Maintaining Software with a Security Perspective

Kanta Jiwnani
Department of Computer Science
University of Maryland
College Park, Maryland 20742.
kanta@cs.umd.edu

Marvin Zelkowitz
Department of Computer Science, University of Maryland
and Fraunhofer Center for Experimental Software Engineering
College Park, Maryland 20740.
marv@zelkowitz.org

Abstract

Testing for software security is a lengthy, complex and costly process. Currently, security testing is done using penetration analysis and formal verification of security kernels. These methods are not complete and are difficult to use. Hence it is essential to focus testing effort in areas that have a greater number of security vulnerabilities to develop secure software as well as meet budget and time constraints. We propose a testing strategy based on a classification of vulnerabilities to develop secure and stable systems. This taxonomy will enable a system testing and maintenance group to understand the distribution of security vulnerabilities and prioritize their testing effort according to the impact the vulnerabilities have on the system. This is based on Landwehr's classification scheme for security flaws and we evaluated it using a database of 1360 operating system vulnerabilities. This analysis indicates vulnerabilities tend to be focused in relatively few areas and associated with a small number of software engineering issues.

Keywords: Intrusion classification, Security, Testing, Vulnerabilities

1. Introduction

Any piece of software containing a security flaw can make a secure environment vulnerable to an attack. We define an attack as allowing an individual to either obtain information from a computer system for which access rights have not been lawfully obtained or cause the system to be unusable for its intended purpose. We will refine this definition later.

Software vulnerabilities are present due to many causes: poor development practices, ignoring security policies during design, incorrect configurations, improper initialization, inadequate testing due to deadlines imposed by financial and marketing needs etc. Security is not considered during all phases of development, but is often added later in an ad-hoc manner. Security models often describe the security policies to be followed, but the actual implementation cannot follow the model exactly. Therefore security compromises are made.

The increasing exploits of vulnerabilities in systems show that there are growing needs to develop more secure software. As we later show, security vulnerabilities often remain from one release of an operating system to the next. Providing a mechanism for focusing security concerns could allow system developers to more effectively address security issues over the lifetime of a product.

Later versions of software often contain vulnerabilities that exploit the same characteristics or conditions that were exploited by attackers in the earlier versions. Programmers and testers are neither taking adequate measures to keep a check on these characteristics or conditions by neither improving programming practices nor testing for these characteristics or conditions specifically. A classic example of this problem seen even today is the Windows XP buffer overflow vulnerability discovered shortly after its release. This could allow hackers to take over a computer and erase disks, alter data and install their own programs. In spite of Microsoft assigning one or two testers for each developer, and using a large arsenal of testing methods (e.g., usage tests, user interface tests, gorilla tests, "bug bash" tests, usage and stress tests, application programming interface tests, Applets tests, NT verify tests, regression tests, Apps-16 and 32 tests, Non-NT tests, Synthetic GUI Application tool tests, RATS tests, ad-hoc tests and other testing strategies [1]) a simple and most widely known flaw could destroy the security of Microsoft's newest version of Windows. This shows that general testing strategies in use in organizations are insufficient.

Our goal is to devise a classification of vulnerabilities to abstract information about problems in software development, their location and their impact on the system to concentrate and increase testing effort in those areas, and help the software development community to get feedback to improve continuously. Security rests upon confidentiality, integrity, and availability of information, and we have tried to base our taxonomy on these principles.

The taxonomy presented in this paper endeavors to classify vulnerabilities in systems to find which parts of systems have a greater concentration or which types of vulnerabilities are dominant in various systems so as to focus the type of testing that would be needed to find them. This regularity information, if it exists, may guide the testing group to write test cases to discover greater number of vulnerabilities even before they can be exploited and help prevent and eliminate them during the development process itself. This may also help system designers to integrate security requirements while building new systems. This may also reduce the cost of maintaining systems since most of the vulnerabilities will be eliminated at the development stage itself. Using this approach, it would be possible to identify and eliminate flaws like Windows XP buffer vulnerability before software deployment.

The next section reviews security testing strategies and classification schemes for security flaws and describes our own model. In Section 3, we apply this taxonomy in evaluating over 1360 security flaws found in several releases of Windows NT and Linux. We show that the use of a taxonomy such as ours could focus testing activities to help uncover such vulnerabilities more easily in future system releases. Section 4 concludes the work followed by potential for future work.

2. Security Classifications

Presently, organizations test the security of their systems, firewalls and networks either by using commercially available vulnerability analysis tools like STAT Scanner, ISS Scanner, Cybercop, by hiring a "tiger" team to simulate hackers (i.e., penetration testing), or by using formal methods. Penetration testing tests the network on a particular day and its results may vary from day to day. It does not find all the vulnerabilities in systems [2] and is prone to several difficulties [3]. Formal methods [4, 5] use mathematical description of the system implemented to verify whether the system does actually meet all the security requirements. However, it is difficult to specify the requirements and the system in a mathematical form.

Other methods for security testing have been developed, including syntax testing [6], property-based testing [7], fault injection [8, 9], mutation testing [10] and Gligor's testing method [11]. Again these techniques are limited to finding specific security flaws. Also, there are the general testing techniques like path testing, domain testing, and data flow testing [12]. However these techniques are not specifically adapted for security issues.

2.1. Classification schemes

A number of security flaw taxonomies have been developed including the Protection Analysis (PA) Taxonomy (1978), the Research in Secured Operating Systems (RISOS) security taxonomy (1976), Spafford's taxonomy (1992), Landwehr's taxonomy (1994), Aslam's taxonomy (1995), Bishop's taxonomy (1995), Du and Mathur's taxonomy (1997), Brian Marick Survey (1990) and Chillarege's Orthogonal Defect Classification**.** This section reviews the above taxonomies. We will redefine a security attack as executing any software whose purpose is to utilize one of the security vulnerabilities identified by one of these taxonomies.

Flat taxonomies. We define a flat taxonomy as one that divides the set of security vulnerabilities according to one general criterion. These are the simplest taxonomies. We then look at multidimensional taxonomies that seek to classify such vulnerabilities according to several such criteria.

Protection Analysis (PA) Taxonomy: As a 25 year old taxonomy, it was one of the first to address security concerns. The objective of the PA project [13] was to provide a basis for categorizing protection errors according to their security-relevant properties using an automated, pattern-matching approach. This taxonomy was based on 100 flaws found in six different operating systems. It had four global categories: improper protection (initialization and enforcement), improper validation, improper synchronization, and improper choice of operand or operation. The categories in this taxonomy were broad and the same flaw could be classified into multiple categories. The contribution of this study was the introduction of several types of security flaws like time-of-check-to-time-of-use (TOCTTOU), allocation/deallocation of residuals, and serialization errors that remain relevant.

RISOS Taxonomy: The Research in Secured Operating Systems (RISOS) security taxonomy [14] was based on flaws found in three operating systems: IBM's OS/MVT for the IBM 360, UNIVAC's 1100 Series operating system, and Bolt Beranek and Newman's TENEX system for the PDP-10. The classification consisted of seven categories: incomplete parameter validation, inconsistent parameter validation, implicit sharing of privileged or confidential data, asynchronous validation or inadequate serialization, inadequate identification or authentication or authorization, violable prohibition or limit and exploitable logic error. The main contribution of this study was the classification of integrity flaws found in operating systems. It also led to classifying the same flaw in multiple categories.

Spafford's taxonomy: Spafford characterized several common system vulnerabilities [15] with operational (administrative) flaws, design flaws, and faults as its three main categories. It did not help us to abstract detailed information about characteristics of vulnerabilities.

Aslam's Taxonomy: Aslam defined a classification of security faults [17, 18] in the Unix Operating System. The classification scheme had two broad categories: coding faults comprising faults introduced during software development and emergent faults resulting from improper installation of software, unexpected integration incompatibilities, or when a programmer fails to completely understand the limitations of the run-time modules. It attributes the cause of all non-synchronization security errors to the improper evaluation of condition. This is a very narrow viewpoint since it may be possible to correct the error without even changing any condition in the program. Selection Criteria, software testing methods and design and implementation of a prototype database to store vulnerability information were also specified.

Brian Marick Survey: Marick published a survey of software fault studies [22] from the software engineering literature. Most of the studies reported faults that were discovered in production quality software. The results were insightful but no conclusions about development phases were possible.

Chillarege's Orthogonal Defect Classification [23, 24] is a method developed at IBM's Watson Research Center for classifying software defects based upon the semantics of defect correction and links the defect distribution to the development progress and maturity of the product. But *ODC Triggers* do not reflect security issues.

Multidimensional taxonomies. These taxonomies classify flaws according to more than one attribute.

Bishop's taxonomy: Bishop proposed a flaw taxonomy [19] for the Unix Operating System. The taxonomy uses six axes classifying every vulnerability as: nature of the flaw, time of introduction, exploitation domain, effect domain, minimum number of components needed to exploit the vulnerability, and source of identification. However, these axes do not divide the security domain according to software functionality.

Du and Mathur's Taxonomy: They classified flaws from three viewpoints: cause of the flaw, the nature of their impact, and the type of change or fix made to remove the flaw [20, 21]. The first dimension was similar to Landwehr's while the third dimension has categories like spurious entity, missing entity, misplaced entity and incorrect entity that do not cover all possibilities.

Landwehr's Taxonomy: Carl Landwehr et al [16] categorized 50 security flaws according to three criteria: (1) the *genesis* of the flaw (how did it enter the system?), (2) *time of introduction* (when did it enter the system?), and (3) *location* (where in the system is it manifested?). Since our goal was to look at the impact that security flaws have on an evolving product, how a flaw occurs, when it occurs, and its impact (i.e. location) appeared to be the right mix of criteria. We centered on Landwehr's model as the basis for our work.

2.2. Our Taxonomy

Landwehr's categorization of security flaws gave us the realization that genesis and location were the two most useful dimensions from our perspective. We also included a third dimension, the impact of the vulnerability on the system. Critical impact areas should prioritize testing effort. This 3-dimensional classification scheme helped us to extract information from the set of vulnerabilities and to discover regularities in vulnerabilities across different operating systems. This abstraction tells us about frequent occurrences of security errors indicating that either current development practices overlook these checks, or testing is not done adequately for these specific conditions. With this information, it is possible to assign a testing group to test for these frequent occurrences more vigorously.

Each vulnerability was classified according to the following classification.

2.2.1. Software development issues. Landwehr's genesis (Figure 1) provided the basis for describing the way each vulnerability entered the system. In general it is the type of security flaw, which is present:

- *Validation errors* occur when a program fails to check that the parameters supplied or returned to it conform to its assumptions about them, or when these checks are misplaced.
- *Domain errors* occur when the intended boundaries between protection environments are porous including implicit sharing of privileged/confidential data or when then the lower level representation of an abstract object, supposed to be hidden in the current domain, is in fact exposed.
- *Serialization flaws* permit asynchronous behavior of different system components to be exploited. Many time-of-check-to-time-of-use (TOCTTOU) flaws fall in this category. *Aliasing flaws* arise when two names for the same object can cause its contents to change unexpectedly and consequently, invalidate checks already applied to it. Serialization and aliasing flaws are combined into one category.
- An *identification/authentication flaw* permits operations to be invoked without sufficiently checking the identity and the authority of the invoking entity.
- *Boundary condition flaws* occur due to omission of checks to assure that constraints (table size, file allocation, or other resource consumption) are not exceeded.
- *Trojan horse* refers to a program that masquerades as a useful service but exploits rights of the program's user – rights not possessed by the author of the Trojan horse – in a way the user does not intend.
- *Covert Channel* is defined as a path to transfer information in a way not intended by the system's designers.
- *Other Exploitable logic errors* include all errors that do not fall in any of the above categories.

We have simplified this categorization by not distinguishing between intentional and inadvertent as well as malicious or non-malicious flaws in Landwehr's taxonomy. From a testing perspective, it is essential to test systems adequately to discover such security flaws, and the programmer's intent is not important in this context. A *trapdoor* is a hidden piece of code that responds to a special input, allowing its user access to resources without passing through the normal security enforcement mechanism. A *logic/time bomb* is a piece of code that remains dormant in the host system until a certain “detonation” time or event occurs. Both these flaws occur only when the developer deliberately includes them in software. In terms of discovery, they are the same as a Trojan horse, only intent differs. We also redefine the following category to eliminate ambiguity:

- *Exploitable logic errors* occur due to use of incorrect logic during implementation.

Our simplified genesis dimension is then: validation errors, domain errors, serialization or aliasing errors, errors due to inadequate identification or authentication, boundary and condition errors, Trojan horse, covert channel and exploitable logic (Column 1 of Figure 3).

2.2.2. Location of flaws in the system. This dimension (Figure 2) describes the location in software where the vulnerability is present:

- *System Initialization:* Flaws in this area can occur either because the operating system fails to establish the initial protection domains as specified or because the system administrator has not specified a secure initial configuration for the system.
- *Memory Management* and *Process Management* are functions that operating systems provide to control memory and CPU time. Errors in these functions may permit one process to gain access to another improperly or to deny service to others. These two categories are separate in the taxonomy.
- *Device Management* errors occur when the I/O routines fail to respect parameters provided to them or when they validate parameters provided in memory locations that can be altered, directly or indirectly, after checks are made by user programs.
- *File Management:* Operating systems include file systems, which implement access controls to share and protect their files. Errors in these controls, or in the management of the underlying files, can result in security flaws.
- *Identification/Authentication* functions of the operating system maintain special files for user Ids and passwords and provide functions to check and update those files as appropriate. It is important to scrutinize these functions as well as scrutinize all the possible ports of entry into a system to ensure that these functions are invoked before a user is permitted to consume or control other system resources.

As with the genesis dimension, we have modified the location dimension of Landwehr’s taxonomy. Landwehr et al. proposed a general taxonomy to include flaws found in operating systems, hardware, support software or application (user) software. Since we were seeking a taxonomy to help us find vulnerabilities in operating systems, we only include those categories that fall under operating systems. Also, we have eliminated the category called *Other/Unknown*.

2.2.3. Impact of flaws on the system. This dimension describes the effect on the system due to an exploit of a vulnerability. These are the visible impact of an attack. This dimension can be prioritized to suit an organization’s testing efforts. The categories are as follows:

Genesis	**Intentional**	**Malicious**	Trojan Horse	Non-Replicating
				Replicating (virus)
			Trapdoor	
			Logic/Time Bomb	
		Non-Malicious	Covert Channel	Storage
				Timing
			Other	
	Inadvertent	Validation Error (Incomplete or Inconsistent)		
		Domain Error (Including Object Re-use, Residuals, and Exposed Representation Errors)		
		Serialization or aliasing		
		Identification or Authentication Inadequate		
		Boundary Condition Violation (Including Resource Exhaustion and Violable Constraint Errors)		
		Other Exploitable Logic Error		

Figure 1. Landwehr’s Genesis of security flaws.

Location	**Software**	**Operating Systems**	System Initialization
			Memory Management
			Process Management/Scheduling
			Device Management (Including I/O, networking)
			File Management
			Identification/Authentication
			Other/Unknown
		Support	Privileged Utilities
			Unprivileged Utilities
		Application	
	Hardware		

Figure 2. Security flaw taxonomy: Flaws by Location.

- *Unauthorized Access:* Action(s) that result in any disclosure/modification of data, use of resources, or execution of code with higher privileges by a user violating the system security policy.
- *Root/System Access:* Actions that allow an attacker to execute system processes or take any action with system/root privileges violating the system security policy.
- *Denial of Service:* Actions that prevent any part of a system from functioning in accordance with its intended purpose or delay time critical operations. This may prevent authorized users to access resources or system services.
- *Integrity Failure:* Actions that result in disclosure of system state information violating the system security policy.
- *Crash/Hang/Exit: Crash* may result due to actions that result in sudden, sometimes drastic failure of a software application, or operating system or a device such as a disk. A system may *hang* when computer programs conflict or do not run properly due to malicious action(s) by an attacker paralyzing the system. *Exit*: Action(s) resulting in an unexpected termination of an application/service.
- *Failure:* Action(s) leading to temporary or permanent termination of the ability of an application, system service to perform its required function.
- *Invalid State:* Action(s) that lead to a system state not permitted by the system security policy.
- *File Manipulations:* Action(s) that result in unauthorized access, modification, or deletion of file contents by a user without the required privileges.
- *Errors due to clock changes:* Action(s) leading to system clock access that may result in an unpredictable system state.

Development Issues	**Location**	**Impact**
Validation Errors	System Initialization	Unauthorized Access
Domain Errors	Memory Management	Root or System Access
Serialization or aliasing errors	Process Management or Scheduling	Denial of Service
Errors due to Inadequate Identification or Authentication	Device Management	Integrity Failure
Boundary and Condition Errors	File Management	Crash, Hang, or Exit
Trojan Horse	Identification or Authentication	Failure
Covert Channel		Invalid State
Exploitable Logic Errors		File Manipulations
		Errors due to clock changes

Figure 3. Security Flaw Taxonomy from a Security Testing Perspective

By making the simplifying assumptions given previously, our taxonomy is shown schematically in figure 3.

3. Applying the taxonomy

We wanted to see if our taxonomy could be useful in finding vulnerabilities in released software. We obtained a file of 1200 vulnerabilities found in Windows NT from Harris Corporation and 160 in Linux compiled from Red Hat Linux Errata, and we classified those vulnerabilities that existed in successive releases of the software in order to see if the classification mechanism identified error prone components of the system.

Rank	Development Issues	Location	Impact	No.
		Windows NT		
1	Exploitable Logic	System Initialization	Unauthorized Access	115
2	Identification/ Authentication	System Initialization	Unauthorized Access	109
6	Identification/ Authentication	Identification/ Authentication	Unauthorized Access	42
8	Validation Error	Memory Management	Unauthorized Access	34
		Linux		
1	Validation Error	Memory Management	Unauthorized Access	25
2	Identification/ Authentication	Identification/ Authentication	Unauthorized Access	13
7	Exploitable Logic	System Initialization	Unauthorized Access	5
9	Identification/ Authentication	System Initialization	Unauthorized Access	4

Figure 4. Common vulnerabilities of WINNT and Linux

3.1 Distribution of vulnerabilities

The first dimension of our taxonomy (Figure 3), Software Developing Issues, has eight categories. The second dimension, Location of flaws, has six categories and the third dimension, Impact of flaws, has nine categories. Hence, we had 8x6x9 = 432 possible triples for each vulnerability. As a first test we classified each flaw and ranked the triples according to the number of vulnerabilities present in each triple where Rank 1 indicates the highest number of vulnerabilities. The last column in Figure 4 indicates the number of vulnerabilities found in a triple. If errors occurred randomly, then each triple should have approximately the same number of vulnerabilities. However, in both Windows NT and Linux, four of the top ten triples were the same (Figure 4). These areas seem appropriate for increased system testing.

3.2 Repetitive security failures

We classified 1360 vulnerabilities found in Windows NT Versions 3.51, 4.0, 2000, XP and Red Hat Linux Versions 5.2, 6.2, 7.0, and 7.1. We used the STAT Scanner, a Vulnerability Assessment Tool for Windows, Unix and Linux environments, developed by Harris Corporation to analyze vulnerability trends in Windows NT 4.0 and Linux systems. The following data is based upon scanning systems with Windows NT 4.0 Service Pack 1 through Service Pack 6a and Post-SP6a Security Rollup Package (i.e., interim bug fix releases to Windows NT 4.0) to collect the following:

- Number and type of security flaws *present* in each service pack.
- Number and type of security flaws *fixed* in each successive service pack from the previous release.
- Number and type of new security flaws *found* in each successive service pack, which were absent in the prior service pack.

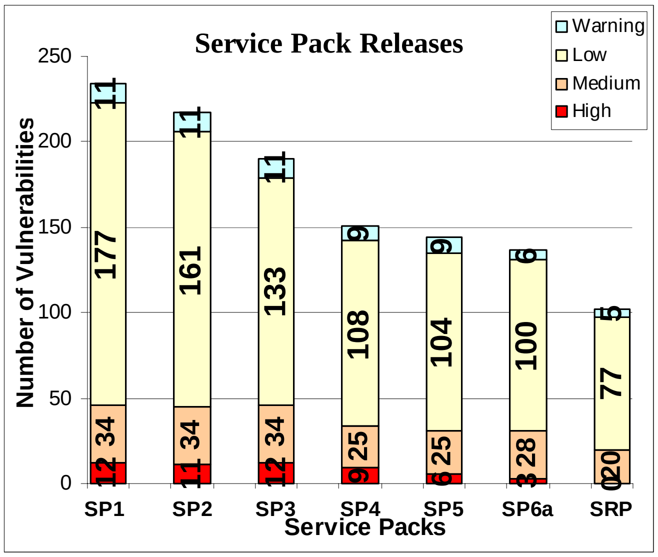


Figure 5. Number of security flaws in Service Packs according to risk levels.

The successive changes in security flaws present in each release provide more relevant data than just the total number of flaws present. It provides a snapshot of how such flaws are found and fixed between service pack releases. Starting with a baseline of 234 vulnerabilities present in Service Pack 1 (SP1) given in Figure 5, the bars above the X-axis in Figure 6 indicate the number of flaws that were not present in prior releases (i.e., represent new security flaws), while those below indicate the number of flaws present in the prior service pack, but were fixed in this service pack. It can be seen that the

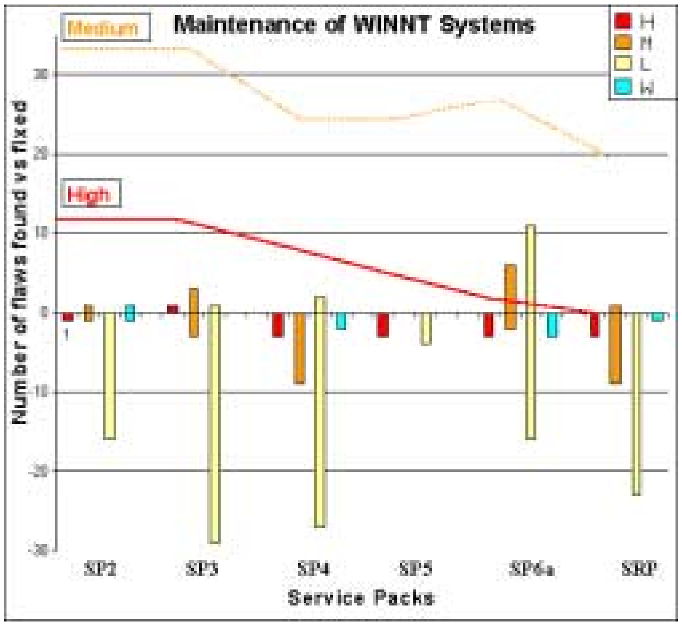


Figure 6. Number of remaining/found/fixed security flaws in successive service packs.[*]

majority of flaws fixed in each service pack release are low risk level. New medium level flaws were introduced in SP2, SP3, SP6a and SRP. Only one high-risk flaw was detected during the releases (in SP3). Medium risk flaws continue to exist in future releases while high-risk flaws get fixed.

Our goal was to look where security flaws occurred, not just their number. We classified the vulnerabilities found in each service pack according to our taxonomy and ranked the various categories under each dimension by the number of vulnerabilities present in each category. Rank 1 indicates the highest number of vulnerabilities. (The data is given in Figures 11 and 12.) The results of this ranking are shown in Figures 7 and 8. The maximum number of flaws were found in System Initialization in the location dimension in all Service Packs and in Exploitable logic (Rank 1 in SPs 1, 2 and 3 and Rank 2 in SPs 4, 5 and 6) and in Inadequate Identification or Authentication (Rank 1 in SPs 4,5 and 6 and Rank 2 in SPs 1, 2 and 3) in the dimension of Software development issues.

We also compared the distribution of high and medium risk level flaws with the total distribution of security flaws. We again ranked the categories in each dimension according to the number of high and medium flaws present in each category. This distribution or ranking scheme is shown in Figures 9 and 10. Comparing these figures with Figures 7 and 8 respectively, we can conclude that the distribution of high and medium risk level security flaws reflects the same regularities as the distribution of total number of vulnerabilities. Hence, using this taxonomy to identify high-risk flaws in one release of a system (defined as the rank of security flaws for that dimension), may potentially eliminate or prevent a majority of security flaws by orienting testing to search more intensely for these flaws.

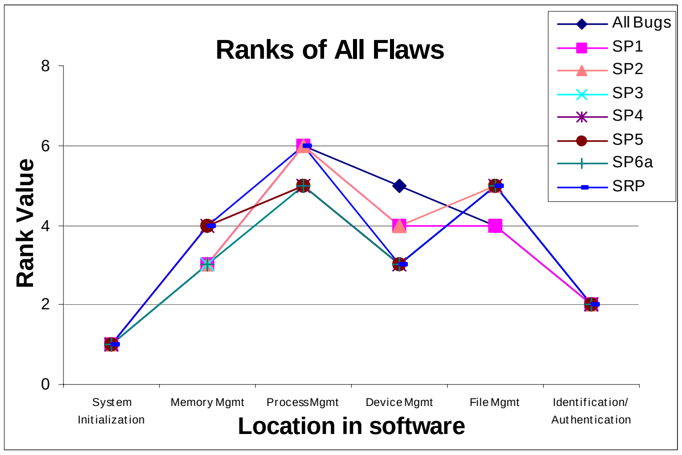


Figure 7. Ranking of locations in software.

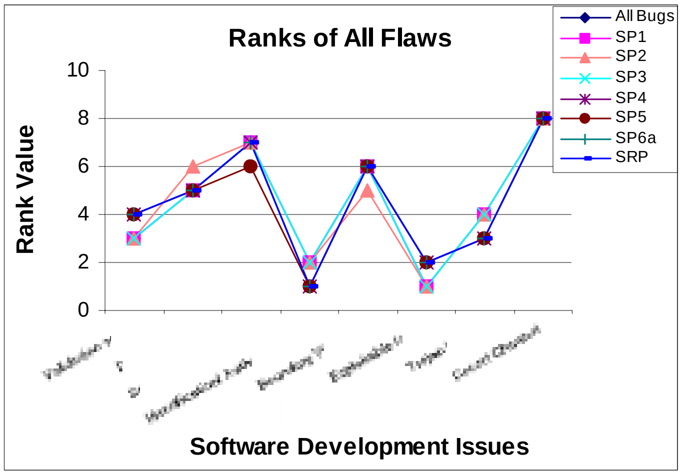


Figure 8. Ranking of software development issues.

A number of key points can be observed from the data:

- Ranks of each category in each service pack and the combined list of all flaws are similar. This indicates that the problem areas in these service packs are similar and the vulnerabilities are concentrated in certain combinations of the three dimensions. This also shows that the problem areas in security can be identified using this classification scheme.
- Since high and medium risk flaws lie in the same heavily concentrated areas, developers and testers should be more successful in eliminating these risk flaws and thus the next release would have a higher level of security.

[*]The numbers of flaws shown in each SP are relative to the previous SP. For example, from Figure 4, SP1 has 234 flaws. SP2 flaws can be obtained from this number by adding the number of flaws that appear above the X-axis in Figure 5 and subtracting the number of flaws below the x-axis. Hence SP2 has 234+2-19=217 flaws.

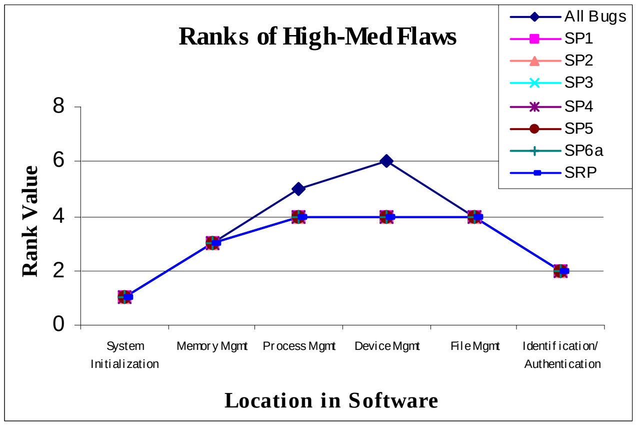


Figure 9. Ranking of locations in software counting only high and medium risk flaws.

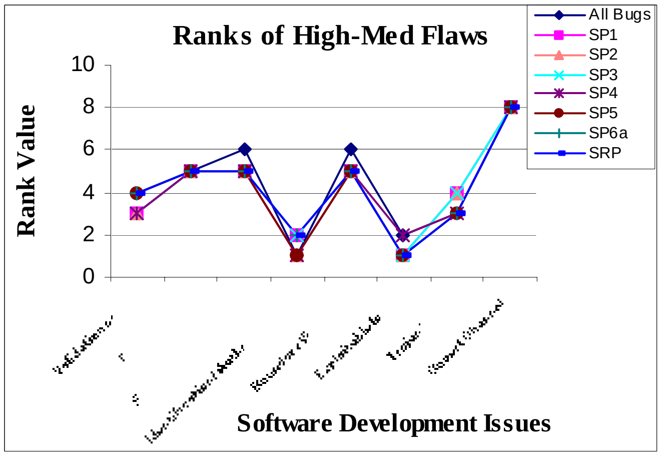


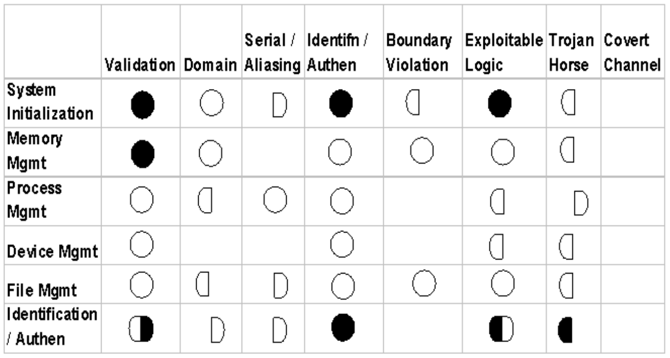
Figure 10. Ranking of software development issues counting only high and medium risk flaws.

- New security flaws found in successive service packs were present in the heavily concentrated areas showing that they may have been prevented and hence eliminated even before they could be exploited if the testing effort was concentrated in these problem areas.

Concentration of security flaws in a set of specific categories can lead to any of the following scenarios:

1. It may be easier to exploit the categories that have a larger concentration of flaws. This implies testing for these categories would lead to a more difficult to break-in and hence more secure system.
2. Test plans do not test these problem areas adequately. Using this taxonomy would lead to building more complete test plans.

This should lead to categories with fewer flaws, which may be more difficult to find and exploit. It is important to note we may not have knowledge of all flaws in the system, but using this taxonomy will help us eliminating the more common and easily exploitable bugs.



	Validation	Domain	Serial / Aliasing	Identifn / Authen	Boundary Violation	Exploitable Logic	Trojan Horse	Covert Channel
System Initialization	●	○	D	●	ᗡ	●	ᗡ	
Memory Mgmt	●	○		○	○	○	ᗡ	
Process Mgmt	○	ᗡ	○	○		ᗡ	D	
Device Mgmt	○			○		ᗡ	ᗡ	
File Mgmt	○	ᗡ	D	○	○	○	ᗡ	
Identification / Authen	◑	D	D	●		◐	◖	

Figure 11. Comparison of flaws in Windows NT(1200) and Linux (160).

For Figure 11, *Black* indicates many (≥50) Windows NT or Linux (≥8) flaws while *White* indicates fewer flaws using a ratio of 7:1 for relative number of Windows NT to Linux flaws. Left semicircle is Windows and right is Linux. Looking at all security flaws in Windows NT and Linux, we observe that most cases (31 out of 48) have similar characteristics in both systems. Emphasizing testing on only the 5 black circles identifies half (48% of Windows and 63% of Linux) of the vulnerabilities in both systems. Looking at each system independently, the black semicircles represent 60% of Windows and 68% of Linux flaws.

4. Conclusions

The work presented in this paper extends the existing research in security vulnerabilities to the perspective to help predict, prevent and eliminate security vulnerabilities in existing and new systems. The information reflects an organization's environment and is therefore more useful to accurately detect the problem areas in that environment. This information can be retained within an organization hence preventing the risk factor introduced by signing a contract with a "tiger" testing team. We have shown that concentrating testing effort in the problem areas of the organizations' environment one can develop more secure software which will even prevent future vulnerability exploits.

Thus this classification scheme would not only help the software development community in reducing maintenance costs of systems by fixing flaws in early stages of the development cycle, but also serve as a database to derive *security metrics* or *baselines for testing*. Hence, Security testing can now be established as a systematic and repeatable process to be able to collect data about the achieved level of security of the product and controlling the process to reach the desired level of security.

4.1. Further testing of the taxonomy.

Hedbom et al. [25] compared the security of Windows NT and UNIX. They found that Windows NT had slightly more rigorous security features than "standard" UNIX but the two systems display similar vulnerabilities. They reached the conclusion that there are no significant differences in the "real" level of security between these systems. This implies we would find similar regularities in the distribution of security vulnerabilities in UNIX and Windows NT.

We have compiled a list of 160 vulnerabilities found in various versions (from Versions 5.0 to 7.2) of Red Hat Linux and from Figure 4, we reach the conclusion that Windows NT and Linux vulnerabilities reflect similar regularities. This is a very striking result; however, we think the number is small to represent all the vulnerabilities present in Linux and we hope to grow the database to derive further conclusions.

We would like to use this taxonomy to classify security flaws found in different operating systems like variants of Unix, Linux and Sun Solaris to understand the relationships between the type of security flaws and the systems they are found on. If the relationships are similar, we could then conclude that the hacker community exploits a specific set of vulnerabilities and with the help of this taxonomy, these vulnerabilities could be eliminated more readily. This would also lead us to test the generality of this taxonomy.

4.2. Research in testing techniques

One research direction would be to evaluate the various testing techniques like path testing, domain testing, data flow testing to find out the vulnerabilities discovered by these traditional testing techniques and map this information to our taxonomy. This would be very useful to organizations, as they would know what technique to use after detecting the problem areas in the software.

Another direction would be to build tools and automated tests to test characteristics or conditions of software using data from the vulnerability database. For example, if Memory Management had a greater concentration of security flaws, then we would like to be able to have a tool that would perform all the memory and data structure checks and check for consistency of values stored in memory as well as boundary limits of the data structures so as to be able to remove the possible flaws in this category. Thus for regression testing, one would have to run the union of set of automated tests that checks for various problem areas detected in software.

Acknowledgements

We would like to acknowledge Mr. Jim Dingemanse and Mr. William Wall from Harris Corporation, who graciously provided access to STAT Scanner, a Vulnerability Assessment tool. We would also like to acknowledge Dr. William Arbaugh and Dr. Atif Memon of the University of Maryland and Dr. Ray Vaughn of Mississippi State University for their valuable guidance. This research was partially supported by grant CCR0086078 from the National Science Foundation to the University of Maryland.

References

[1] Michael A. Cusumano and Richard W. Selby, "Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People," The Free Press, 1995.

[2] Gula, Ron. "Broadening the Scope of Penetration Testing Techniques". July 1999.

[3] C. Pfleeger, S. Pfleeger and M. Theofanos, "A Methodology for penetration testing," Computers and Security, 8(7), 613-620, 1989.

[4] R. R. Linde, "Operating System Penetration," AFIPS National Computer Conference, pp. 361-368, 1975.

[5] E. J. McCauley and P. J. Drongowski, "The Design of A Secure Operating System," National Computer Conference, 1979.

[6] R. Kaksonen, Marko Laakso, Ari Takanen, "Vulnerability Analysis of Software through Syntax Testing," Technical Research Centre of Finland, 2000. Available online at http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/

[7] G. Fink and M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance," *ACM SIGSOFT Software Engineering Notes*, **22**(4), July 1997.

[8] J. Voas, and G. McGraw, "Software Fault Injection: Incoculating Programs Against Errors," John Wiley & Sons, Inc., 1998.

[9] Wenliang Du and A. P. Mathur, "Vulnerability Testing of Software System Using Fault Injection," Department of Computer Sciences, Purdue University; Coast TR 98-02; 1998.

[10] Eugene H. Spafford, "Extending Mutation Testing to find Environmental bugs," Software Practice and Principle, 20(2), pp. 181-189, Feb 1990.

[11] V. D. Gligor , C. S. Chandersekaran, W. Cheng, W. D. Jiang, A. Johri, G. L. Luckenbaugh and L. E Reich, "A New Security Testing Method and its Application to the Secure Xenix Kernel," Proceedings of the IEEE Symposium on Security and Privacy, pp. 40-58, 1986.

[12] B. Beizer, "Software Testing Techniques," Van Nostrand Reinhold, New York, 1990.

[13] Bisbey, R. and D. Hollingsworth, "Protection Analysis Project Final Report," Information Sciences Institute, University of Southern California, Marina Del Rey, CA, 1978.

[14] R. P. Abbott et al, "Security Analysis and Enhancements of Computer Operating Systems," Report NBSIR 76-1041, Institute for Computer Science and Technology, Natl. Bur. of Stnds, 1976.

[15] Eugene H. Spafford, "Common System Vulnerabilities," Proceedings of the Workshop on Future Directions in Computer Misuse and Anomaly Detection pp. 34-37, 1992.

[16] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," ACM Computing Surveys, Vol. 26 (3), pp. 211-254, 1994.

[17] T. Aslam, "A taxonomy of Security Faults in the Unix Operating System," M.S. Thesis, Purdue University, 1995.

[18] T. Aslam, "Use of a taxonomy of Security Faults," Technical Report 96-05, COAST Laboratory, Department of Computer Science, Purdue University, March 1996.

[19] M. Bishop, "A Taxonomy of UNIX System and Network Vulnerabilities," Technical Report CSE-95-10, Purdue University, May 1995.

[20] Wenliang Du and Aditya P. Mathur, "Categorization of Software Errors that led to Security Breaches," In Proceeding of the 21st National Information Systems Security Conference (NISSC'98), Crystal City, VA, 1998.

[21] R. A. Demillo and A. P. Mathur, "A grammar based fault classification scheme and its application to the classification of the errors of TEX," Technical Report SERC-TR-165-P, Purdue University, 1995.

[22] Brian Marick, "A survey of software fault surveys," Technical Report UIUCDCS-R-90-1651, University of Illinois at Urbana-Champaign, December 1990.

[23] Ram Chillarege, "ODC for Process Measurement, Analysis and Control," Proc. of the Fourth International Conference on Software Quality, ASQC Software Division, Oct 3-5, 1994 McLean, VA.

[24] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, Man-Yuen Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements," IEEE Transactions on Software Engineering, Vol 18, No. 11, Nov 1992.

[25] Hans Hedbom, Stefan Lindskog, Stefan Axelsson and Erland Jonsson, "A Comparison of the Security of Windows NT and UNIX", Third Nordic Workshop on Secure IT Systems, November 1998.

Appendix

The following tables present the raw data used to generate Figures 7-10.

Location	**All flaws**	**SP1**	**SP2**	**SP3**	**SP4**	**SP5**	**SP6a**	**SRP**
System Initial	505	163	153	133	105	101	98	78
Mem Mgmt	77	16	15	14	9	8	6	3
Process Mgmt	10	3	3	3	3	3	2	0
Device Mgmt	15	10	10	10	10	10	6	4
File Mgmt	31	10	7	3	3	3	2	1
Identn/Authen	215	28	25	22	17	16	17	15

Figure 12. Number of flaws found in various service packs categorized according to the locations in software.

S/W Development	**All Bugs**	**SP1**	**SP2**	**SP3**	**SP4**	**SP5**	**SP6a**	**SRP**
Validation Error	155	40	32	24	17	16	14	7
Domain Error	21	9	7	7	5	5	4	4
Serialization/ Aliasing	1	0	0	0	0	0	0	0
Identn/Authen	218	66	63	60	56	55	49	41
Boundary Violation	12	8	8	5	1	0	1	1
Exploitable logic	326	88	84	70	50	47	48	36
Trojan Horse	120	19	19	19	18	18	15	12
Covert Channel	0	0	0	0	0	0	0	0

Figure 13. Number of flaws found in various service packs categorized according to software developing issues.