RC 23538 (W0502-133) 25 February 2005
Computer Science

IBM Research Report

A Software Flaw Taxonomy: Aiming Tools At Security

Sam Weber, Paul A. Karger, Amit Paradkar
IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598, USA

Research Division
Almaden – Austin – Beijing – Delhi – Haifa – T.J. Watson – Tokyo – Zurich

This paper has been submitted to the Software Engineering for Secure Systems – Building Trustworthy Applications (SESS'05) Workshop to be held as part of 27th International Conference on Software Engineering, to be held 15-21 May 2005 in St. Louis, MO, USA.

A Software Flaw Taxonomy: Aiming Tools At Security

Sam Weber Paul A. Karger Amit Paradkar
(samweber|karger|paradkar)@watson.ibm.com

IBM Research Division
Thomas J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598

ABSTRACT

Although proposals were made three decades ago to build static analysis tools to either assist software security evaluations or to find security flaws, it is only recently that static analysis and model checking technology has reached the point where such tooling has become feasible. In order to target their technology on a rational basis, it would be useful for tool-builders to have available a taxonomy of software security flaws organizing the problem space. Unfortunately, the only existing suitable taxonomies are sadly out-of-date, and do not adequately represent security flaws that are found in modern software.

In our work, we have coalesced previous efforts to categorize security problems as well as incident reports in order to create a security flaw taxonomy. We correlate this taxonomy with available information about current high-priority security threats, and make observations regarding the results. We suggest that this taxonomy is suitable for tool developers and to outline possible areas of future research.

1. INTRODUCTION

Although efforts for building tools to detect software security flaws were started as early as 1973 [9], only recently has such tooling received significant attention. The increased awareness of the importance of security, combined with the advancement of static analysis and model checking techniques, has meant that such tooling is not only technologically feasible but of practical importance. For example, a recent United States Department of Defense report [31] states that such tooling is "important for researchers and developers to pursue, and for the government to co-sponsor."

When starting work on a software security tooling effort, we wanted to determine, in a methodological manner, what kinds of security flaws existed, their relative susceptibility to currently-known techniques, and have some idea of their relative importance. In order to do so we wanted to have a taxonomy – an ordered system that indicates natural relationships – of security flaws. Unfortunately, existing work either organized flaws by characteristics that were not useful for our purposes, or were out-of-date.

In the following paper, we first describe the previous work in this area, then describe the methodology that we followed and the resulting taxonomy. We then correlate our taxonomy with available information about prevalent system attacks, and finally describe our planned further work.

1.1 Taxonomy Types and Related work

During the previous three decades, many lists and taxonomies of security problems have been developed for various purposes. Although space limitations prevent us from providing a detailed survey of this material, we will provide an overall view of the entire field and then carefully examine the work that more closely relates to ours.

Many published lists and taxonomies, such as [5, 24, 16], concern themselves with either *vulnerabilities* or *attacks*.[1] A vulnerability, as we will use the term, is a means whereby a hostile entity can successfully violate a system's security. For example, a web application might be vulnerable to a "poisoned cookie" (a maliciously altered cookie, which the web app will trust without verification). An "attack" refers to the tool or technique with which an attacker will attempt to detect and exploit a vulnerability.

Other taxonomies and lists, such as [8, 7, 9, 23, 1], are concerned with *flaws*. A flaw is a defect in a system which can result in a security violation. Every vulnerability must be due to at least one flaw, but it is possible for a flaw not to cause any vulnerability: the flaw might be masked by another part of the system. Additionally, different flaws might result in the same vulnerability.

Attack and vulnerability taxonomies are quite useful to system administrators and testers, who want to know how their system might be attacked and need assistance to defend themselves. However, the weak association between flaws and vulnerabilities means that flaw taxonomies are more relevant to code inspection tool designers, such as ourselves. The fact that flaws might be masked, and therefore not exploitable, whereas vulnerabilities are, by definition, always exploitable, is not necessarily a drawback to flaw-detection. Developers of high-assurance systems are

[1]The terms "attacks", "vulnerabilities" and "flaws" are not used with consistent meaning in the literature. We have attempted to use terminology that is minimally confusing and which is consistent with as much prior art as possible.

concerned about the possibility of system updates revealing previously hidden flaws, especially since the results can be appear in components far removed from the update. Therefore, such developers consider it important to detect flaws, regardless of whether or not they are currently exploitable. Flaw, rather than vulnerability or attack taxonomies, are therefore what we are concentrating on.

Two flaw taxonomies produced in the 1970's were the result of the RISOS project[1] and the Protection Analysis project[9]. The former classified operating system flaws into seven categories. The latter produced four types of flaws and subdivided these general types into ten categories. Both pieces of work have received some criticism for ambiguity (as in [10]), perhaps unfairly, but their categories are generally a subset of those found in more modern work.

The Protection Analysis project deserves further mention because its intended aim was the same as ours: to produce flaw-detection tools. They sketched algorithms for such tools, but the static analysis technology available at the time was not sufficient to realize them.

Landwehr et al. [23] classified security flaws in three dimensions: genesis (how it was introduced into the system), time of introduction (when in the development cycle it originated), and location (in which component it exists). The 'genesis' classification is the most pertinent to us. This classification starts by dividing flaws into those that are "intentional" or "inadvertent", where intentional flaws are further subdivided into those that are deliberately inserted into the system ("malicious"), versus being the unintended but direct result of design features ("non-malicious"). Further refinements yield thirteen ultimate categories.

As Howard points out [16], Landwehr et al's inclusion of Trojan Horses and viruses into their taxonomy is incorrect: if there is a virus in a system, the flaw is not the virus itself, but rather whatever defect allowed the virus to be injected. From our point of view, the major problem with this taxonomy is that it is not fine enough and somewhat outdated: flaws that are now common are grouped into categories along with others that seem distinctly different or which lend themselves to different detection techniques. However, this work is well-thought-out enough to serve as our starting point.

Aslam [7] created a flaw taxonomy whose primary purpose was to serve as an organizational method for databases of flaws and to assist in static analysis efforts.

One of Aslam's criticisms of Landwehr's work, which also strongly influenced his taxonomy, was that Landwehr's classification of intentional flaws as either "malicious" or "non-malicious" was inappropriate, because it requires a decision as to the motives of the programmer, which can be hard to determine. Although we agree that for Aslam's main purpose, database classification, decisions about people's motives are problematical, we strongly disagree with this with regards to static analysis efforts. For example, Ashcraft and Engler [6] describe how they've found many security flaws in code by means of a static analysis which verified whether user-supplied integers were range-checked against *some* values, not necessarily the correct ones. Clearly this tool can be trivially defeated by a programmer intent on inserting a Trojan Horse into the system, but has proven to be effective at finding accidental security flaws. This example demonstrates that the programmer's intent, even if impossible to determine ex post facto, is a strong determinator of what flaw detection techniques are applicable.

Aslam's taxonomy reflects this decision to omit intent from consideration, and suffers thereby. The main defect in this taxonomy, however, is that its divisions are not useful for our purposes. For example, "interaction errors between functionally correct modules", misunderstood exception handling and errors created by faulty compilers are all assigned to the same category, and the taxonomy admits of no coding faults that are neither synchronization errors nor condition validation errors.

Aslam uses his taxonomy to state which flaws might be detected using static analysis. Unfortunately, this work has not aged well: both input validation errors and synchronization errors are now fruitful lines of static analysis research, even though Aslam stated that these are both "not viable."

Lastly, we must take into account Bishop and Bailey's [10] criticisms of taxonomies, especially because these criticisms could be applied to ours. Their major principle is that each item should be unambiguously assignable into exactly one taxonomy category. They point out two reasons why taxonomies fail to achieve this. First, a given flaw or vulnerability might have different causes depending upon whether one's viewpoint is that of an attacker, the flawed process, or the operating system. This criticism is not pertinent to our work: we always take the viewpoint of the flawed system, not the attacker, and we consider an operating system to be simply a low-level subcomponent of the system as a whole.

Bishop and Bailey's second general observation raises more serious issues. They observe that systems have various levels of abstraction, and that a single flaw might have different classifications, depending upon which abstraction level's viewpoint is taken. Although we agree with this observation, we disagree that this should be necessarily viewed as a problem with a taxonomy.

We maintain that the ultimate purpose of a taxonomy is that it be *useful* for its intended purposes. It is not uncommon for a poorly specified component API to cause a misunderstanding between the component's implementor and user, and this misunderstanding, in turn, to result in a security problem. As the API was not specified properly, there is necessarily an ambiguity, depending upon what level of the system one considers responsible – exactly the situation that Bishop and Bailey consider problematical. Although we could simply define away this issue by declaring that the flaw is a design error (perhaps "insufficient API specification") and therefore out-of-scope, by doing so we wouldn't be able to discuss a frequently occurring source of security issues. We do not wish to reduce the usefulness of our work because of semantic quibbles. Therefore, instead of demanding that each flaw be uniquely classified, we maintain that if a flaw can be classified under multiple categories, that this circumstance be the result of the characteristics of the flaw itself. In particular, the fact that a flaw arising out of an underspecified API can often be classified multiple ways accurately reflects the fact that such a flaw can be detected and fixed in multiple ways.

2. TAXONOMY

2.1 Methodology

In this section we discuss the methodology we used to arrive at our taxonomy.

We started by reviewing the security literature. As de-

scribed above, we selected Landwehr's work as a suitable starting point, modifying it in response to its critics. In a second pass we reviewed current descriptions of vulnerabilities and threats, and interviewed practitioners in the field, in order to ensure that newer flaws were being represented. Finally, in order to determine what characteristics were pertinent to static analysis, we studied the static analysis literature.

One major design choice was that flaw categories should be defined *positively*, not negatively. In other words, categories like "Other concurrency flaws" should be avoided. Negatively-defined categories provide little to no information to assist tool designers and thus served no purpose in our work. The only exceptions were the application-specific security flaws, which, by definition, are too application dependent to be classified in any other fashion.

Another design decision revolved around which flaws we were considering to be out-of-scope. Since the purpose of our work is to aid designers of code analysis tools, clearly configuration errors are not relevant – these errors are not visible at the code level. The question of design errors is more difficult, however, since every design decision is reflected in the code, but not necessarily in a convenient form. The decision we reached was that our taxonomy would include only those design flaws which are more easily detectable through code inspection than through other methods. For example, although improper use of cryptography is a serious security issue, it is much easier to detect this problem by inspecting the design documents than by code analysis techniques. Conversely, even though it is possible to discover underspecified APIs by reviewing documentation, in practice ambiguities are often discovered by finding that programmers have contradicted each other.

2.2 Taxonomy

Our taxonomy is in Table 1. This section describes each category and includes references to specific incidents.

As in Landwehr, the major division in our organization is between "inadvertent" and "intentional" flaws. The former essentially are bugs. The latter consists of both "malicious" flaws, which were deliberately inserted, and "non-malicious" flaws that are side-effects of features that were deliberately added to the system. (Usually, the system designers are not aware of non-malicious flaws, but we call them intentional because they were designed into the system.)

The malicious flaws consist of trapdoors (code which allows someone to gain illicit access to the system), and time and logic bombs (code which is designed to disrupt the system when certain conditions happen).

A covert channel is a means whereby two entities (normally a Trojan horse sender and a cooperating evil receiver) are not permitted to communicate according to the security policy, but are able to do so by means of side-effects of operations they are allowed to do. This can happen either by means of manipulating storage, or by modulating the time that various operations take to perform. One subtle example of a covert channel is the disk-arm elevator algorithm covert channel, first identified by Schaefer, et. al. [28]. The disk-arm elevator algorithm is a commonly implemented performance optimization in which disk requests are satisfied in the order in which the disk arm reaches the requested blocks, rather than first-come-first-served. The Trojan horse sender can influence the order of disk request completions by controlling in which direction the disk arm is moving. This control is achieved by the order in which the Trojan horse issues its own disk requests. Countermeasures to this channel are discussed by Karger and Wray [20]. These are intentional but non-malicious flaws because the illegal communication occurs not due to bugs in the system's implementation, but due to the system's design.

Intentional	Malicious	Trapdoor
		Logic/Time Bomb
	Non-malicious	Covert Channel: Storage
		Covert Channel: Timing
		Inconsistent access paths
Inadvertent	Validation Error	Addressing error
		Poor parameter value check
		Incorrect check positioning
		Identification/ Authentication Inadequate
	Abstraction Error	Object Reuse
		Exposed Internal Representation
	Asynchronous flaws	Concurrency (including TOCTTOU)
		Aliasing
	Subcomponent misuse/failure	Resource Leak
		Responsibility Misunderstanding
	Functionality Error	Error handling failure
		Other security flaw

Table 1: Flaw Taxonomy

The final category of intentional flaws is "inconsistent access paths." This category describes the flaws in which a system allows the creation of a given object or state by multiple methods, but with different security checks. These flaws most frequently are the result of system updates: security checks are modified in some but not all relevant places, or an additional mechanism to obtain a given object or state is added to the system without the designers being aware of the necessary checks that should be performed. Karger and Schell [19, Section 3.3.4.2] shows how a change in the descriptor segment from the original Multics processor (GE 645) to the later Multics processor (HIS 6180) led to an exploitable flaw. On the 645, each ring had its own descriptor segment, while on the 6180, one descriptor segment served all rings. If a segment was referenced by normal dynamic linking, then security checks were done properly. However, all ring 0 segments had a second entry in the descriptor segment that was left over from the 645 design that allowed unlimited access. This was safe on the 645, because ring 0 had its own descriptor segment, but on the 6180, a user-ring process could use one of these left-over entries by explicitly coding the correct segment number, rather than relying on the dynamic linker. Merging all the descriptor segments resulted in an exploitable inconsistent access path.

The largest subcategory of inadvertent flaws are validation errors: flaws that result from not checking input sufficiently before acting on it. We subdivide these flaws into the following categories:

Addressing Error: This category consists of those flaws whereby a reference to an area of memory or storage is not checked properly. This includes the classic buffer overflow problem. Good descriptions of the buffer overflow have been done by Aleph One [2] and Howard and LeBlanc [17, chapter 5]. Another classic example is passing a pointer into the kernel, the target of which the application does not have proper access rights. Schroeder and Saltzer [29] describe this problem and its countermeasures very well.

Poor parameter value check: Besides references to storage and memory, other parameters such as file names usually need to be validated. We separate validation of address from non-address parameters, both because addressing errors tend to have more serious security consequences, and because different model checking and static analysis techniques have been used in the literature to address the two classes. A simple example of such a poor parameter value check was found and fixed in the Multics operating system by Robert Mabee in about 1971. A Multics ring number was required to be between 0 and 63, and a user-specified ring number was checked to ensure that it was greater than or equal to the current ring. Unfortunately, that check missed the possiblity that setting a ring number to 64 would be greater than the current ring number, but when inserted into the bit field, would be truncated mod 64, and therefore be set to 0, the most privileged ring in the system.

Incorrect check positioning: Incorrect check positioning occurs when the programmer validates input parameters, but does the validation checks in the wrong order, or after the parameter has been used in some fashion. A simple example from a file system would be to check whether a file exists or not, before checking whether the user has access to search the containing directory. Returning a file does not exist error would reveal informtion that the user was not supposed to see.

Identification/Authentication Inadequate: This flaw occurs when the system does not completely check whether the caller has sufficient permissions to do the requested operation, or is the entity that it claims to be. We separate this class from the other validation errors because usually this requires access to meta-data or protocol-level information. Landwehr, et. al. [23, case U4] cites a case where "sendmail" on UNIX allowed the user to specify any file as the configuration file, even if the user did not have access to the file.

In modular systems, each module presents a more-or-less abstract model to its callers. Abstraction errors (called "domain errors" in some security literature) occur when this model breaks down. There are two kinds of breakdown: "Object reuse" and "Exposed internal representation." Almost all computer resources, such as memory, disk space, and database connections, are reused when the allocating component has finished with them. Object reuse flaws occur when an allocating component can detect that it has received a recycled object. Hebbard, et. al. [15] report an object reuse attack that allowed retrieval of user passwords in MTS, the Michigan Terminal System. Similar problems have existed in many other operating systems. In more modern software, Microsoft Word has suffered from problems where deleted text was not actually erased from the file, and document authors have been embarrassed [32] when text they thought had been deleted was exposed in publically released documents.

Exposed internal representation flaws can happen when a caller can discern or manipulate the internal details of the module's implementation. Perhaps the best known exposed internal representation flaw was found in IBM's OS/360 operating system. Developers of the system in the early 1960s had strict memory consumption budgets, and so a "clever" developer reduced his memory consumption by placing a critical control block in user-space memory, rather than in the operating system's memory. However, the user could then modify that control block to gain supervisor-level privilege over the entire system. [11, chapter 9, p. 100].

Concurrency flaws include race conditions, livelocks, deadlocks, and time-of-check to time-of-use (TOCTTOU) errors. Karger and Schell [19, section 3.3.1] show a TOCTTOU vulnerability in the Multics argument validator itself, in which a parameter was passed with an extra indirect and tally stage. Each time the CPU referenced the parameter, the indirect word was incremented by the value of the tally. Because the Multics argument validator did not consider the possibility of tally modifiers, the attacker could cause the validator to see only arguments to which the attacker had proper access, but after the validator completed its checks, the indirect pointer was set to point to something that the attacker did NOT have access.

Other asynchronous flaws are due to aliasing: different names referring to the same object. For example, if a method's input parameter shares a substructure with one of the method's output objects, then unpredictable effects might occur depending upon when the method modifies its output. It should be noted that alias effects occur even in non-threaded programs, which can surprise programmers. Perhaps the classic example of such aliasing flaws is the ability in FORTRAN to change the value of constants by passing them as parameters to a subroutine. This is discussed in some detail in Peter Neumann's RISKS Digest [13].

Resource leaks, where a system allocates some resource but fails to release it, or releases it late, are one category of subcomponent misuse flaws. Resource leaks frequently result in denial of service issues. Dynamic allocation can also lead to covert channels, because if an attacker can temporarily run the system out of some resource and then release the resource, then instead of a denial of service, you have a communications channel. This issue is discussed in [21, p. 1156].

Another important kind of subcomponent misuse is "responsibility misunderstanding". Many security problems are caused by subcomponent interfaces that are either ambiguous or misunderstood. A good example are the variety of flaws that have been found when a privileged program uses **printf** control strings on the input variables. Because **printf** can cause recursive processing of the input string, an application programmer can easily leave a vulnerability, such as a buffer overflow, by failing to understand all the

implications of **printf** [17, pp. 147–152].

Our final subcategory of inadvertent flaws are functionality errors – flaws that are directly involved in implementing the system's functionality. Error handling failures are a common problem, because system designers often do not adequately plan how exceptional conditions should be handled. Landwehr, et. al. [23, Case U6] show an example of the UNIX "su" command allowing a user to gain root privileges if there were no available file handles left. The flaw was that the designers of "su" never anticipated an error trying to open the system password file.

Finally, the system's intended functionality can be misimplemented, causing a security problem. Since these flaws are by definition system-specific, we are unable to give a more precise characterization. An example of such a flaw existed in early versions of the VAX/VMS operating system. The VAX/VMS file system included a version number in every file name. The "save" command in the text editor always created a new version of the file, so that the user did not lose work. However, if the access permissions on the file were changed to other than the default, then new versions would be created with the default permissions, rather than the potentially more restrictive permissions of the previous version of the tile. Thus, a user could inadvertantly make a private file publically readable or writeable, simply by editing the file and making a small change [12, p. 9-15].

2.3 Observations and Rationale

Previously, we have described general principles which distinguish our taxonomy from the literature and made specific comments about certain previous works. Although space limitations preclude a detailed item-by-item description of how our taxonomy compares with others, especially Landwehr's [23], we do wish to draw the reader's attention to our subcomponent misuse category.

By recognizing subcomponent misuse as a specific category, we reflect the fact that modern software is composed of multiple components, which is poorly addressed in previous work. One result is that we generalize the notion of "resource leak" to include not only operating system-defined entities but to entities such as database connections that are created and managed by subcomponents. More importantly, we enable useful identification and analysis of certain flaws.

For example, Chen and Wagner [14] created an automata corresponding to the Unix security model and used model-checking to discover that in a certain ftp server, there were errors in how interrupts were handled that enabled an attacker to cause two certain interrupts to occur and thereby obtain root access. Also, Zhang, Edwards and Jaeger [34] used a type-based static analysis tool to discover improper placement of authorization calls in the Linux kernel, which created an exploitable security hole. Using Landwehr's taxonomy, we would be forced to consider the flaws discovered in these works as "Other exploitable logic errors", [2] which is not useful, and other taxonomies would not fare better. We recognize that both of these works essentially establish correctness conditions upon subcomponent APIs, and use this information to discover flaws. We further observe that is it often the case that subcomponents are shared between systems, which allows the API model and correctness condition generation work to be shared between clients, increasing the effectiveness of a validation tool.

[2]Although the flaws found in Chen and Wagner were in interrupt code, the flaws themselves were not concurrency-related.

3. RELATION TO CURRENT CONCERNS

Recently there have been a number of publications listing or arguing that certain vulnerabilities are currently common or urgent. Both as a validation method for our taxonomy and as a topic of interest in its own right, we have attempted to correlate these vulnerabilities of interest with the underlying flaws, as represented in our taxonomy.

Some words of caution should be made about interpreting this work as defining the most important research topics. Lists of the currently most common vulnerabilities or attacks are useful because system administrators need to know what they should expect to experience. However, attack techniques are subject to rapid change. Attackers typically use only the easiest mechanism that is reasonably effective for their purposes and will quickly change approaches when that technique becomes adequately defended against. Researching only the currently experienced issues can leave one unprepared when attackers change techniques.

It is also important to ensure that deploying a particular solution to a security problem will actually fix the problem and not create a new problem. The experience of the analog cellular telephone industry with tumbling and cloning fraud is quite relevant here [18]. Originally, the cellular telephone network did not validate the credentials of a cell phone immediately, when the cell phone customer was roaming outside his or her home territory. Because of this, the perpetrator could randomly create credentials and place a few calls before they were blacklisted. (This was called "tumbling fraud".) In response to this problem, the cellular telephone industry spent a great deal of money installing pre-call-validation systems so that, when a call was made, the credentials were checked immediately. Unfortunately, the analog cell phones did not encrypt the credentials, and it was easy to pick up the credentials as they passed through the air between the phone and the base station. The criminals quickly realized that they could intercept these credentials and clone a legitimate customer's phone. The result was that within a month of deploying pre-call validation in each region, tumbling fraud went to zero, but cloning fraud became widespread in that region. The fraud continued at the same level, but now, instead of just making free calls, the perpetrators billed free calls to legitimate customers who became unhappy when they received bills for calls they didn't make. The result of putting in an expensive security counter measure but not addressing the whole problem was that the situation was made worse.

The Open Web Application Security Project (OWASP) has maintained a list of the ten most critical web application vulnerabilities ([30]). Each of the vulnerabilities on their list is described in detail, which has enabled us to associate the listed vulnerabilities with underlying flaws.

The first item on the OWASP list is "Unvalidated Input". This includes buffer overflows, SQL insertion, format string attacks, cookie poisoning and hidden field manipulation. Besides the obvious category "Addressing error", this vulnerability is also caused by "Poor parameter value check", "exposed internal representation", and "Responsibility misunderstanding" flaws. For instance, format string attacks are not caused by unvalidated input (after all, in almost all cases character strings that include '`%s`' substrings

Flaw Categories	OWASP Item	
Trapdoor		
Logic/Time Bomb		
Storage Covert Channel		
Timing Covert Channel		
Inconsistent access paths		
Addressing error	⑤	1
Poor parameter value check		1 4
Incorrect check positioning		
Authentication Inadequate	② ③	
Object Reuse		
Exposed Internal Representation		1
Concurrency		
Aliasing		
Resource Leak		9
Responsibility Misunderstanding	⑥	1
Error handling failure	⑦	4
Other Functionality Error		8

Out of scope: ⑩ 2 4 8 9

Legend: ⊕ OWASP item # completely in category
OWASP item # partially in category

Table 2: Taxonomy vs OWASP Top Ten

should be legal), but are caused by programmers invoking library routines without realizing that routines like `fprintf` reparse their arguments – a "responsibility misunderstanding." Other items correspond to only one of our flaw categories, while others are either completely or partially out-of-scope, being either design or configuration errors.

In Table 2 we list our flaw categories along with which OWASP vulnerabilities completely or partially map to them. In order to prevent the vulnerabilities that map to only one category from being visually overwhelmed by the others, we encircle the former.

We observe that none of the OWASP items correspond to intentional flaws. There are several possible causes for this. Many current exploits can result in total takeover of the target servers, and it might be that companies consider that insiders could do no more damage than outside intruders, and thus defending against the latter should take priority. Other possibilities are that companies do not wish to disclose any incidents of insider fraud, or that companies have very little means of detecting such problems and therefore have no idea of whether they are affected.

A notorious difficulty with writing and debugging web applications is dealing with concurrency: web applications are intrinsically highly threaded and often distributed. Programmers usually have little experience with locking and threading, and we see no a priori reason why these problems should not result in security violations. It is surprising that they are not represented in a list of major web application security vulnerabilities. We speculate that deterministic attacks are currently easier to perform and more predicable than ones that rely on creating race conditions. However, we predict that as existing vulnerabilities are fixed, these attacks will increase in frequency.

The SANS organization creates and maintains a list of the twenty most critical internet security vulnerabilities ([27]). Unfortunately, this list was not able to be used by us in the same manner as the OWASP list. The SANS list consists of vulnerabilities like "Web Browsers" and "Databases", and it was not possible to determine specific flaw categories underlying such vulnerabilities.

Anderson, Irvine and Schell argue in [4] that subversion (that is, insertion of trapdoors, time bombs and logic bombs into systems) is a pressing concern, citing among other incidents, a case where a logic bomb was deliberately inserted into natural gas pipeline equipment resulting in "the most monumental non-nuclear explosion and fire ever seen from space." The logic bomb was planted by Gus Weiss of the CIA [33] and described in more detail by Reed [26, pp. 266-270]. Other work that makes similar arguments includes [3, 22, 25].

4. CONCLUSIONS AND FUTURE WORK

In light of the current interest in tools that detect software security problems, we've presented a security flaw taxonomy oriented towards such efforts. Additionally, we have reviewed the literature in the area, and showed how this taxonomy relates to available information on currently experienced attacks.

We are presently using our taxonomy to determine the relative difficulty of applying existing static and dynamic analysis techniques to various flaw categories. To be precise, there are a number of techniques which can be used to approach security problems, each with specific advantages and disadvantages. We believe that our taxonomy will allow us to decide which techniques are suitable for each kind of flaw, and illuminate which research directions are feasible.

5. ACKNOWLEDGMENTS

The authors would like to acknowledge the kind assistance of the experts we interviewed in the course of this work: Klaus Julisch, Sue McIntosh, Dave Safford, Doug Schales, Wietse Venema, and the Security department of IBM's Thomas J. Watson Research Center as a whole. We also would like to thank Elaine Palmer for her support of this project and her ongoing feedback. Other members of this project who have helped on this taxonomy and who are working on the testing tools and strategy that will use the taxonomy are Matthew Kaplan and David Toll. We also thank the people who have reviewed and commented on this paper, including J. R. Rao and others.....

6. REFERENCES

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tukubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. NBSIR 76-1041, The RISOS Project, Lawrence Livermore Laboratory, Livermore, CA, USA, Apr. 1976. Published by the Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC, USA.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 8 November 1996. URL: http://www.phrack.org/show.php?p=49&a=14.

[3] E. A. Anderson. Demonstration of the subversion threat: Facing a critical responsibility in the defense of cyberspace. Master's thesis, Naval Postgraduate School, Mar. 2002.

[4] E. A. Anderson, C. E. Irvine, and R. R. Schell. Subversion as a threat in information warfare. *Journal of Information Warfare*, 3:51 – 64, 2004.

[5] J. P. Anderson. Computer security technology planning study. Technical Report ESD–TR–73–51, Vols. I and II, James P. Anderson and Co., Fort Washington, PA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, Oct. 1972. URL: http://csrc.nist.gov/publications/history/ande72.pdf.

[6] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes, May 2002. In IEEE Symposium on Security and Privacy, Oakland, California.

[7] T. Aslam. A taxonomy of security faults in the UNIX operating system. Master's thesis, Purdue University, Aug. 1995.

[8] T. Aslam, I. Krsul, and E. H. Spafford. Use of a taxonomy of security faults. In *Proc. 19th NIST-NCSC National Information Systems Security Conference*, pages 551–560, 1996.

[9] R. Bisbey and D. Hollingworth. Protection analysis: Final report. Technical Report ISI/SR-78-13, Information Sciences Institute, University of Southern California, Marina del Rey, CA, May 1978. URL: http://csrc.nist.gov/publications/history/bisb78.pdf.

[10] M. Bishop and D. Bailey. A critical analysis of vulnerability taxonomies. Technical Report CSE-96-11, Department of Computer Science at the University of California at Davis, Sept. 1996.

[11] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, 1975.

[12] M. G. Carter, S. B. Lipner, and P. A. Karger. Protecting data & information: A workshop in computer & data security. Technical Report EY-AX00080-SM-001, Digital Equipment Corporation, Maynard, MA, 1982.

[13] Changeable constants. *The RISKS Digest: Forum On Risks To The Public In Computers And Related Systems*, 16(38), 2 September 1994. URL: http://catless.ncl.ac.uk/Risks/16.38.html.

[14] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, Nov. 18–22, 2002.

[15] B. Hebbard, P. Grosso, T. Baldridge, C. Chan, D. Fishman, P. Coshgarian, T. Hilton, J. Hoshen, K. Hoult, G. Huntley, M. Stolarchuk, and L. Warner. A penetration analysis of the michigan terminal system. *Operating Systems Review*, 14(1):7–20, Jan. 1980.

[16] J. D. Howard. *An Analysis Of Security Incidents On The Internet 1989 - 1995*. PhD thesis, Carnegie Mellon University, Apr. 1997.

[17] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, third edition, 2003.

[18] P. A. Karger. Network security: Threats and solutions. In *The Internet and Telecommunications: Architectures, Technologies, and Business Developments*, pages 127–133. International Engineering Consortium, Chicago, IL, 1998.

[19] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical Report ESD–TR–74–193, Vol. II, HQ Electronic Systems Division, Hanscom AFB, MA, USA, June 1974. URL: http://csrc.nist.gov/publications/history/karg74.pdf.

[20] P. A. Karger and J. C. Wray. Storage channels in disk arm optimization. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–61, Oakland, CA, 20–22 May 1991.

[21] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991.

[22] L. Lack. Using the bootstrap to build an adaptable and compact subversion artifice. Master's thesis, Naval Postgraduate School, June 2003.

[23] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computer Surveys*, 26(3):211–254, Sept. 1994.

[24] U. Lindquist and E. Jonsson. How to systematically classify computer security intrusions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 154–163, 1997.

[25] J. Murray. An exfiltration subversion demonstration. Master's thesis, Naval Postgraduate School, June 2003.

[26] T. Reed. *At the Abyss : An Insider's History of the Cold War*. Presidio Press, New York, 2004.

[27] SANS. The twenty most critical internet security vulnerabilities (version 5.0). web publication: http://www.sans.org/top20, Oct. 2004.

[28] M. Schaefer, B. Gold, R. Linde, and J. Scheid. Program confinement in KVM/370. In *Proceedings of the 1977 ACM Annual Conference*, pages 404–410, Seattle, WA, 16–19 Oct. 1977.

[29] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, Mar. 1972.

[30] The Open Web Application Security Project. The ten most critical web application security vulnerabilities. Web publication: www.owasp.org, Jan. 2004.

[31] United States Department of Defense. Software assurance: mitigating software risks in the dod it and national security systems. Technical report, DoD OASD(NII) forwarded to Committee on National Security Systems (CNSS), Oct. 2004.

[32] M. Ward. The hidden dangers of documents: Dot.life - how technology changes us. *BBC News - World Edition*, 18 August 2003. URL: http://news.bbc.co.uk/2/hi/technology/3154479.stm.

[33] G. Weiss. Duping the soviets: The farewell dossier. *Studies in Intelligence*, 39(5), 1996. URL: http://www.odci.gov/csi/studies/96unclass/farewell.htm.

[34] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the USENIX Security Symposium*, Aug. 2002.