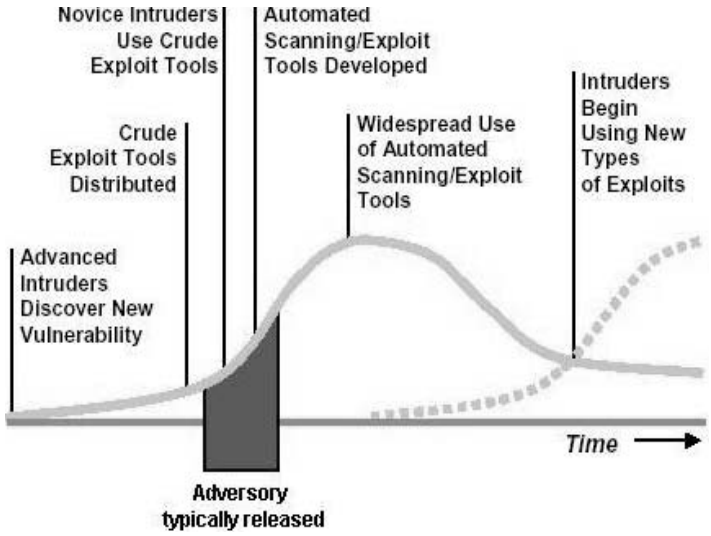

Figure 1: Vulnerability exploit cycle (CERT Coordination Center).

- Scan local system for all possible vulnerabilities;
- Notify the existence of potential vulnerability on the local system should certain environmental factors such as system services which are currently off get activated;
- Analyze relationship between different vulnerabilities, e.g., whether one vulnerability can be exploited to lead to another.

Related Work

There have been a number of popular tools that scan for any presence of vulnerability or configuration weaknesses in a system. Some notable examples are: COPS [4], SATAN [5] and Nessus [6]. These tools are code-based scanning applications where the logic of vulnerability checking is embedded tightly in the scanner's code. This means that including a new check for a vulnerability entry requires one to update the scanner's code, its sub-component(s), or its configuration file. In contrast, our system uses a generic scanner which makes use of vulnerability descriptions stored separately in a vulnerability database. While a code-based solution is generally more powerful, it requires that code/plug-ins be written. There are trust and verification issues which we discuss later in this section.

There is some existing work which reorganizes and integrates information in existing vulnerability databases into one that is more of a ''real database.'' NIST has developed ICAT [7], a searchable index of vulnerability entries leading the users to various vulnerability resources and patch information. Similarly, Purdue University maintains a web-based search system called ''Public Cooperative Vulnerability Database'' [8]. These databases are, however, designed mainly for vulnerability search based on categorized attribute values, and not for automated applications.

Krsul [9] proposes a comprehensive taxonomy of vulnerabilities for possible further processing or automated manipulation. A database is also proposed. It is hard to compare the database since no specific applications were co-designed with it.

Windows Update [10] is a Microsoft online tool for automatically updating Windows operating systems and Microsoft applications with recent patches. It illustrates some important issues with automatic tools. Windows Update (and its more automatic cousin, Windows Software Update Services [11]) are closed systems. We propose an open system which can cater for heterogeneous environments. Windows Update has a ''black-box update model'' which allows easy and seemingly automatic patch update, yet the non-transparency of the system leads to the following issues:

1. **Privacy and Trust issues**: As there is no open specification or possible inspection on the scanner, no complete trust can be put on the scanner. This is the case with any code based system. It is difficult to determine if the scanner is performing the correct actions while preserving local system security policy. Since the update system is hosted on the vendor's server, there is no guarantee that local information will not leak to an external source thus breaching local system privacy. In contrast, our system is open to inspection. The database can also be deployed locally or organization-wide as discussed in the deployment section.
2. **Non-standard vulnerability checking**: Windows Update behaves more as a vendor patch update mechanism rather than standardized vulnerability entry checking. Thus, little coverage is given back to users in terms of standard vulnerability report information. This might be too limiting for system administrator who, for example, wants to ensure that his systems are up-to-date against recent vulnerability reports regardless whether patches for the vulnerability are available or not.
3. **No control over the scanner**: Users need to trust that Windows Update works as it is supposed to. The importance of this issue is highlighted by a recent incident of Swen-style Trojan horse which posed as a legitimate update [12]. While this example is a social-engineering style attack, it illustrates the fact that the Windows Update mechanism can itself be a vulnerability. In our system, as the database contains a machine readable description, all steps of the scanner can be verified.

Some related concerns of the Windows Update mechanism is discussed in an article by Berlind [13]. We argue that any automatic update or alert processing mechanism should be based on an open model which can be independently verified. In addition, it should be possible for the user to bypass the automatic system in cases where the security policy may not allow the execution of foreign code or connection to external hosts. Morover, the administrator/user should be able to determine the consequences of a patch or alert on his system.

Movtraq: A New Vulnerability Database

The integrated vulnerability database which we have called *Movtraq (Machine Oriented Vulnerability and Tracking) database* is designed to be compiled from multiple source vulnerability databases and is usable directly by an automatic scanner (see Figure 2).

Design Considerations

The main challenge in designing the new database is to determine what the actual contents of each vulnerability entry should be. For our proof-of-concept, we have focused on what the database should contain rather than on a general database schema. The data fields corresponding to a vulnerability fall into three general categories: general information and references; vulnerability factors and its environmental requirements; and impact of vulnerability.

General Information Fields

The general information portion mostly contains references to several public vulnerability databases such as CERT, Bugtraq, etc. The purpose of these fields is to give the user a reference to the original source of information to obtain additional information. This is mainly for human consumption.

Vulnerability Factors and Environmental Requirements

The second category, vulnerability factors and environment data, provides the main content of machine processable vulnerability information. A vulnerability has to exist within a context, hence it is described in terms of its original source factor and associated environmental factors. By ''original source factor,'' we mean the system component(s) (application or operating system) where the vulnerability originates. ''Environmental factors'' refers to settings/configuration or services in the local system which make the system subject to the vulnerability.

We distinguish between two kinds of vulnerabilities:

- vulnerability which currently exists on the system; and
- vulnerability which *potentially* exists on the system.

There are a number of different combinations of original source and environment factors:

Case 1: Vulnerability factors: match & Environment factors: match

We will get this result when a particular vulnerability's original source exists on the local system and the settings of local system match all the environment factors. In this case, we will conclude that the vulnerability exists on the system.

Case 2: Vulnerability factors: match & Environment factors: no match

This occurs when we can detect the origin of the vulnerability on the local system, however the settings of the local system does not match the environment factors. So the vulnerability is not applicable but it has the potential to affect the system if the environment changes. For example, consider the case of ''Apache Web Server Chunk Handling Vulnerability'' [14]. Even if apache is installed, we will not be affected by the vulnerability as long as we do not provide http services.

Although this second case appears to be an exception, it is actually not uncommon as a full installation of the operating system and application programs may have been done. Hence, many installed components in the system may not usually be in use.

Case 3: Vulnerability factors: no match & Environment factors: match

In this case, the vulnerability would appear to be not applicable. However, there is a subtle issue. Consider the case of OpenSSL (an open source implementation of the SSL protocol) which had several stack overflow vulnerabilities which are exploitable [15]. OpenSSL may not be installed as an individual component, so even if there is a database entry for the OpenSSL vulnerability, this would return a negative result in terms of vulnerability data factors. However, OpenSSL is commonly included in applications such as Apache, Sendmail, Bind, Linux and Unix based systems. Thus, it is necessary to check for the existence of such applications which may indicate that such an OpenSSL vulnerability exists even if OpenSSL is itself not detected. This highlights that one may need several database entries corresponding to a vulnerability given some of these indirect potential factors.

Case 4: Vulnerability factors: no match & Environment factors: no match

The vulnerability does not exist on the local system.

Vulnerability Impact (Consequences)

The third category of data concerns the impact of vulnerability, which describes the possible consequences of a vulnerability if it is successfully exploited. In our database, this is stored as a vulnerability description expression which is machine processable and describes the vulnerability impact in a precise and concise form. There is no need to use any taxonomy or qualitative impact factor (e.g., critical, high, medium,

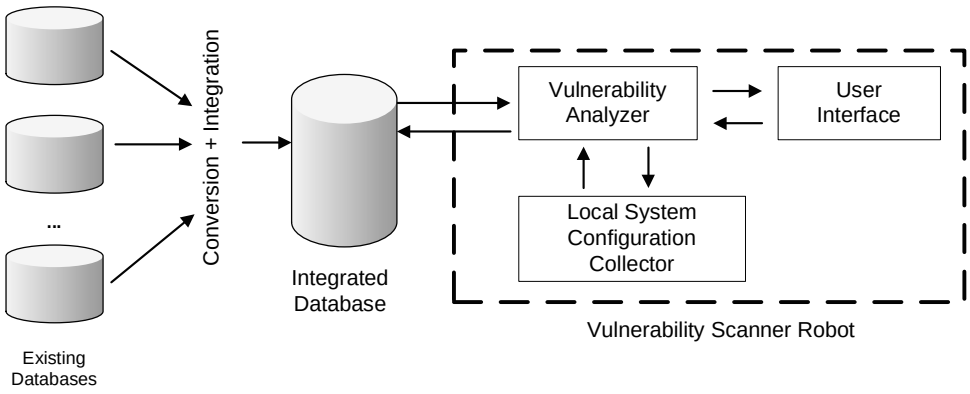

Figure 2: Vulnerability database and scanner.

low) which is not precise and may not make sense in the context of a particular system. It also enables checking of the relationship between different vulnerabilities and whether they can affect one another.

Database Structure

As we have argued, the exact structure of the database is not so important. Rather, it is the content and having it in a more precise machine processable format. In our proof-of-concept design, the database has seven main entities namely:

- Vulnerability Entity – names the vulnerability and links it to the specification of the vulnerability and environment.
- Vulnerability Specifications Entity – collects the vulnerability factors and the impact.
- Environment Specifications Entity – collects the environmental factors.
- Operating System Entity – vulnerability requirements originating from the operating system.
- Application Entity – vulnerability requirements specific to an application.
- Services Entity – vulnerability requirements specific to a service.
- Exploit Entity – details of exploits and impact.

An entity relationship diagram which gives an overview of the relationship between these data items is given in Figure 3.

We will briefly mention some of the key fields from an integration and machine processable perspective. We have mainly omitted fields in the general information category which are present in the database for human consumption.

- **Vulnerability Entity** – a textual description for the vulnerability, identifiers such as CERT ID, BugTraq ID, CVE ID and also other keys corresponding to other tables.
- **Vulnerability Specifications Entity** – vulnerability consequences*, hardware requirements, name of vulnerable application/service*. A service could be a daemon.
- **Environment Specifications Entity** – existence of required user/application or service object* which may be exploitable, existence of a file object*, remote exploitation flag, application/services environment, hardware requirements.
- **Application/Services Entity** – name of application/service, application/service ID, vulnerable versions, hardware requirements. Services have additional fields like protocols, port numbers, etc.
- **Operating System Entity** – similar to application entity but for the operating system.
- **Exploit Entity** – actual exploit (could be a URL, filename, etc.), privileges needed*, consequences of using the exploit*.

The fields which have been labeled by (*) make use of the vulnerability description expressions or vulnerability target objects from the next section. Note that some fields which have a similar function occur a few times in a different context, e.g., hardware requirements may be different for the application and environment, there are two different consequences – one from the vulnerability and one from using a specific exploit.

Integrating the Data

One of the difficulties with dealing with security/vulnerability alerts is the need to integrate the information from multiple sources. Our prototype database

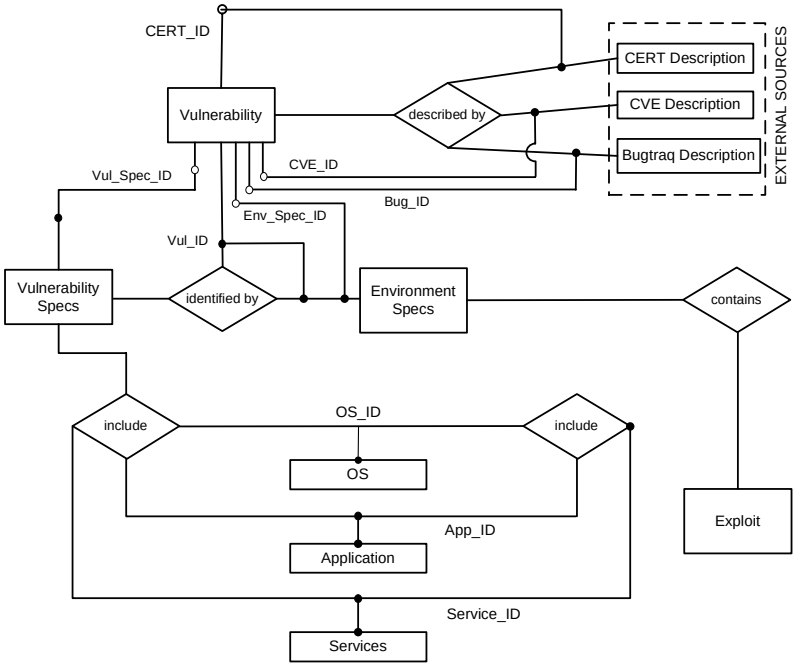

Figure 3: Vulnerability database structure.

is no exception and was built by integrating data from multiple vulnerability sources such as CERT, BugTraq, CVE, vendors and software developer sites. Ideally, one would prefer a single source for the vulnerability information (even if it is only in text form). However, the reality is that due to the distributed handling and speed of dealing with vulnerabilities, one has to accept that integration may be required.

The following example, which is the ''OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability,'' illustrates the need for integration. It has a CVE ID of CAN-2002-0656 [15].

BugTraq from SecurityFocus provides:
BugTraq ID: 5363
Application environment: Apache v1.0 - 1.3.26
OS environment: Linux, Microsoft Windows
Proof of concept exploit: available
Minimum user rights for exploit: u#R[1]

CERT vulnerability advisory provides:
CERT ID: CA-2002-23
Vulnerable application version:
OpenSSL prior to 0.9.6
Vulnerability impact: @G u#S

Vendor/software information:
From OpenSSL (www.openssl.org) we get the vulnerable application range as: 0.9.1c - 0.9.5a.
From apache documentation we know that usually the user is root.

In general, determining the complete environmental requirements and the consequences of the vulnerability from the textual descriptions can be a tedious and time consuming process. This is one rationale for a better system such as the one described here.

Vulnerability Description Expressions

The main machine oriented data fields in the database belong to three categories: system components of the vulnerability; environment factors of the vulnerability; and consequences of the vulnerability.

The first category for various system components is usually specified as versions of the operating systems and applications. This can be straightforwardly encoded in the database. The other two categories require a machine friendly specification.

After studying 943 vulnerability notes from CERT advisory database, we found that most of the information for these two categories can be described effectively using the *vulnerability description expression* described below. These expressions are inspired by the rule language in KuangPlus system [16].

An expression is written with the syntax:
⟨*VulnerabilityExpression*⟩ = ⟨*TargetObject*⟩ | ⟨*Action*⟩⟨*TargetObject*⟩

[1]This is a vulnerability description expression to describe the impact, see the next section.

An action is written prefixed by '@'. Table 1 illustrates the actions and the types of the corresponding target objects.

Syntax	Semantics
@G ⟨*u*|*g*⟩	Gain ⟨*user object u* | *group object g* ⟩
@R|W ⟨*f*|*m*⟩	Read|Write ⟨*file object f* | *memory object m*⟩
@A ⟨*f*|*m*⟩	Access (read and write) ⟨*file object f* | *memory object m*⟩
@C ⟨*f*⟩	Create ⟨*file object f*⟩
@K ⟨*f*|*m*⟩	Corrupt ⟨*file object f* | *memory object m*⟩
@X ⟨*f*|*c*⟩	Execute ⟨*file object f* | *code object c*⟩
@S|I ⟨*n*|*a*⟩	Crash | Disrupt ⟨*node object n* | *application object a*⟩
@D ⟨*n*|*a*|*s*⟩	Deny ⟨*node object n* | *application object a* | *service object s*⟩
@U ⟨*r*⟩	Use ⟨*resource object r*⟩
@E ⟨*r*⟩	Exhaust ⟨*resource object r*⟩

Table 1: Actions in vulnerability description expressions.

Rather than giving a formal definition of target objects, we have listed examples of target objects in Table 6. In Table 6 the following prefixes are used: '%' is used to denote an actual value; '#' is used to denote a symbolic value; and '&' is used for expressing users/groups associated with an application/service.

As our proof-of-concept implementation is for Unix systems, the examples and objects are also Unix based. Vulnerabilities for other operating systems may require extension to the types of target objects and actions.

Examples using Vulnerability Expressions

The following examples use the expressions to describe various vulnerability consequences.[2]

- @D n#N: Denial of Service for the whole network (Ref: Cisco IOS Interface Blocked by IPv4 Packet CERT ID VU#411332).
- @G u#S: Gain superuser rights. (Ref: Linux Kernel Privileged Process Hijacking Vulnerability, Bugtraq ID: 7112).
- @G u#R : Gain Remote user right (Ref: Apache htpasswd Password Entropy Weakness, Bugtraq ID: 8707).
- @S a#mailman; @D a#mailman: Crash mailman application, and deny its service. (Ref: Red Hat Linux GNU Mailman Remote Denial Of Service Vulnerability, Bugtraq ID: 10147).
- @R f%/etc/passwd : Read file /etc/passwd.
- @X f#*(4777) : Execute a file with setuid permission.

The following are examples of portions of the machine oriented fields in the database for several vulnerabilities:

[2]For simplicity, multiple expressions are separated by semicolon.

Syntax	Semantics
⟨u⟩:	**User Objects**
u#R	Remote user
u#L	Local user
u#S	Super user
u#*	All users
u#P	Physical user
u%100	User with UID 100
u%nobody	User 'nobody' on the system
u#U	User whose privilege is beyond that of the current user
u&App	User running corresponding application process
u&Svc	User running corresponding service (i.e., daemon)
u&Kernel	User who can access or control the OS kernel
⟨g⟩:	**Group Objects**
g#*	All groups
g&⟨App|Svc⟩	Group of the corresponding application process/service
g%50	Group with GID 50
g%sys	Group 'sys' on the system
⟨f⟩:	**File Objects**
f#*	All files
f#passwd	Pathname corresponding to the passwd file
f#shell	Pathname corresponding to shell files, e.g., ''/bin/bash''
f#system	Pathname corresponding to system files in the OS
f#*(4777)	All files with permission 4777
f#F	Files beyond current user access rights
f%/etc/passwd	The file ''/etc/passwd''
f&App	File associated to the running application process
⟨m⟩:	**Memory Object**
m#M	Memory area beyond the current user's access right
⟨n⟩:	**Node Objects**
n#S	Scanned node where an application program is installed and/or related service is running
n#L	Nodes in local area network
n#N	Network
n%IP	Node at IP address (may be a range)
⟨a|s⟩:	**Application/Service Objects**
a#AppName	Application Name
s#SvcName	Service Name
⟨r⟩:	**Resource Objects**
r#M	Memory
r#CPU	CPU
r#B	Network Bandwidth
r#D	Disk Space
⟨c⟩:	**Code Object**
c#(⟨u⟩)	Piece of code with the execution privilege of the user object *u*, e.g., privilege escalation

Table 2: Objects in vulnerability description expressions.

- *MySQL Password Handler Buffer Overflow Vulnerability*:
  CVE_ID: CAN-2003-0780
  Bugtraq_ID: 8590
  Vul_Con: @X C#(u%mysql); @G u%mysql; @G u#L
  Vul_OS: null
  Vul_App: Various Mysql versions
  Env_User: u#L
  Env_File: null
  Env_Rem: No
  Exploit: No
  Env_OS: null
  Env_App: Mysql
- *Linux Kernel IOPERM System Call IO Port Access Vulnerability*:
  CVE_ID: CAN-2003-0246
  Bugtraq_ID: 7600
  Vul_Con: @A f#F
  Vul_OS: Various Linux distributions
  Vul_App: null
  Env_User: u#L
  Env_File: null
  Env_Rem: No
  Exploit: No
  Env_OS: Linux kernel 2.4.0 - 2.4.21, 2.5.0 - 2.5.69
  Env_App: null
- *Linux 2.4 Kernel execve Race Condition Vulnerability*:
  CVE_ID: CAN-2003-0462
  Bugtraq_ID: 8042
  Vul_Con: @A f#F; @X c#(u#S); @G u#S
  Vul_OS: Various Linux distributions
  Vul_App: null
  Env_User: u#L
  Env_File: f#*(4111)
  Env_Rem: No
  Exploit: Yes
  Env_OS: Linux Kernel 2.4.0 - 2.4.21
  Env_App: null
- *Multiple Vulnerabilities In OpenSSL*:
  CVE_ID: CAN-2002-0656
  Bugtraq_ID: 5363

Vul_Con: @X c#(u&App); @G u#L
Vul_OS: null
Vul_App: Various Apache versions and OpenSSL-based applications
Env_User: u#R
Env_File: null
Env_Rem: Yes
Exploit: Yes
Env_OS: null
Env_App: Corresponding service provided by the vulnerable application

Translation Issues

From our experiments in translating text-based vulnerabilities into vulnerability expressions, we encountered the following issues:

- The vulnerability description in the database sources is sometimes rather vague. Some examples are: ''could expose sensitive information to local attackers'' (Bugtraq ID 8233), ''gain access to sensitive information'' (Bugtraq ID 9558), or ''leads to unauthorized access to attacker-specified resources'' (Bugtraq ID 9778). We require a more specific consequence which either means describing it in a catch-all fashion or much more work is required to understand the vulnerability.
- Our vulnerability expression language is designed to capture general expressions at the OS level. It does not express various application specific descriptions, such as: ''to access variables outside the Safe compartment'' (Perl, Bugtraq ID 6111), or ''could compromise the private keys of ElGamal signing key implementation'' (GnuPG, Bugtraq ID 9115). To deal with such consequences, these are approximated by translation into the closest vulnerability expressions capturable by our language. In the two examples above, we can rewrite them into: access of memory and files beyond the current user's right, respectively.
- Some vulnerability entries, particularly those of CAN(didate) type, are listed as ''unknown consequence'' (e.g., Bugtraq ID 10428). Hence, we either have to ignore such entries for the moment, or use a special form to indicate unknown consequences.

Movtraq Scanning Robot

To demonstrate the use of the Movtraq database, we have implemented a prototype automatic vulnerability scanner (called the *Movtraq scanning robot)*. The robot runs on two different versions of Unix: Redhat Linux and FreeBSD. This is to demonstrate a degree of platform independence.

The overall structure of the robot together with the database is depicted in Figure 2. The integrated Movtraq database is stored in MySql. The scanner consists of a local system configuration collector which collects information about applications, operating system (which processes are running, which ports are open, hardware details, etc.) and services on the system. Software versions are obtained by using the `rpm` utility on Redhat and `pkg_info` utility on FreeBSD. The scanner is written in Perl and queries the MySql Movtraq database using SQL.

The robot has three basic scanning options:

- Vulnerability checking: checks if the system is vulnerable to the vulnerabilities specified in the database (a Case 1 vulnerability).

```
1. Apache Mod_Auth_Any Remote Command
   Execution Vulnerability
Application version check: positive
Service port check: negative
Conclusion: source application is detected,
default port required is not open,
potential vulnerability exists but does not
affect current system configuration

2. Sun One/iPlanet Web Server Vulnerability
   to DOS
Application version check: n/a
Conclusion: source application not detected,
safe from vulnerability

3. Linux Kernel IOPERM System Call
   IO Port Access Vulnerability
OS version check: positive
OS environment check: positive
Conclusion: vulnerability detected!

4. MySQL Password Handler Buffer Overflow
   Vulnerability
Application version check: positive
OS environment check: skipped
Conclusion: vulnerability detected!
```

Listing 1: Sample scanner log.

- Potential vulnerability checking: checks for software vulnerability which exists but the system is not currently vulnerable due to environmental reasons (a Case 2 vulnerability). This can be useful since it may be the case that the system can become vulnerable later, e.g., if a service which was off is turned on.
- Vulnerability with exploit checking: enhances vulnerability checking to see if the listed exploits are directly applicable – this adds the constraints of the exploits into the checking process.

An abbreviated sample log from running the scanner illustrates how application, version and environmental checking is performed; see Listing 1. Only some of the pertinent checks from the log are shown to illustrate the following points:

- **Example 1**: apache vulnerability exists but environment check fails since the required port is not open.
- **Example 2**: no vulnerability since application is not installed.
- **Example 3**: an OS vulnerability so only OS checking is used.
- **Example 4**: vulnerability inherent to MySQL version, OS environment checking is skipped as it is not required.

Vulnerability Chaining Analysis

An interesting use of the scanner is that it can be used to test if existing vulnerabilities can be combined together (chaining) to create more vulnerabilities. This mimics what a hacker might do to take advantage of indirect weaknesses on the system.

Consider the following example which is typical of a privilege escalation attack. Suppose the system has the following two vulnerabilities:

```
Name:     Buffer Management Vulnerability
          in OpenSSH
Vul_ID:   57
CVE_ID:   CAN-2003-0693  Bugtraq_ID: 8628
Vul_Con:  @G u#L
Vul_OS:   null           Vul_App: Openssh apps
Env_Usr:  u#R            Env_File: null
Env_Rem:  Yes            Exploit: No
Env_OS:   null
Env_App:  Service provided by the vulnerable app

Name:     Linux 2.4 Kernel execve Race
          Condition Vulnerability
Vul_ID:   48
CVE_ID:   CAN-2003-0462  Bugtraq_ID: 8042
Vul_Con:  @G u#S
Vul_OS:   Linux          Vul_App: null
Env_Usr:  u#L            Env_File: f#*(4111)
Env_Rem:  No             Exploit: Yes
Env_OS:   Linux kernel 2.4.0 - 2.4.21
Env_App:  null
```

In this example, the scanner discovers that both vulnerability 48 and 57 are present. From Vul_ID: 57 a remote user (u#R) can gain local rights (@G u#L), and this chains onto Vul_ID: 48 which has a local environment requirement (local user: u#L and setuid executable file:f#*(4111)). Thus it discovers that a remote user may be able to exploit the two vulnerabilities to gain local root access.

Chaining analysis illustrates the benefit of a machine oriented approach and the use of vulnerability expressions to analyse relationships between vulnerabilities.

Operating System and Local Configuration Mapping

Because environmental and application vulnerability data are expressed as vulnerability expressions, these abstractions may need to be further refined. In the context of a particular local system configuration, operating system distribution, etc., additional localization may be needed to map the abstractions to concrete objects. One may choose to have additional databases to do this mapping from vulnerability target objects to the actual objects on the system. Our robot prototype does not do this since it has been tested only on RedHat and FreeBSD.

Deployment Strategies for vtraq

The prototype Movtraq system is sufficiently useful to be deployed in a number of ways. Some of the potential scenarios depicted in Figure 4 are:

- **Scenario 1**: Local vulnerability database, local client. Here, each local machine hosts its own database. The Movtraq database is meant to have been downloaded (securely) from another server. This has the advantage that the database is local and thus all operations can be done locally. The disadvantage is that an up-to-date database has to be maintained from every host.
- **Scenario 2**: Organization-wide database, local client. This simply extends scenario 1 to an organizational context where there is an organization-wide database server. Where multiple machines have exactly the same configuration, one may choose to only check on a subset of the machines.
- **Scenario 3**: Internet-based database, local client. Lastly, like in automatic update systems, a database server somewhere on the internet serves as the database repository.

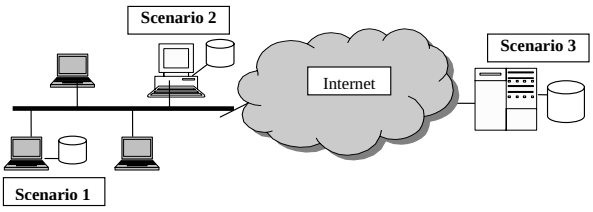

Figure 4: Deployment Options for movtraq.

These strategies are suitable for our Movtraq proof-of-concept system but one could have more

general systems. For example, one could have a scanner which is partially local and partially remote. This may be useful in an organizational context where any system configuration changes are registered with a separate non-local configuration database. Any security alerts are then checked externally against this configuration database.

Discussion

We believe that there is a real need for vulnerability databases which integrate the necessary pieces of information for evaluating the impact of any new vulnerability and allows the appropriate action to be taken automatically. Furthermore, in order to be timely, we argue that the vulnerability evaluation process should not be dependent on having humans process alerts. This does not mean that we advocate not having humans at all in the loop but rather that the loop should not be dependent on the speed of a human response. Thus, it is important that there be a not only human readable vulnerability database but also one which is geared for automatic processing by machines. As far as we are aware, the existing systems for disseminating alerts are still primarily human oriented as are the key source databases.

We have demonstrated a proof-of-concept database which allows effective integration of data from multiple sources and can be used directly by an automatic vulnerability scanner. In the workshop report on security vulnerability databases [17], it was remarked that some of the difficult issues are to do with terminology and the schema of the database. Our database design uses both abstraction and separation of exploits from vulnerabilities – both of which are highlighted in the report. In particular, the use of abstraction, which for us is how the database caters for automated analysis and machine processing, simplifies the issue of terminology and taxonomy. This is a plus point since these are often controversial from a textual description viewpoint.

The database described here is meant to be a proof-of-concept system and is not necessarily comprehensive. However, the prototype scanner demonstrates that we capture the essential elements of a machine-oriented database. As this prototype was designed for Unix systems, for other operating systems, such as Microsoft Windows, both the database and vulnerability expressions may need to be enhanced. However, the fundamental concepts in the design should still be applicable.

Finally, our proposal also addresses a number of important practical issues:

- **Integration of Vulnerability Information**: An integrated database is ideal but may not be practical given that many separate parties are involved in putting together the requisite information. However, it is fairly simple as an additional step to put out the information in the kind of machine oriented form we have advocated and also to concentrate on the relevant data from a machine perspective. In our prototype, we have only built a small integrated database since it is rather time consuming to do so manually from scratch using the existing data sources. However, once vulnerability information is disseminated in the right format, integration becomes significantly easier.
- **Verifiable Vulnerability Processing**: It is certainly the case that any automatic update or scanning system would be welcome by system administrators. However, unless one can deal with the privacy and trust issues, there are significant downsides to the use of such systems. Again, an integrated machine oriented database such as Movtraq allows decoupling of the information from the processing and as it is simply a database, it can be subject to verification.

Further work would involve convenient GUIs, fully featured implementation, Windows compatibility, and a more sophisticated vulnerability model.

Acknowledgments

We acknowledge the support of the ‘‘Defence Science and Technology Agency’’ and ‘‘Temasek Laboratories".

Author Information

Sufatrio holds a B.Sc. from University of Indonesia and a MSc from National University of Singapore. He is currently a Ph.D. student in the School of Computing and an associate scientist in Temasek Laboratories, National University of Singapore. His interests include intrusion detection systems and infrastructure for secure program execution. He can be reached electronically at tslsufat@nus.edu.sg .

Roland H. C. Yap obtained his Ph.D. from the Monash University. He is currently an associate professor in the School of Computing, National University of Singapore. His interests include systems security, operating systems, programming languages and distributed systems. He can be reached electronically at ryap@comp.nus.edu.sg .

Liming Zhong graduated from National University of Singapore in 2004. Currently he is working as an IT security specialist in Quantiq International Singapore. His interests cover intrusion detection systems, network and system forensic analysis. Reach him electronically at rick@Quantiqint.com .

Bibliography

[1] CERT Coordination Center, *CERT/CC Statistics 1988-2003*, http://www.cert.org/stats/cert_stats.html , 2003.

[2] CERT Coordination Center, *CERT/CC Overview Incident and Vulnerability Trends*, http://www.

cert.org/present/cert-overview-trends/module-2 . pdf , 2003.

[3] Lipson, H. F., *Tracking and Tracing Cyber-Attacks: Technical Challenges and Global Policy Issues*, CERT Coordination Center, available at http://www.cert.org/archive/pdf/02sr009.pdf , 2002.

[4] Farmer, D. and E. H. Spafford, ''The COPS Security Checker System,'' *Summer USENIX Conference*, 1990.

[5] http://www.fish.com/satan .

[6] http://www.nessus.org .

[7] http://icat.nist.gov/icat.cfm .

[8] https://cirdb.cerias.purdue.edu/coopvdb/public .

[9] Krsul, I., *Software Vulnerability Analysis*, Ph.D. Thesis, Purdue University, COAST technical report 98-09, 1998.

[10] http://windowsupdate.microsoft.com .

[11] http://www.microsoft.com/windowsserversystem/sus/default.mspx .

[12] Keizer, G. ''Trojan Horse Poses as Windows XP Update,'' *TechWeb News*, http://www.informationweek.com/story/show-Article.jhtml?articleID=17300290 , 2004.

[13] Berlind, D., ''Why Windows Update Desperately Needs an Update,'' *ZDNet Technical Update*, http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2914519,00.html , 2003.

[14] http://www.cert.org/advisories/CA-2002-17.html .

[15] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0656i .

[16] Howard, J., *Kuangplus: A General Computer Vulnerability Checker*, M.IS. Thesis, Australian Defence Force Academy, 1999.

[17] Meunier P. C. and E. H. Spafford, *Final Report of the Second Workshop on Research with Security Vulnerability Databases*, CERIAS TR 99/06, 1999.