Common Vulnerability Markup Language

Haitao Tian, Liusheng Huang, Zhi Zhou, and Hui Zhang

Department of Computer Science, University of Science and Technology of China,
Hefei, Anhui, China
tht@mail.ustc.edu.cn

Abstract. Discovering, disclosing, and patching vulnerabilities in computer systems play a key role in the security area, but now vulnerability information from different sources is usually ambiguous text-based description that can't be efficiently shared and used in automated process. After explaining a model of vulnerability life cycle, this paper presents an XML-based common vulnerability markup language (CVML) describing vulnerabilities in a more structural way. Besides regular information contained in most of current vulnerability databases, information about classification, evaluation, checking existence and attack generation is also given in CVML. So it supports automated vulnerability assessment and remedy. A prototype of automated vulnerability management architecture based on CVML has been implemented. More manageable vulnerability databases will be built; promulgating and sharing of vulnerability knowledge will be easier; comparison and fusion of vulnerability information from different sources will be more efficient; moreover automated scanning and patching of vulnerabilities will lead to self-managing systems.

1 Introduction

With the fast development of Electronic Commerce and Electronic Government worldwide, security of computer systems is attracting more and more attention. As the current state of the art in security is "penetrate and patch", vulnerabilities that can lead to violating security mechanisms of systems are the focus in the area of security. Attackers (or Hackers) seek for and take advantage of existing vulnerabilities on target system to subvert the confidentiality, integrity or availability of information resources while security administrators assess their systems periodically and patch known vulnerabilities timely. Obviously the earlier and the more we know about the vulnerabilities in our systems, the more secure will they be. Discovery of vulnerabilities in a complex software system requires experience, creativity and good knowledge of the system, and unfortunately most people do not have this expert knowledge. So it is important for people to disseminate and share vulnerability knowledge with one another.

Software vulnerabilities are discovered by security researchers and disclosed in varying fashions. These include public disclosures, security advisories and security bulletins from vendors. However the terms and formats currently used in the field of

J. Zhou, M. Yung, Y. Han (Eds.): ACNS 2003, LNCS 2846, pp. 228–240, 2003.

computer vulnerability tend to be unique to different individuals and organizations. Moreover much of this information from different sources is informal, text-based description and can not currently be efficiently combined and shared among people not to mention support automated in-depth process.

After explaining a model of vulnerability life cycle, this paper presents a common vulnerability markup language (CVML) that describes vulnerabilities in a more structural and precise way based on XML. CVML intends to provide a uniform way to express and exchange information in the whole process of vulnerability management. With CVML, more manageable vulnerability databases will be built, promulgating and sharing of vulnerability knowledge will be easier, combine and refinement of vulnerability information from different sources will be done more efficiently. Moreover automatic scan and patch of system vulnerabilities are well supported. This paper is organized as follows: in section 2 the background and related work are discussed, then high level view of CVML is proposed in section 3, we give the document type definition (DTD) of CVML in section 4 and potential applications with CVML in section 5, Finally conclusions and future work is presented.

2 Background and Related Work

Complex information and communication systems give rise to design, implementation and management errors. These errors can lead to vulnerabilities—flaws in an information technology product that could allow violations of security policy. The discovery and exploitation of system vulnerabilities is requisite in each successful intrusion. In other words no vulnerabilities, no penetration.

2.1 Vulnerability Life Cycle

In general, vulnerabilities appear to transition through distinct states as depicted in Fig.1: birth, discovery, disclosure, the release of a fix, publication, exploitation or patching. Denoting the flaw's creation, birth usually occurs unintentionally during a large project's development when designers or developers of systems make mistakes. When someone (system administrator, attackers or other researchers) discovers that a product has security or survivability implications, the flaw becomes a vulnerability. It is clear that, in many cases, original discovery is not an event that can be known, the discoverer may never disclose his finding. Disclosure reveals details of the problem to a wider audience. It may be posted to a vulnerability-specific mailing list such as Bugtraq[7], or other security organizations. Vulnerability is correctable when the vendor or developer releases a software modification or configuration change that corrects the underlying flaw. Vulnerability becomes public when a computer security incident response center like CERT [8] or corresponding vendor could issue a report or advisory concerning the vulnerability. Usually correction is prior to publicity in order to prevent abuse of vulnerabilities.

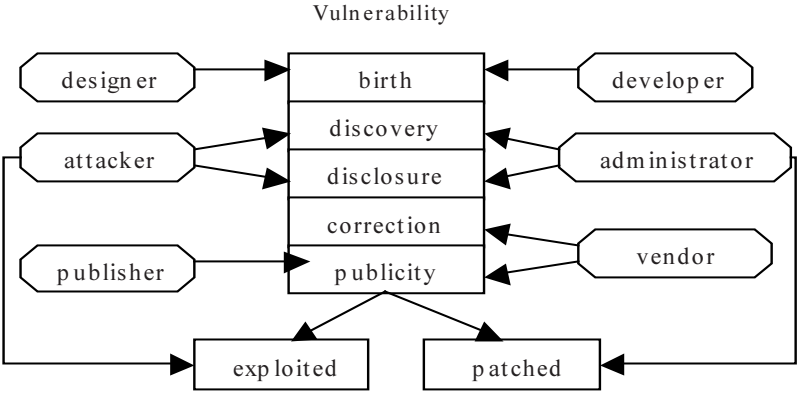

Fig. 1. Distinct states in vulnerability life cycle show a vulnerability-centered security architecture.

Attackers or security testers scan target systems, find vulnerabilities and exploit them for intrusions while security administrators take considerable time to get vulnerability information from security bulletins and scan their systems to patch known vulnerabilities timely. CERT or other computer security incident response teams respond, collect and publish vulnerability information. Software vendors are required to give advisories and patches of their products. In such a vulnerability-centered security architecture, lack of a consistent, structural and machine readable expression of vulnerability till now embarrasses automatic system scan and attack generation, interaction of different related products, combination and refining of vulnerability information from different sources. A specific example is six organizations have issued eight separate bulletins for one Microsoft Outlook vulnerability, with the time between the first and last bulletins spanning nearly a year. These bulletins variously described the vulnerability as the Outlook overrun vulnerability, MIME (Multipurpose Internet Mail Extensions) buffer overflow, MIME name vulnerability, long filename vulnerability, and mail tool vulnerability. The common vulnerability markup language we propose in next section can solve these problems.

2.2 Related Work

Most of the research on vulnerability has focused on discovery and collecting such as[7,8], there are also some meaningful studies on the classification of vulnerabilities in[2,9,10,11] which is the foundation of further description. To our knowledge, there is not a standard language to efficiently express vulnerabilities till now. But we can give some work toward the right direction.

The MITRE Corporation is well known for their creation of CVE[4]. The CVE is a dictionary of vulnerabilities that links vulnerability reports through a common naming system, but it doesn't include technical details or information on risk, impact, and

solutions. Still new and evolving, neither the lists of references nor the coverage of vulnerabilities is yet complete. It is more important to standardize all-sided descriptions of vulnerabilities than just names. Moreover the CVE names with the format CVE-YYYY-NNNN don't have a beneficial meaning. CVE also has a long screening process and a board to make final decisions, which makes it impossible to keep up with the flood of vulnerabilities.

MITRE just released a new standard for vulnerability assessment OVAL[5] in November, 2002. The open vulnerability assessment language is a common language for security experts to discuss and agree on technical details about how to check for the presence of a vulnerability on a computer system. The vulnerabilities are identified by OVAL queries like SQL that perform the checks. OVAL's purpose is only to check if software is vulnerable or not in other words in what conditions a vulnerability exists, which is only a part of our CVML. OVAL also faces the same problem of having a long screening process that might delay the release of vulnerability checks. However we can use OVAL in our CVML to check the existence of vulnerabilities.

The Open Web Application Security Project (OWASP) presented VulnXML in July, 2003 which is an open standard format for web application security vulnerabilities[6]. Anyone could describe web application security vulnerabilities with VulnXML so that the knowledge could be openly shared with both tools and users. VulnXML includes details about the vulnerability as well tests to check if the vulnerability exists. This is actually a step in the same direction as our CVML but VulnXML is for web application security vulnerabilities only.

The Intrusion Detection Exchange Format Working Group (IDWG) of the Internet Engineering Task Force is working on a common intrusion language specification [15] based on XML, which describes data formats for sharing information of interest to intrusion detection and response systems, and to management systems that may need to interact with them. However that work focuses on standardizing the information exported by intrusion detection systems (such as various alerts), but not on vulnerabilities themselves, and it is mainly to be used for automated intrusion detection.

3 Common Vulnerability Markup Language

CVML is an XML-based machine language to describe vulnerabilities in a uniform way for exchange and automatic process of vulnerability information. It is mainly composed of four sections of information about vulnerabilities: general related, judge-existence related, exploitation related, solution related.

3.1 Generally Related Information

In CVML general related information of vulnerabilities includes: ID, name, CVE name (optional), references, related dates, revision history, summary, classification, evaluation, additional note, credit. Most of these elements that are also included in

current vulnerability databases like [7,8] are easy to understand. However, we want to explain some characteristic elements.

As mentioned above the value of CVE name with the format CVE-YYYY-NNNN is little for organization and further process such as searching vulnerability for a specific product. In CVML the vulnerability name consists of a unique vendor identifier (e.g. MS for Microsoft, IBM for IBM, etc), the affected software name or the affected filename, the nature of the defect causing the problem and the impact. For instance "CVE-1999-0228" is named 'MS-NT-Service-Rpcss-InputValidation-DoS' in CVML. In this way we can organize the vulnerabilities into a forest-like namespace that facilitates classification and searching.

The element of classification demonstrates vulnerabilities in three aspects: how, where, when. How means the genesis of a vulnerability which can be one of these: Input Validation, Boundary Condition, Buffer Overflow, Access Validation, Exceptional Condition, Environmental Error, Configuration Error, Race Condition error, and other error etc. Where indicates the location of the vulnerability in the system (such as System Initialization, Memory Management, Process Management, Device Management, File Management, Identification and Authentication, Network Protocol, Support Software, Application, Hardware etc. when shows in what phase of the software life cycle the vulnerability is introduced (design, implementation, maintenance or operation). The taxonomy in CVML is based on previous work and contributes to the statistic and analysis of vulnerabilities that is important for the improvement of software design and development.

The element evaluation evaluates the vulnerability with three attributes: direct impact, total cost of exploit and rating. The direct impact tells the direct result caused by the vulnerability such as root access, user access, system down, disclosure of sensitive data etc. Total Cost of Exploit means the total resources (such as hardware and software resources) and skills (such as knowledge of specific system or the ability of programming with assembly on certain system) needed to exploit the vulnerability. It is approved that vulnerabilities with lower TOE incline to be more popular. Rating gives the severity of the vulnerability in the manner of quality or quantity. Generally speaking vulnerabilities with more severe direct impact and less total cost of exploit are more serious. Precise evaluation scheme of vulnerabilities is an interesting topic, but not the theme of this paper.

3.2 Check-Existence Related Information

The element check-existence provides information about how to check for the presence of vulnerability on a computer system. It includes three sub-elements: affected platform, affected software, and affected file to indicate where the vulnerability exists. Affected platform is composed of operating system and architecture (such as sparc, intel x86). Affected software consists of name and version. Affected file has two sub-elements: path and name. OVAL queries [5] can also be included for compatibility. Information here contributes to automatic vulnerability scan by security assess tools.

3.3 Solution Related Information

The section of solution contains information about how to correct the vulnerability. It consists of two parts: workaround and patch. Workaround provides operations taken in the absence of patching to avoid the vulnerability (such as disable vulnerable service, restrict buffer size with a system management tool, make specific configuration of the software etc). The element "configuration" in workaround can be classified into registry-configuration mainly referring to software running on Windows and file-registration referring to others. The element "registry configuration" contains registry key, name, type, data of a registry entry. The element "file configuration" demonstrates an entry in a designated file. The reference URL refers to sites for more information. Patch gives the patch name, a short note and the update URL pointing to a download site or a web service to fix the vulnerability, which can enable automatic update or hot-fix for system security. The sub-element patch-trace of patch specifies how to check whether the patch has been installed on the target system. It can include an entry name, an entry value of certain registry key for MS Windows, it also can refer to some specific entry in a specific file such as "inetd.conf" in Solaris. The sub-element patch-trace also contributes to system scan for vulnerabilities because a specific patch indicates the non-existence of one or more vulnerabilities. So we also can prevent updating software repeatedly in automatic system patching.

3.4 Exploit Related Information

The element of exploit is from the view of attackers or security testers and designed to support the automatic attack illation and generation. We can model attacks with 2-layer topology: functionality-level referring to concrete atomic vulnerability exploit and concept referring high-level logical description for illation. The exploit element contains three child elements: precondition, procedure, post-condition. Precondition and post-condition contribute to concept level description while procedure represents the functionality level.

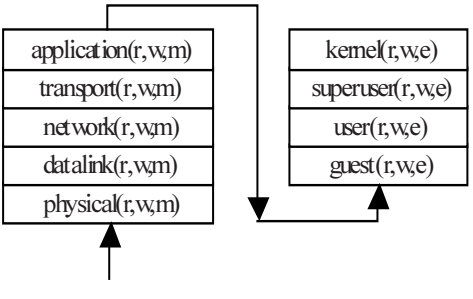

Fig. 2. Privilege Chaining

Precondition and post-condition gives both the privilege required or got on the target system for exploit of the vulnerability and high-level concepts in form of any attack description languages such as in [12,13]. As most of current operating systems are based on TCP/IP network, so we present a stratified structure of privileges shown in Fig.2. The right Modify on certain layer means the ability of arbitrary modification or redirection of entities on the layer. A successful computer intrusion consists of a few phases. An attacker increases his privilege on the target system in each phase of the attack until he gets the ultimate goal: super user privilege or absolute control of the system. For example an attacker will have the privilege Transport (Write) for a system without a firewall, which means he can write packets to the transport layer. He can use Nmap[14] to increase his privilege to Application (Read) that means he knows the services running on the target system. Then he can search for vulnerabilities in these applications running to elevate his privilege continually. Assuming vulnerability A can lead to local user access with remote access and vulnerability B can lead to super user access with local user access, therefore exploit sequence A, B would provide root access. In other words we can chain concrete vulnerabilities according to the privilege needed (in precondition) and got (in post-condition) to find a sequence of exploits taken for an appointed privilege, which we call privilege chaining. Privilege chaining can be used in attack generation although it is of a big granularity. We can also model a specific Operating System with finely granular privileges and make precise conclusion in intrusion through privilege chaining. Detail about definition and ratiocination of vulnerabilities in concept level is beyond the scope of this paper and it is confirmed that any study results in concept level can be used in CVML for ratiocination in automatic attack in a more formal and accurate way than privilege chaining.

Procedure gives the functionality-level steps taken by the attacker in the form of any attack description languages or source code in programming language such as c, c++, Perl etc. Compilation and execution of this part by the attacker will exploit the vulnerability and it can be called as a module by the attack engine in the concept-level based on the ratiocination made in the concept level.

4 DTD of CVML

We give the complete document declaration (DTD) of CVML and it is easy to understand with the comments.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT vulnerability (VId, VName, CVEName?, Summary, Date, History?, Reference*, Classification, Evaluation, CheckExistence, Solution?, Exploit, VNote, Credit)>

<!ELEMENT VId (#PCDATA)>
<!--This is a unique identifier number assigned to the vulnerability like “200305080001” -->
```

```
<!ELEMENT VName (#PCDATA)>
<!--This is the name in CVML like “MS-Windows-IE5.0-bufferoverflow”-->
<!ELEMENT CVEName (#PCDATA)>
<!--This is the CVE name if it exists-->
<!ELEMENT Summary (#PCDATA)>
<!--This is an abstract of the vulnerability -->
<!ELEMENT Date EMPTY>
<!ATTLIST date
discovery  CDATA #REQUIRED
firstPublish  CDATA #REQUIRED
lastModified  CDATA #REQUIRED>
<!--This field tells the related dates of discovery, publish and last modified -->
<!ELEMENT  History (Revision+)>
<!ELEMENT  Revision (RevisionDate, RevisionNote)>
<!ATTLIST  Revision version CDATA #REQUIRED>
<!ELEMENT RevisionDate (#PCDATA)>
<!ELEMENT RevisionNote (#PCDATA)>
<!--  This field is intended to record the revision history of the vulnerability -->
<!ELEMENT Reference (#PCDATA)>
<!ATTLIST Reference
                Source  CDATA #REQUIRED
                URL  CDATA #IMPLIED >
<! --This field provides resources such as Bugtraq bid’s, CERT announcements etc.
URL should point directly to the entry of the particular vulnerability if possible -->

<!ELEMENT Classification (Genesis, Location, Introduce)>
<!ELEMENT Genesis (#PCDATA)>
<!--This indicates the genesis of the vulnerability, its value can currently be one of
the following: Input Validation, Boundary Condition, Buffer Overflow, Access Vali-
dation, Exceptional Condition, Environmental Error, Configuration Error, Race Con-
dition error, and other error etc.-->
<!ELEMENT Location (#PCDATA)>
<!--This indicates the location of the vulnerability in computer system. It can be one
of the following: System Initialization, Memory Management, Process Management,
Device Management, File Management, Identification and Authentication, Network
Protocol, Support Software, Application, Hardware etc. -->
<!ELEMENT Introduction (#PCDATA)>
<!-- This field indicates when the vulnerability is introduced in the system. It can be
one of the following: design, implementation, maintenance or operation -->

<!ELEMENT Evaluation EMPTY>
<!ATTLIST Evaluation
            Impact CDATA #REQUIRED
            AttackCost CDATA #REQUIRED
            Rating CDATA #REQUIRED>
```

<!-- This field evaluates the vulnerability in 3 aspects, Impact can be root access, user access, system down, disclosure of sensitive data etc. Attack cost or rating now can be high, medium, low. They also can be a number got through certain evaluation scheme. -->

<!ELEMENT CheckExistence (Platform+, AffectedSoftware+, Affectedfile*) >
<!ELEMENT Platform (OS,Arch)>
<!ELEMENT OS (OSName, OSVersion)>
<!ELEMENT OSName(#PCDATA)>
<!ELEMENT OSVersion(#PCDATA)>
<!-- This should be the official prodocut name and version such as Microsoft Windows NT Workstation 3.51 -->
<!ELEMENT Arch (#PCDATA)>
<!-- This should be the official architecture string such as i386, ia64 , ppc etc. If the vunerability exists in all the architecture running under certain OS, it can be blank. -->
<!ELEMENT AffectedSoftware (SoftwareName, SoftwareVersion)
<!ELEMENT SoftwareName (#PCDATA)>
<!ELEMENT SoftwareVersion (#PCDATA)>
<!--This gives name and version of the affected software -->
<!ELEMENT AffectedFile (FileName, Filepath, fileattribute?)>
<! ELEMENT FileName (#PCDATA)>
<!ELEMENT FilePath (#PCDATA)>
<!ELEMENT FileAttribute (#PCDATA)>
<!-- This section gives the affected file if necessary-->

<!ELEMENT Solution (Workaround*, Patch*)>

<!ELEMENT WorkAround (WorkNote, Configuration?, WorkUrl*)>
<!ELEMENT WorkNote (#PCDATA)>
<!-- This is the note of the workaround.-->
<!ELEMENT Configuation (RegistryCfg*,FileCfg*)>
<!ELEMENT RegistryCfg (CfgKey, CfgName, CfgType, CfgData)>
<!ELEMENT CfgKey(#PCDATA)>
<!ELEMENT CfgName(#PCDATA)>
<!ELEMENT CfgType(#PCDATA)>
<!ELEMENT CfgData(#PCDATA)>
<!-- This indicates the registry entry for software configuration on Windows in the workaround to wipe off the vulnerability-->
<!ELEMENT FileCfg (CfgPath, CfgName, CfgEntry)>
<!ELEMENT CfgPath(#PCDATA)>
<!ELEMENT CfgName(#PCDATA)>
<!ELEMENT CfgEntry(#PCDATA)>

```
<!--  This demonstrates an entry in a designated file for software configuration in the workaround to wipe off the vulnerability-->
<!ELEMENT WorkUrl (#PCDATA)>
<!--This is the workaound with a few URLs pointing to detailed information-->

<!ELEMENT Patch (PatchName, PatchNote, PatchTrace, PatchUrl*)>
<!ELEMENT PatchName (#PCDATA)>
<!ELEMENT PatchNote (#PCDATA)>
<!ELEMENT PatchTrace(RegistryTrc*, FileTrc*)>
<!ELEMENT RegistryTrc (TrcKey, TrcName, Trctype, TrcData)>
<!ELEMENT TrcKey(#PCDATA)>
<!ELEMENT TrcName(#PCDATA)>
<!ELEMENT TrcType(#PCDATA)>
<!ELEMENT TrcData(#PCDATA)>
<!--  This indicates the registry entry for check for the existence of patch-->
<!ELEMENT FileTrc (TrcPath, TrcName, TrcEntry)>
<!ELEMENT TrcPath (#PCDATA)>
<!ELEMENT TrcName (#PCDATA)>
<!ELEMENT TrcEntry (#PCDATA)>
<!--  This indicates the file entry for check for the existence of patch-->
<!ELEMENT PatchUrl (#PCDATA)>
<!--This gives the patch and related download sites. -->


<!ELEMENT Exploit (Precondition, Procedure, Post-condition)>

<!ELEMENT Precondition (PrivilegeNeed, PreConcept)>
<!ELEMENT PrivilegeNeed (#PCDATA)>
<!ATTLIST  PrivilegeNeed  language CDATA#REQUIRED>
<!ELEMENT PreConcept (#PCDATA)>
<!ATTLIST PreConcept  language CDATA#REQUIRED>
<!--  This gives the privilege needed for exploit in any language and high-level concept in any language. -->

<!ELEMENT Procedure(#PCDATA)>
<!ATTLIST  Procedure language CDATA#REQUIRED>
<!--  This is the procedure of exploit in source code or any other form of certain language-->

<!ELEMENT PostCondition(PrivilegeGot, PostConcept)>
<!ELEMENT PrivilegeGot(#PCDATA)>
<!ATTLIST  PrivilegeGot language CDATA#REQUIRED>
<!ELEMENT PostConcept(#PCDATA)>
<!ATTLIST  PostConcept  language CDATA#REQUIRED>
```

<!-- This gives the privilege got and high-level conceptual result of the exploit in any language. -->

<!ELEMENT VNote (#PCDATA)>
<!-- This is additional note of the vulnerability.-->
<!ELEMENT Credit (#PCDATA)>
<!-- This gives the acknowledgment-->

5 Application of CVML

CVML describes security vulnerabilities using XML in a structured, machine-readable way and provides a new security interoperability standard. It can be widely used in vulnerability research and security management of computer systems. We have implemented a prototype of automatic vulnerability management architecture shown in Fig.3.

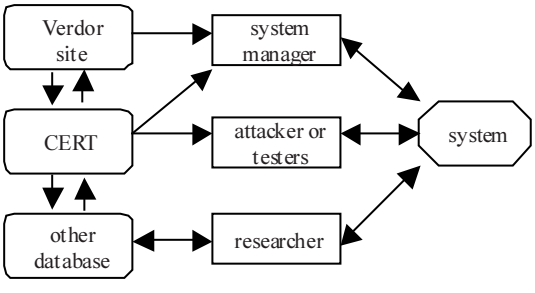

Fig. 3. Vulnerability Management with CVML

With CVML we can storage structured vulnerability information in database. Better analysis and data mining of vulnerabilities (for instance based on the element classification, evaluation etc) in individual database are provided. In our prototype, the combining component of database can collect, compare and combine vulnerability information from any sources (such as other databases, online submission etc.) in CVML, therefore leading to timely share of vulnerability knowledge and a large distributed database on the internet. The answering component of the database answers to a standard request from any sources (such as System Manager, Attack Tools, Assess Tools etc.) with related information in CVML. The subscription provider component will send messages in CVML about new vulnerabilities relevant to certain platform or software to the clients as soon as possible. Thus all kinds of security tools compatible with CVML could interact with vulnerability databases conveniently to support security scan or test.

According to a survey by SecurityFocus [7] system security administrators spend an average of 2.1 hours/day hunting for security information relevant in all kinds of security bulletins and mailing lists, then download and install patches to fix the sys-

tem. It is important to make this procedure automatic. In our prototype, System Manager is in charge of the security administration of one or more computer systems (usually in a local network). Ideally we expect in future all software installed on the system should register in the manager with the security related information (such as the version, the configuration, the update or patch sites etc). Under current circumstance system manager will scan the system periodically to get the related information of all software installed, the scan result is put into a XML file called System Description. Then system manager can use assess tools or assess engine in itself to check if any vulnerability exists using the CheckExist, Workaround, and Patch in CVML. The assess engine could maintain locally a cache of vulnerabilities relevant to its scope of usage. It also could inquire or subscribe for related vulnerabilities from certain databases. According to rules made by system administrators, System Manager will take some action (such as downloading and installing the patch automatically or only warning).

The attack tools or attack engine can be used for testing the security of systems or performing real attacks. It is based on the System Description (for test) or other system scan results (for real attacks). We have implemented in the prototype the privilege chaining process and some concept-level ratiocination of known vulnerabilities for automatic attack generation mainly using the element Exploit in CVML. Its interaction with vulnerability databases is similar to assess tools or assess engine.

We can see in our vulnerability management architecture, the component system manager scans, assesses, and tests the system periodically, update vulnerable software automatically when possible. This self-managing of systems not only alleviates the heavy burden of security administrators, but also improves the security of system remarkably.

6 Conclusions and Future Work

Until now, discovering, disclosing and patching vulnerabilities efficiently in information systems play the central role in security area. Current vulnerability information from different sources is mainly ambiguous text-based description that is not machine-readable and can't be efficiently shared, data-mined, combined or other in-depth automatic process. This paper presents a common vulnerability markup language (CVML) based on XML that describes vulnerabilities in a more structural way to support unified expression, easy sharing, automated management of vulnerability information. There are mainly 4 parts in CVML: General information, how to Check Existence, Solutions, and how to Exploit. In CVML, vulnerabilities can be described precisely from the view of both attackers and defenders. Architecture of automated vulnerability management based on CVML is also proposed. In this architecture, more manageable vulnerability databases are built; promulgating and sharing of vulnerability knowledge are easier; compare, and fusion of vulnerability information from different sources are more efficient; moreover automated scanning and patching of vulnerabilities leads to self-managing systems.

It is obviously not precise enough to check the existence of a vulnerability only based on the name and version of software installed. How can we do more efficiently? The granularity of privilege chaining is too big, how to improve it? How to improve the concept-level ratiocination of attack generation? How to improve the interaction between the concept-level and the functionality-level? These problems are in the scope of our future work.

References

1. IEEE.: The IEEE Standard Dictionary of Electrical and Electronics Terms. Sixth Edition. Institute of Electrical and Electronics Engineers Inc., New York NY (1996) 373
2. Amoroso, E.G.: Fundamentals of Computer Security Technology. Prentice-Hall PTR, Upper Saddle River NJ (1994)
3. John, D.H., Thomas, A.L.: A Common Language for Computer Security Incidents. Sandia Report, Sand98-8667. Livermore CA USA (1998)
4. http://cve.mitre.org
5. http://oval.mitre.org
6. http://www.owasp.org/vulnxml
7. http://www.securityfocus.com
8. http://www.cert.org
9. Carl, L.: A Taxonomy of Computer Program Security Flaws. Technical Report. Naval Research Laboratory (1993)
10. Brian, M.: A Survey of Software Fault Surveys. Technical Report, UIUCDCS-R-90-1651. University of Illinios at Urbana-Champaign (1990)
11. Taimur, A. (ed.): Use of A Taxonomy of Security Faults. Technical Report, TR96-051. COAST Laboratory, Department of Computer Sciences, Purdue University (1996)
12. Eckmann, S., Vigna, G., Kemmerer, R.: STATL. Technical Report, UCSB (2000)
13. Cuppens, F., Ortalo, R.: Lambda: A Language to Model a Database for Detection of Attacks. In: Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)
14. http: //www.insecure.org/nmap
15. http:// www.ietf.org/ids.by.wg/idwg.html