A Taxonomy of UNIX System and Network Vulnerabilities

Matt Bishop

CSE-95-10

May 1995

A Taxonomy of UNIX System and Network Vulnerabilities

Matt Bishop

Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

1. Introduction

Ambrose Bierce defined “history” as “a record of mistakes made in the past, so we shall know when we make them again.” Although sardonic, his definition describes the state of affairs of computer system vulnerabilities.

A “vulnerable state” is “any state which enables a user to read information without authorization, modify information without authorization, or grant or deny an entity access to a resource without authorization.” “Exploiting a vulnerability” means that a system is in a vulnerable state and a user (called an *attacker*) reads or writes the information without authorization, or grants or denies service to another without authorization. In both these definitions, “without authorization” means “in violation of the system’s security policy.” A “vulnerability” (also called a “flaw” or a “hole”) is the property of the system, its attendant software and/or hardware, or its administrative procedures, that cause it to enter a vulnerable state.

That attackers can, and do, exploit system vulnerabilities is widely known; indeed, detailed descriptions of how to find vulnerable states have appeared in various periodicals such as *PHRACK* and *2600*, and on the USENET [1]. All these discussions essentially show how to probe a system for clues that indicate the system is running software known to be vulnerable, or that it is being managed in such a way as to allow an attacker to exploit a vulnerability. The issue of how to find the underlying vulnerabilities in the first place is finessed.

Research projects have addressed this issue. The seminal flaw hypothesis methodology [8] used people who knew the system being attacked to find specific vulnerabilities and then hypothesize more vulnerabilities from the known one. It introduced the notion of “classes of vulnerabilities” in which these were grouped by how well one flaw suggested another. Work at the Information Sciences Institute [2] and Lawrence Livermore National Laboratories [9] extended this notion by postulating broad categories of flaws and grouping known vulnerabilities into these categories. Independently, several penetration studies of systems were made [5][6][7].

In the late 1980s interest in the study of vulnerabilities became more prominent for two reasons. The first was the increase in the number of systems connected to the Internet, and the attendant increase in exploiting system vulnerabilities. System administrators often found themselves attacked before they even knew of a vulnerability's existence, and the strictures preventing the discussion of existing (and unfixed) security holes began to break down. The second was the advent of intrusion detection systems, first proposed in [12] to detect statistically unexpected patterns of behavior. Investigators used a database of rules to describe known attacks against systems, and the intrusion detection mechanisms would look for events matching those rules. This helped drive the collection of system vulnerabilities and their integration into a database; and since it would be desirable to have the intrusion detection system find new attacks, the interest in classifying vulnerabilities grew, with an eye towards being able to detect new ones. Work by [3] and [11] proposed taxonomies.

In this paper, we shall build on prior work to present another taxonomy, and argue that this classification scheme highlights characteristics of the vulnerabilities it classifies in a more useful way than other work. We shall then examine vulnerabilities in the UNIX operating system, its system and ancillary software, and classify the security-related problems along several axes, after which we shall examine the earlier work to see if this taxonomy holds for other systems. The unique contribution of this work is an analysis of how to use the Protection Analysis work to improve security of existing systems, and how to write programs with minimal exploitable security flaws. This contrasts the work to [4], which argued that a preventative approach using formal methods to design secure systems is appropriate. We emphatically agree; however, as nonsecure systems continue to be used, our work is presented with the hope it will guide maintainers and software implementers to improve the security of these flawed systems and software.

The perennial question of revealing sensitive information arises; by describing security vulnerabilities, we present information that an attacker could use to penetrate a computer system; so such vulnerabilities are rarely discussed in the literature unless they are quite old. Showing that our model is useful requires us to discuss existing vulnerabilities. However, we view the debate over the wisdom of discussing vulnerabilities as tangential to the thrust of this *particular* paper, so to emphasize the issue of relevance to our work over the dangers of revealing security vulnerabilities, we have chosen to discuss those which are "widely known," that criterion being met if the vulnerability has been discussed in a nonstandard publication (such as *PHRACK* or *2600*), a nonrestricted electronic mailing list such as the *RISKS* or *BUGTRAQ* mailing lists, or a USENET newsgroup such as *alt.security* or *comp.security.unix*.

In the next section, we present what we need from a taxonomy of computer system vulnerabilities; we analyze the earlier taxonomies and show why they are inadequate. Then, building upon them, we design a classification scheme that meets our needs. In the third section, we analyze several well-known vulnerabilities in the UNIX system, and in implementations of network protocols, and show how they would be classified, both in our scheme and in the earlier ones. In the fourth section, we show how to use the taxonomy to detect the vulnerabilities and their exploitation, as well as how to eliminate them when possible. We close with suggestions for future work.

2. The Previous Models

In this section, we state our goal in designing a model, then briefly present previous models of system vulnerabilities, and see how well they meet our purpose. We extend and modify them as necessary.

2.1. The Goal of this Study

The goals of this work are:

1. To describe the vulnerabilities in a form useful to intrusion detection mechanisms;
2. To present techniques for finding these vulnerabilities; and
3. To present techniques to inhibit or eliminate exploitation of those vulnerabilities.

The first goal requires a bit of analysis. An intrusion detection mechanism acts by analyzing a sequence of commands and related characteristics (such as timing) to determine if an entity is exploiting, or attempting to exploit, a vulnerability. With most systems, this requires the use of a rule base and pattern matching keyed to sequences of events that are known to cause compromise.

Consider a broader view. Every successful attack exploits a vulnerability. Within the attack, lots of events occur that “set up” the exploitation but do not depend upon the particular vulnerability. The actual exploitation (that is, the violation of the security policy) is typically only a few events. For example, consider the “mkdir” bug in Version 7 UNIX [13] (see Appendix A). The vulnerability is *not* the ability to change the ownership of a system file; the vulnerability is the race condition (specifically, the lack of atomicity in the creation/change owner steps) that allows one to change ownership of a system file, as an exploitation of that vulnerability. Thus, *any* such sequence of instructions is suspect. So, rather than simply looking for the sequence “make directory node; delete directory node; link file to the name of the deleted directory node; change ownership of directory node,” one should instead look for sequences of the form “instantiate *name*; delete node corresponding to *name*; reinstantiate *name*; modify characteristics of node with that *name*” to discover other attacks of this nature.

Were this done, a very recent attack would have been foiled. The “xterm” attack (see Appendix A) uses the more general sequence: check to be sure the real user can access the file, open it for writing. The check for access is an instantiation in that it sets the process’ idea of characteristics of the node corresponding to *name*; the open is a modification of the characteristics of the node in that it binds the node to an internal identifier independent of the node’s name. So, the attack takes the same form as the “mkdir” vulnerability: instantiate *name*; delete node corresponding to *name*; reinstantiate *name*; modify characteristics of node with that *name*.” This provides a new way to exploit the old vulnerability. (Race conditions are discussed in more depth later.)

The point is that a model should highlight the underlying vulnerability and not its exploitation. Models used for intrusion detection focus on the exploitations using known sequences of system calls (or applications) rather than on the underlying vulnerability. Focussing on the latter presents an opportunity for much more general rules and patterns and, hence, much more effective detection of exploitations.

This leads to the second goal. The two race conditions above are the same given the appropriate view of the system. What is needed now is to characterize “instantiation” and “modification” in terms of system calls, and then all race conditions involving access to files will be handled. Similar characterizations are needed for process interaction, and other types of interactions. When those characterizations are complete, the issue of vulnerabilities through race conditions will be well characterized and can be checked for automatically.

The third goal follows from this. Techniques in program analysis, intrusion detection, and other areas of security can analyze programs or log files to determine when an attack involving a race condition could arise; for example, this technique has been used to find security problems with security packages! Thus, it is an effective analysis technique, and can be used to find new security holes in programs and systems that will allow attackers to exploit vulnerabilities.

1. Improper protection (initialization and enforcement)
   - 1a. improper choice of initial protection domain – “incorrect initial assignment of security or integrity level at system initialization or generation; a security critical function manipulating critical data directly accessible to the user”;
   - 1b. improper isolation of implementation detail – allowing users to bypass operating system controls and write to absolute input/output addresses; direct manipulation of a “hidden” data structure such as a directory file being written to as if it were a regular file; drawing inferences from paging activity
   - 1c. improper change – the “time-of-check to time-of-use” flaw; changing a parameter unexpectedly;
   - 1d. improper naming – allowing two different objects to have the same name, resulting in confusion over which is referenced;
   - 1e. improper deallocation or deletion – leaving old data in memory deallocated by one process and reallocated to another process, enabling the second process to access the information used by the first; failing to end a session properly
2. Improper validation – not checking critical conditions and parameters, leading to a process’ addressing memory not in its memory space by referencing through an out-of-bounds pointer value; allowing type clashes; overflows
3. Improper synchronization;
   - 3a. improper indivisibility – interrupting atomic operations (*e.g.* locking); cache inconsistency
   - 3b. improper sequencing – allowing actions in an incorrect order (*e.g.* reading during writing)
4. Improper choice of operand or operation – using unfair scheduling algorithms that block certain processes or users from running; using the wrong function or wrong arguments.

Table 1. The categories of flaws in the Protection Analysis study, as presented in [4]. The examples are taken from [4].

2.2. Analysis of Previous Models

We now consider previous models of security vulnerabilities. There are four main works: the Information Sciences Institute study [2], the RISOS project [9], a study from the Naval Research Laboratory [3], and a discussion of vulnerabilities for intrusion detection [11].

The Protection Analysis project, undertaken at the Information Sciences Institute, had as its goal the derivation or discovery of patterns of errors that would enable the automatic detection of security flaws. The classification scheme that was developed is summarized in Table 1.

Note that these categories are not mutually exclusive. In particular, categories 2 and 4 may often overlap, because lack of proper validation allows the passing of an improper operand. Also, in many cases, the “time-of-check to time-of-use” flaw (category 1c) also implies that an operation which should have been atomic was not atomic, and so could be argued to fall under category 3a. This is more properly an example of problems at different levels of the abstraction hierarchy, introduced during the refinement of that hierarchy.

1. Incomplete parameter validation – failing to check that a parameter used as an array index is in the range of the array;
2. Inconsistent parameter validation – if a routine allowing shared access to files accepts blanks in a file name, but no other file manipulation routine (such as a routine to revoke shared access) will accept them;
3. Implicit sharing of privileged/confidential data – sending information by modulating the load average of the system;
4. Asynchronous validation/Inadequate serialization – checking a file for access permission and opening it non-atomically, thereby allowing another process to change the binding of the name to the data between the check and the open;
5. Inadequate identification/authentication/authorization – running a system program identified only by name, and having a different program with the same name executed;
6. Violable prohibition/limit – being able to manipulate data outside one's protection domain; and
7. Exploitable logic error – preventing a program from opening a critical file, causing the program to execute an error routine that gives the user unauthorized rights.

Table 2. The categories of flaws in the RISOS study, as presented in [9]. The examples are taken from [9].

For example, how are files represented at the various levels of the hierarchy? At the highest level, users would use file names; at an intermediate level, programs would use integers (file handles or descriptors). System calls would take names as arguments if they manipulated the file system, and file handles if they manipulated the contents of a file. But at the lowest level, files are represented to the system as ordered pairs (on the UNIX system, the components would be the device number and the inode number). The high-level abstractions would be translated to pairs, and the kernel would manipulate the pairs. Unless file descriptors are bound to the pairs rather than to names, category 1c flaws (the "time-of-check to time-of-use" flaws) are immediately introduced, since the name refers to one pair, and the file handle may well refer to another pair. These flaws are a direct result of the way the notion of "file name" has been refined.

We should note that category 1a flaws are those which exist when the system is started; all others relate to nonsecure system transformations, specifically failures to enforce the given security policy properly. Whether or not the system can enforce such policies is beside the point, which is that those mechanisms do not act appropriately.

The RISOS Project, undertaken at Lawrence Livermore National Laboratories, developed seven classes of vulnerabilities (see Table 2). Although their focus was on the operating system software and not on supporting programs and applications, their list is quite applicable to the latter two. As with the Protection Analysis study, these problems may occur anywhere in the design hierarchy. However the Protection Analysis model includes initial state (category 1a), whereas all the RISOS categories speak to enforcement (although it could be argued that the “protection domain” referred to in category 6 means that domain specified by the security policy, in which case RISOS category 6 includes initial state). With that single exception, the two schemes overlap (see Table 3).

The RISOS study focuses on the exploitation of the flaws rather than the nature of the condition which causes them. For example, consider class 3 (implicit sharing of privileged/confidential data). This category covers all methods for exploiting covert channels (both timing and storage). Modulating the load average is one such channel; sending information through the creation and deletion of a file is another. The RISOS study lumps these very different methods into one class, whereas the Protection Analysis study separates the two. The storage channel would fall into PA category 1c (improper change), since it involves monitoring changes to another process’ files in a shared area; the timing channel would fall into category 1b (improper isolation of implementation detail), since the timing information is a detail of implementation that can be monitored. Other methods of exploiting covert channels could fall into PA categories 3b (improper sequencing) and 4 (improper choice of operator or operand) as well, if the method of signalling involved flaws in those categories.

Landwehr’s study [3] is interesting because it attempts to classify vulnerabilities according to genesis, time of introduction, and location. The “genesis” category has two major subcategories. Those flaws which were “inadvertent” fall into one of the above classes ([3] uses a modification

Table 3. The relationship between the categories of the PA study and the RISOS study.

RISOS category	PA categories
1&2	2
3	1b, 1c, 3b, 4
4	1c, 3a, 3b
5	1c, 1d
6	1b, 2, 4
7	any

of the RISOS classification); those which are “intentional” are either “malicious” (*e.g.*, Trojan horses, trapdoors, logic or time bombs) or “non-malicious” (*e.g.*, covert channels, because they are an artifact of the nature of many systems and cannot be entirely eliminated). The time of introduction of a flaw is either during development (either in the requirements/specification/design phase, the source code, or the object code), during maintenance, or during operation. The location refers to what has the flaw: the hardware or the software; the latter is broken into applications, support (privileged and unprivileged utilities), and the operating system proper. Let us consider these three axes independently.

The basic problem with the first axis, that of classifying flaws by genesis, is it confuses the vulnerability with the exploitation of the vulnerability. Specifically, a Trojan horse may use the inadequacy of identification when it violates a security policy, and exploits a vulnerability to give the user confidential information. The distinction between “intentional” and “inadvertent” is more the “exploitation of the vulnerability” and the “vulnerability” itself. The extra layer of “intent” detracts from the identification of the specific nature of the flaw.

The time of introduction classification scheme encompasses procedural vulnerabilities as well as those in the software and hardware. Landwehr divides the lifetime of software into three phases, the development phase, the maintenance phase, and the operation phase. In software engineering terms, “maintenance” is defined as any “work done to change the system after it is in operation” [16]. But what precisely does “in operation” mean? When software is in the beta test stage, it may be “in operation” at sites other than the developer’s site, yet that is considered “development.” Further, Landwehr’s example of an operational flaw is the infection of a program with a virus. From our point of view, the infection is not a flaw but the exploitation of a flaw (which is that the protections are incorrectly set). So this classification needs to be made more explicit.

The third axis discusses where the vulnerability occurs: in hardware, software, and if the latter, what kind of software. For example, the two race conditions described earlier occur in system software. However, they could equally well occur in user programs; of course, since those are not privileged, the vulnerability poses a threat to the user owning those programs only. Thus, for our purposes, this category is not particularly useful; as we shall see, however, a slight modification makes it very useful.

Spafford’s characterization of security vulnerabilities [11] identifies operational (administrative) flaws, design flaws, and faults as the three main categories of flaws. All of these are subsumed by the classes discussed above, and require no further discussion.

2.3. The Synthesis of A Framework

We now build upon the analysis in the previous section to develop axes of classification for vulnerabilities. Because we are focussing on the underlying cause of the flaw, rather than a description of the flaw, we shall use the PA categories to classify security problems. As was argued above, that classification describes the nature of the flaw in more detail than the RISOS classes do. In some sense, this is an artifact of the origins of the research; the RISOS project very specifically looked at operation systems and not arbitrary programs, whereas the PA study emphasized both programs and operating systems.

What other metrics would aid our analysis? Our goal is to identify vulnerabilities. In that sense, how or why they were introduced is irrelevant. Nevertheless, the collection of other information may help software and system designers determine where to put resources to look for flaws, where to look to learn about flaws, and the level of expertise needed to take advantage of the flaws. So, we shall consider other axes.

The problem with Landwehr’s time of introduction classification scheme is that the distinction between “during development”, “during maintenance,” and “during operation” seems somewhat artificial, especially since the requirements of a system often change once the system is installed. For example, many non-secure systems have had security enhancements added to them when the need arose. Is this maintenance or development? For our purposes, we adopt the following definitions. Suppose we have some software *plugh*, at version *xyzzy*. If a security vulnerability exists in all versions of *plugh* up to version *xyzzy*, it is in the class “during development.” (If we can further identify the flaw as being introduced in design or in implementation, we shall do so; but this is not always obvious.) If there is a version *glorkz* before which no version of *plugh* had the vulnerability, but the vulnerability exists in versions of *plugh* from *glorkz* to *xyzzy*, it will be in the class “during maintenance.” If the vulnerability depends only upon the operation of the entity, then we shall put it in the class “during operation.” We should note that from the intrusion detection point of view, the first two classes may be considered together, as a “system” problem, and the third is a “procedural” problem.

Landwehr’s third axis suggests an alternate characterization of location. Where the flaw

	Effect domain: in all cases, the user domain(s) which the vulnerability can effect must be given. The numbers refer to what else is affected: 1 nothing 2 network sessions (datagram or connection) 3 (physical) hardware 4 network sessions and (physical) hardware

Table 4. The exploitation domain and effect domain axes.

occurs is not so important as whom it affects and who can exploit it. For example, if one were to plant a Trojan horse, it could access the protection domain of anyone who executed it. In some cases, the effect depends upon physical access to the hardware (ejecting a CD ROM or floppy, for example, or playing offensive matter over a speaker) or on the victim being on a network (for example, sending sensitive data over a non-secure network puts the data at risk). So, the effects axis has one class per user protection domain, and two subclasses which capture the notions of a “physical” protection domain and a “network” protection domain. Similarly, consider what privileges are needed to exploit the vulnerability. It may require access to a particular user’s protection domain; similarly, it may also require access to the physical hardware (such as replacing EPROMS to change a password) or to a network (to launch an IP spoof). These categories identify the privileges required for exploitation. Table 4 summarizes these.

Note that these are similar to, but different than, Landwehr’s categories, for he focuses on the type of software in which the flaw occurs, whereas we focus on the effect of the vulnerability. Specifically, he does not distinguish between a privileged protection domain and multiple protection domains. On many systems, users can create programs which allow others to access their protection domain; so, the distinction is useful.

Three other potentially useful classes of categories have been identified. The first, the method of exploitation, describes whether the flaw can be exploited using a program, a high-level user command language, or a configuration file. The last category is subsumed by the time of introduction axis; the issue of which language is required to exploit the flaw is clearly intended as a measure of difficulty of exploiting the flaw. However, with the introduction of “cracking scripts” which enable novices to break into systems using very sophisticated techniques, it is not clear that the distinction between the command interface language and a programming language indicates anything more than convenience. Further, from the intrusion detection point of view, the effects of the program exploiting the vulnerability (such as its trace in the system logs) is what matters; and those will be the same regardless of the language used to create the exploitation. So the method of exploitation axis is redundant.

The second proposed axis, the difficulty of exploitation, is irrelevant to the issue of intrusion detection when the method of introducing the exploiting script is considered (*e.g.* cut and paste, or trial and error). Consider race conditions. While these may require trial and error, a good program can drastically reduce the number of tries needed to succeed; further, a user may use cut and paste to probe the race conditions. But the question of number of components required to exploit the vulnerability is a good metric, because it indicates the number of programs' audit records must be analyzed to discover the exploitation – and this is what intrusion detection mechanisms must do. So, another axis we shall use is the minimum number of components needed to exploit the vulnerability.

The third proposed classification scheme, the identification of the flaw, is relevant to intrusion detection because it gives compilers of misuse databases information about where to look for vulnerabilities – and where people have not been looking. Example classes in this category include software (especially for improper settings of initial protection domains), postings to bulletin boards, papers, articles, and so forth.

To summarize, the taxonomy we use has six axes, and every vulnerability is classified on each axis. The first axis is the *nature* of the flaw, and we use the Protection Analysis categories; the second axis is the *time of introduction*, and we use the (modified) classes of Landwehr. Third is the *exploitation domain* of the vulnerability and fourth is the *effect domain*; for these, we use the classes outlined above. The fifth axis is the *minimum number* of components needed to exploit the vulnerability. The sixth axis is the *source* of the identification of the vulnerability.

2.4. The Impact of the Security Policy Upon Vulnerability Classification

The issue of what constitutes a vulnerability is not discussed in any of the above studies, possibly because it was considered self-evident. If a user can access a file which is read-protected, a violation of security has occurred; the characteristic allowing the viewing is a vulnerability. But consider the following situation: suppose an educational site has the policy that no user may look at another user's files. If Tom looks at Nancy's file "hw1," sees it is not read-protected, and then views the file, has a breach of security occurred? In our model, the answer is "yes;" the specific vulnerability is that of “improper protection/improper choice of initial protection domain” (class 1a).

This is a clear case. But consider now Thompson’s compiler bug (see Appendix A). Is this exploitation of a vulnerability? If so, which one?

Let us analyze what happened. First, the “attack” is the modification of the compiler, and the installation of the corrupt compiler and rigged *login* program. The security policy, while unstated, appears to be that accessing the system requires your (chosen or given) password; nothing is said about the ability to change the compiler. Assuming Thompson was a member of the system staff (a reasonable assumption), his modifying the compiler did not produce a state in which any user could violate confidentiality, integrity, or access to services. Hence, this step seems not to involve a vulnerability, not to be an attack.

The second step, recompiling and reinstalling the rigged *login* program, also seems to be allowed under the assumption that Thompson is a member of the system staff. Hence it too is not an attack, nor does it exploit a vulnerability. It does however create a vulnerability, namely the ability of an entity to access a resource (an account) without authorization. So, at this point the system is in a vulnerable state.

The third step, which would be a user using the rigged login program and fixed password to access the system, involves exploiting the vulnerability installed earlier. It is an “attack” as we use the term.

As a socratic exercise, assume Thompson were not authorized to modify the compiler or to install the *login* program. In both cases, his ability to do so would indicate that he had privileges not allowed under the security policy, which shows the existence of improperly set protections, or a vulnerability of “improper protection/improper choice of initial protection domain.”

The key point is that the security policy controls what is, and is not, a vulnerability, and any analysis of system vulnerabilities must be made with this in mind. In what follows, when those aspects of the policy involved in the attack are clear, we shall not specify them; but (as with the Thompson example) when they could change whether what we are discussing is the exploitation of a “vulnerability,” we shall be explicit.

In the next section, we discuss some classic security flaws in the UNIX system, and classify them according to these axes.

3. Example Classification Of Some UNIX Flaws

To show that the model is useful, we classify some of the better-known UNIX security flaws along these lines. See Appendix A for a technical description of the programs and attacks involved. We should note that although we refer to the vulnerabilities by the attacks which exploit them, the item of interest in each case is the vulnerability itself and not the attack. Thus, we have avoided classifying the attacks.

3.1. The *mkdir* Flaw

This flaw springs from a race condition, specifically the non-atomicity of the creation and the change of ownership of the directory node. It is in essence two processes writing to the same file (taking the view that changing the ownership of a file is a "write" and changing the binding of a name to a file is a "write"). Hence, the vulnerability it exhibits clearly falls into the class of "improper synchronization/sequencing" (flaw class 3b). As *mkdir* allows you to change system files, including the password file, it allows access to all protection domains on the system and the effect domain is therefore that of any user; however, it is setuid to *root*, and only *root* can execute the system calls that cause the race. Therefore the exploitation n domain is that of *root*. In terms of time of introduction, this was most likely done during development; it is definitely a software problem and not a procedural one.

Note at the abstract level, this is an example of what should be an atomic operation being divisible; specifically, the creation should be as the owner of the directory. However, at the implementation level, the flaw consists of problems in sequencing operations, as discussed above. This provides an example of where the abstraction flaw falls into one class and the implementation flaw into another, closely related class.

Vulnerability class: 3b, improper synchronization/sequencing
Time of Introduction: 1, during development
Exploitation Domain involved: 1, *root* protection domains
Effect Domain: 1, any protection domain
Minimum number: 2, the *mkdir* process and another process to delete the directory file and link the password file to the name
Source: unknown, but described in [13]

3.2. The *xterm* Flaw

This is the legendary "time-of-check to time-of-use" flaw; specifically the binding of the log file name changes between the time of the access check and the opening for writing. Hence, the vulnerability it exhibits is in the class of “improper change” (flaw class 1c). As *xterm* can be used to alter a protected file, its effect domain is that of any user; also, the user with the effective UID of *xterm* can execute the system calls that cause the race. In terms of time of introduction, this was clearly done during development; it is definitely a software problem and not a procedural one.

Vulnerability class: 1c, improper change
Time of Introduction: 1, during development
Exploitation Domain involved: 1, UID of *xterm* program file
Effect Domain: 1, any protection domain
Minimum number: 2, the *xterm* process and another process to delete the directory file and link the password file to the name
Source: posted to the USENET

3.3. The *sendmail* Display File Problem

This flaw occurs because of a failure to check permissions properly; specifically the failure to be sure the real user can access the configuration file. The vulnerability clearly is one of “improper validation” (class 2). Again, as *sendmail* must run setuid for the *open*(2) system call to be able to open a protected file, the exploitation domain is that of the owner of *sendmail*; without the setuid feature, it cannot alter any files other than those the user could alter himself or herself. In terms of time of introduction, this was clearly done during development; it is definitely a software problem and not a procedural one.

Vulnerability class: 2, improper validation
Time of Introduction: 1, during development
Exploitation Domain involved: 1, UID of owner of *sendmail*
Effect Domain: 1, any protection domain
Minimum number: 1, the *sendmail* process
Source: posted to the USENET

3.4. Thompson’s Compiler Bug

As discussed earlier, under the assumption that Thompson were authorized to modify the compiler and *login* program, the only vulnerability involved is that created by installing the *login* program, which allows users onto the system without proper authorization. This is a vulnerability of class 1b, “improper protection/improper isolation of implementation detail” because users are allowed to bypass the normal authentication controls. As exploiting the flaw requires enough privileges to install a system program, it would be in *root*’s protection domain on most systems; the effect is that any user’s domain could be accessed. This problem was introduced by Ken Thompson after the compiler had been written, so by our definition it is in the class “during maintenance.”

Vulnerability class: 1b, improper isolation of implementation detail
Time of Introduction: 2, during maintenance
Exploitation Domain involved: 1, *root* protection domain
Effect Domain: 1, any protection domain
Minimum number: 1, the back-door in the *login* program
Source: posted to the USENET

3.5. Thompson’s Compiler Bug Redux

Under the assumption that Thompson were not authorized to modify the compiler and *login* programs, the system begins in a vulnerable state; the vulnerability is that Thompson is able to modify the compiler and *login* programs without authorization. This is clearly an “improper protection/improper choice of initial protection domain” vulnerability. As exploiting the flaw requires enough privileges to install a system program, it would be in *root*’s protection domain on most systems; the effect is that any user’s domain could be accessed. The vulnerability was introduced during operation. Note that the *login* vulnerability described above also exists when Thompson installs the rigged *login* program.

Vulnerability class: 1a, improper choice of initial protection domain
Time of Introduction: 3, during operation
Exploitation Domain involved: 1, *root* protection domain
Effect Domain: 1, any protection domain
Minimum number: 1, the process modifying the compiler and *login* program
Source: posted to the USENET

3.6. Reading Passwords from Terminal Buffers

The flaw here is the failure of the UNIX system to erase sensitive information, such as passwords. (In fact, this vulnerability exists on all UNIX file storage management, not just in the kernel buffers; however, exploiting the kernel buffers is usually simpler.) This is clearly an “improper protection/improper deallocation or deletion” vulnerability. As it can give access to any user’s protection domain, its effect domain falls into the class of “user protection domains;” exploiting it requires access to the kernel memory, so it is in the appropriate user’s (or group’s) protection domain (usually the group *kmem*). The vulnerability also exists because of the way the kernel handles reuse of objects, so it was introduced “during development.”

Vulnerability class: 1e, improper deallocation or deletion
Time of Introduction: 1, during development
Exploitation Domain involved: 1, group *kmem* protection domain

Effect Domain: 1, any protection domain
Minimum number: 1, the process reading the terminal buffer
Source: the author of this paper; others posted it (independently) to the USENET

3.7. Reading Kernel Memory

The vulnerability here is the failure to reset the effective GID of a process. This is an example of improper choice of operand or operation, because the choice of effective GID is inappropriate. It was introduced "during development," and involves sufficient privilege to execute the program involved.

Vulnerability class: 4, improper choice of operand or operation
Time of Introduction: 1, during development
Exploitation Domain involved: 1, group *kmem* protection domain
Effect Domain: 1, any protection domain
Minimum number: 1, the process reading the kernel memory
Source: the author of this paper; others posted it (independently) to the USENET

3.8. *rdist* Race Condition

This flaw may be summarized as the creation of a file, then it being deleted and a new file given its name. In other words, the name of the file *rdist* created is rebound to a file of the user's own choosing. It is a classic case of improper sequencing, because two processes are allowed to write to the file (which for our purposes includes changing the binding of the file's name).

Vulnerability class: 3b, improper sequencing
Time of Introduction: 1, during development
Exploitation Domain involved: 1, protection domain of the owner of the directory where the temp file is put
Effect Domain: 1, any protection domain
Minimum number: 2, the *rdist* process and the process switching the binding of the temporary file
Source: USENET

3.9. Classic Trojan Horse

Because this involves confusion about naming, it is a vulnerability of class 1d, "improper protection/improper naming." It is clearly a problem during operation, as the user can (and usually does) set his or her search path. Finally, as a Trojan horse works in the domain of the executor, and as an effect gives others access to, or information from, that domain, the effects and exploitation domains are those of the user executing the program.

Vulnerability class: 1d, improper naming
Time of Introduction: 3, during operation
Exploitation Domain involved: 1, protection domain of executor

Effect Domain: 1, protection domain of executor
Minimum number: 1, the process executing the Trojan Horse
Source: Dan Edwards

3.10. The *fingerd* Flaw

Because this involves writing beyond the end of an array, and hence going out of bounds, it is a vulnerability of class 2, "improper validation." It is clearly a problem introduced at the time of creation of the program, and so is a development flaw; since the use of *gets* is not intrinsic to design, we can safely speculate this is an implementation failure. Finally, exploiting it requires access to the network (as it is a network daemon) using any identity; the effects involve only the protection domain of the daemon.

Vulnerability class: 2, improper validation
Time of Introduction: 1, during development
Exploitation Domain involved: 3, any protection domain, network
Effect Domain: 1, protection domain of the daemon
Minimum number: 2, the *fingerd* process and the one feeding it input
Source: mailing list (phage, set up during the Internet worm crisis)

3.11. IP Spoofing

This attack involves deceiving a network service which relies on an untrustworthy datum, specifically an IP address in a packet; as this is not suitable for validation, this vulnerability is a vulnerability of class 2, "improper validation." This problem is a development problem, because it is introduced (usually) during the implementation of a network program that must authenticate its peer (it is implementation because the design of a program rarely depends on *how* the authentication of the peer is done, merely that it *be* done). Finally, exploiting it requires access to the network (as it is a network daemon) using any identity; the effects involve only the protection domain of the daemon.

Vulnerability class: 2, improper validation
Time of Introduction: 1, during development
Exploitation Domain involved: 3, any protection domain, network
Effect Domain: 1, protection domain of the server
Minimum number: 2, the server process and the spoofing process
Source: research (widely known when the TCP specification was released)

3.12. Summary

Figure 2 summarizes the number of flaws in this section on each axis of the classification scheme. As more flaws are classified (and we emphasize that the 9 listed are by no means exhaus-

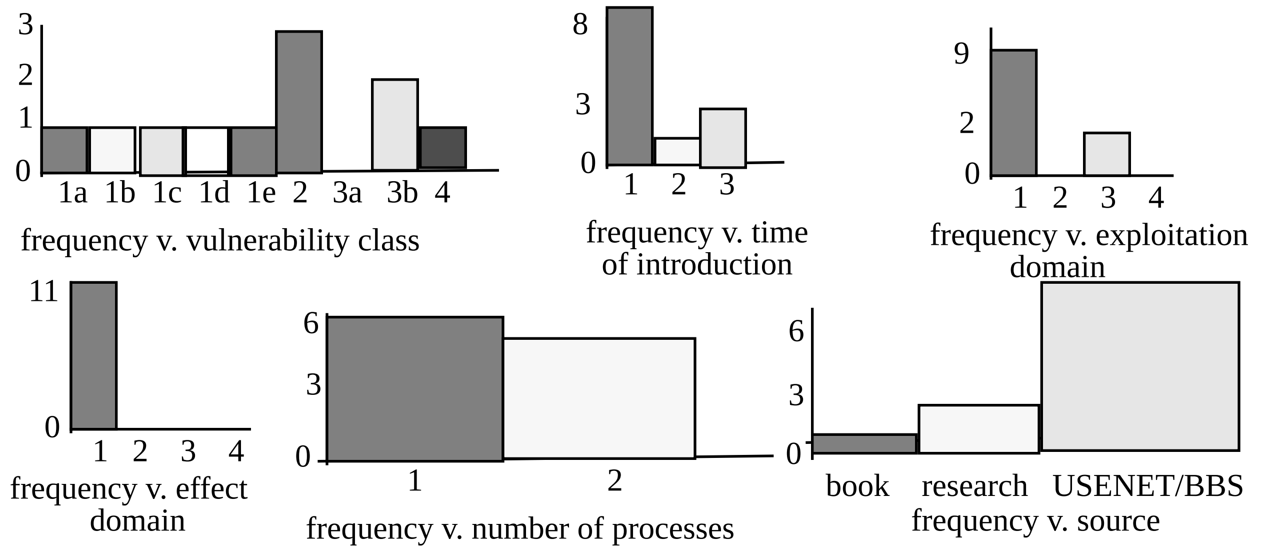

Figure 2. Graphical summary of the frequencies of flaws in each of the six classification schemes used.

tive; for example, of the many flaws listed in [1], most are problems in configuring the protection domain, and experience shows these should be more dominant than in our sample), the data in these charts will become more complete, and more accurate conclusions can be drawn. For now, the most interesting statistic is the location of the first discussion of the flaw, in public: the USENET. This is especially surprising given the low quality of most USENET postings; so, it would seem that the USENET, long demeaned as a frivolity, can be a fruitful source of data for research and analysis in the study of vulnerabilities.

Supporting this analysis are the results of integrating other vulnerabilities into these categories. Of 23 vulnerabilities the origins of which are (believed) to be known, 7 come from USENET, and the other 16 from research (which includes approximately 5 from anonymous papers which can be obtained from bulletin boards). As for the other axes, the categories 1a and 2 dominate the vulnerability classifications, most of the vulnerabilities were introduced in development, with operation running a fairly distant second. All also require 1 or 2 processes.

4. Detection and Elimination of Vulnerabilities

We now turn to the use of the taxonomy in the detection and elimination of vulnerabilities. Unfortunately, there is no mechanistic way to detect a vulnerability and classify it; however, by looking at characteristics of the flaws in each class of vulnerability, we can describe techniques which will aid the analyst in locating problems. In some cases, methods of elimination are clear;

in other cases, containment is more appropriate.

Considerable work has been done on the study of secure operating systems, and intrusion detection mechanisms make the assumption that the kernel is secure (otherwise, the logging mechanisms cannot be trusted). So for this section, we shall focus on both applications-level and program-level security problems, and omit discussion of kernel-level problems entirely. Those interested in such problems are referred to [2].

4.1. Improper Choice of Initial Protection Domain

The vulnerabilities in this class involve incorrectly set permissions when the system starts; these are configuration errors. Detection first requires a knowledge of the security policy, since that policy describes the "proper" configuration. The detection method is to analyze the system to see how closely its configuration conforms to that required by the security policy.

Utility checkers that generate listings of file protections and access rights of processes exist on almost all systems; for the UNIX operating system, *cops* and *tripwire* are the two best known programs. But use of these tools does not solve the problem; it simply presents information which must then be analyzed. To make this analysis effective, one must proceed top-down by understanding what the goals of the protection mechanisms are; only then can one ask if they are being applied correctly. To do so requires a very clear, firm grasp of the site's security policy.

The simplest technique for determining a "safe" configuration of a system is to begin with the security policy, which presents the high-level goals, and refine them accordingly. For example, at an educational institution running the UNIX operating system, the security policy might forbid students sharing or reading other students' homework, but allow faculty to read those homeworks. In this case, there are two groups of users – students and faculty – with faculty able to read students' files but not vice versa. The obvious approach is to have two groups, one of students and one of faculty, and require that all student files be unreadable by world and be group-readable by the faculty group. So, the results of the integrity checking scan (by *cops* or *tripwire*) would show if this were in fact true.

The basic problem with this approach is lack of completeness; specifically, how can one be sure the policy is correctly broken down into checkable elements correctly? Clearly a formal approach such as formal verification or program verification would work best, but these are not yet practicable. The informal technique outlined above requires quite a bit of time initially to analyze the security policy; by rights, this should be done before the computer is selected to ensure it provides security mechanisms adequate to enforce the policy. Once done, it need not be redone. The mechanistic job of checking conformance of the system to the policy simply requires scanning the state of the system and analyzing the output, and can be done automatically.

To return to our example of Thompson's rigged compiler, assume the security policy disallowed Thompson modification access to the compiler. A simple file system scan of Thompson's protection domain would detect that he had such privilege. Thus a security hole would be closed.

Mechanistically, how is such a scan done? The simplest way is to use a file system scanning tool to record all files that Thompson can access, and how. Then those which he can access need to be checked to see if such access can expand his protection domain, and by what objects. If so, the expanded domain needs to be rechecked. Compute the transitive closure of the protection domain under the access functions. At each step, the new elements of the protection domain are compared to the protection domain described by the security policy to see if the two conform. This will uncover any initial protection domain flaws.

4.2. Improper Isolation of Implementation Detail

By their nature, these flaws arise because of multiple paths to a single object; for example, file systems having aliases which allow them to be treated as ordinary objects. The key is the representation of an object as an abstraction, yet reference to it through a path other than through the abstraction's operation.

As an example, suppose a database records its data in a set of files, with each field of a record being stored in a separate file (for example, if the database stored names and account names, one field would contain the names and the other the account names). The database restricts the values of fields when changes are made or records added. Because any program can access the files directly rather than going through the database interface, a potential security problem exists. The underlying flaw is that the instantiation (the files holding the fields of the records) of the abstraction (the records of the database) can be accessed by a mechanism other than the abstraction operations (the database functions).

Detecting flaws of this variety require an analysis of the abstractions used by programs, and their implementation. For example, consider the implementation of the NIS password changing mechanism. In this case, there is a daemon (*yppasswdd*) that changes the NIS master file, and a set of clients (*yppasswd*, *ypchsh*, and *ypchfn*) that communicate the new password record to the daemon. The records were stored as a set of colon-separated fields, one record per line. The relevant fields to this attack are the login shell field (changed by *ypchsh*) and the user information (gecos) field (changed by *ypchfn*). A simple attack was to place a newline in the input to *ypchsh* or *ypchfn*, and then enter a record creating a privileged account; the client would send this information to the daemon, which would update the master file and propagate the changes to systems served by the NIS server. To prevent this attack, the clients all checked for a newline in their input. The abstraction here was the password record; the abstraction operation was the updating of a record by one of the clients.

Unfortunately, any user could write a program to connect to the NIS password daemon directly, bypassing the clients (and the check). The detail of implementation was improperly isolated, and as a result the abstraction operations could be bypassed. The correct view of the abstraction operation is that the daemon, not the clients, implemented it. When this was understood, the check was added to the daemon and the problem was solved.

Searching for these problems requires a systematic decomposition of the system and its attendant programs and information into separate abstractions, at which point the implementation of each abstraction can be checked, and then the interaction of the abstraction operations can be checked. Checking requires a detailed analysis of the nature of the functions implementing the abstraction operations. For this, a security manual of the sort described in [17] is a useful way to specify the security properties of the implementation of the functions.

4.3. Improper Change

Flaws of this category occur when data that is meant to be consistent is not consistent; essentially, one misplaces trust in the integrity of the data. This requires at least two processes, one which relies upon the data and another that alters it.

The reliance upon the data is the key. For example, in the *xterm* flaw, the program assumes that between the check and the open, the file name refers to the same object; as the checking and opening are not part of an atomic operation, that assumption is flawed, leading to the attack. Any pair of UNIX file system calls which involve using the name of an object (as opposed to a descriptor) is vulnerable, although the exploitation may have various effects. For example, if a setuid program first calls *access*(2) to see if a user can write to a file, and then *unlink*(2) to delete it, a

Table 4. Example of system call sequences which create improper change flaws in the UNIX operating system

access	*open*	give read/write access to a protected file
access	*unlink*	delete a system-critical file
access	*chroot*	remove any restrictions on what part of the file tree is visible
creat	*chown*	change ownership of a file the user should not access
open	*rename*	move the wrong file to a system location

malicious user might be able to delete system-critical files using that sequence and thereby deny access to the system. With respect to files, Table 5 summarizes several such pairs of system calls; when these occur in sequence, and refer to the same named object, an improper change flaw has been introduced.

With data in shared memory, similar problems arise (note we assume that the kernel enforces isolation of process memory). If the data is security-critical, such as a file descriptor, it must be kept in protected memory and not shared memory. On most UNIX systems, this is a problem only if the data is made available via some file-sharing mechanism or IPC method such as memory mapping.

Techniques for detecting improper change flaws are very similar to those used for analyzing testing. Research in the application of the testing technique called *slicing* is very promising [18]; its advantage over simple pattern matching is that it detects cases where the same value is stored in different variables, and the different variables are used in the file system or memory management calls that can produce an improper change flaw. The program *rdist* was analyzed using this scheme, and the sequence of system calls shows up quite clearly in the slice (although it should be noted the slicing was focussing on a different problem).

Nevertheless, even with a pattern matching program, the ubiquity of race conditions can be seen. Running a prototype scanner on the *sendmail* program revealed an unknown race condition that enabled an attacker to change the protection mode of any file on the system to world readable [24]. The bug was reported and corrected.

4.4. Improper Naming

At the user level, handling improper naming simply means detecting objects in the user's protection domain with the same name. Such software is straightforward to write; for example, on the UNIX system, a simple shell script can traverse each directory in the user's search path and look for name collisions. Where the problem becomes interesting is if one object has two different names, and therefore two different sets of access privileges.

The UNIX file system associates access control with inodes, not names, so in theory access to the data in a file is controlled by one set of permissions, regardless of the number of names the file has. But in practise, the UNIX system allows direct aliasing (called “hard links”) across directories; that is, the names “/usr/mab/xxx” and “/usr/holly/yyy” may refer to the same file. By preventing the user heidi from searching her directory, holly can prevent her from accessing the data in “/usr/holly/yyy”. But unless the user mab has also prevented heidi from searching his directory, heidi can still gain access to the data. (Note that holly may not want to make the file yyy unreadable by everyone so that others can execute a special program which runs with holly’s privileges to get into the directory, and then uses their privileges to access the file.) Still, detecting instances such as this requires a simple program.

Improper naming also arises with processes, and the UNIX system is particularly susceptible to an attack based upon process name (which is represented as an integer between 0 and some fixed upper bound, the PID). When process *n* communicates with process *m*, it is possible that the process *m* being contacted is not the original process with PID *m* (which has terminated) but is instead another process with the PID *m* (as the UNIX kernel reuses PIDs). This is especially dangerous when temporary file names are based upon the PID of the creating process.

The only way to detect this type of flaw is to look for programs which communicate with other processes and cannot determine when that other process terminates. If so, then the death of process *m* and the reuse of *m* would be undetected by the original process. If, however, the original process refrains from communicating with a terminated process, the flaw is removed.

4.5. Improper Deallocation or Deletion

When an object is improperly deallocated or deleted, the object containing the data is released but the data is not erased. Any subject acquiring that object then has access to the data it contains. This requires that when an object is re-used, its contents be cleared (or set to some fixed value). This is a function of the operating system, and its specifications determine if in fact this sanitizing does occur.

Another form of this attack is the failure to delete ancillary objects when a main object is deleted. A classic instance of this occurred on a UNIX system with a process table of fixed size.

Table 5. Pairs of UNIX library and system calls that allocate and deallocate objects

malloc, calloc	free, cfree	memory allocation (on the heap)
alloca	return	memory allocation (on the stack)
creat, open	close	create/open a file
socket	close	create an IPC channel
fork	exit, wait	create a process
popen	pclose	create a process
fopen	fclose	create/open a file
mmap	close	allocate shared memory

When processes terminated, their process control block remained in the process table until a special process, called the "mop," deleted this information. The mop ran every 30 seconds. Denial of service attacks were frequent, and simply required filling the process table. When the mop tried to run, it could not find a free slot in the process table, and the kernel would issue a panic, crashing the system.

In a program, improper deallocation may refer to using a deleted object, or deallocating an object which was not allocated. For example, on many versions of the UNIX system, when memory is released using *free*, one could still use the values in that memory so long as the memory were not reallocated. This is of course a prime setting for a deallocation flaw, since if a *malloc* call is missed, the data in the *free*d space is suspect.

The simplest way to check for improper deallocation or deletion flaws is to look for all allocations and deallocations, and be sure that every deletion or deallocation has a corresponding open (creation) or allocation. Table 6 summarizes several pairs of allocation and deallocation functions in the UNIX system. Testing techniques, specifically slicing [18], can easily focus on the functions performing the allocation and deallocation, and this will enable rapid checking of the associations.

Problems arise when the flaw involves more than one process; unless shared memory is involved, the flaws will involve files or processes. In this case, the programs for each process must be analyzed together, and more research is needed to determine how to do this efficiently. One step will be the locating of programs which interact through process or file manipulation; the security manual can help here.

4.6. Improper Validation

Improper validation means that insufficient checks are made upon data, and the failure to do so creates a security problem. Note that protection data need not be involved; the failure of *gets* to check the length of its input is a wonderful example; this failure, and the use of *gets* in a system daemon, allowed the Internet worm to gain access to system users (usually daemon or nobody, sometimes root).

Within a single program, the only defense against improper validation is a very careful programming style – checking the types and ranges of arguments to functions, being sure that array (and memory) bounds are not overwritten, and so forth. Using both static and run-time checking are essential, because of the pointer problem; generating code for checking array bounds when an index is involved is easy, but when pointers are used, is much more difficult (for example, if two character arrays are sequential, if a pointer references the first array and then the second, was that intentional or overflowing bounds?). The tools needed are a syntax checker like the UNIX *lint* program or the GNU ANSI C compiler (with warning flags added); these check for type problems, in some cases range overflow (for example, when a parameter has the type of a set or enumerated type, and the corresponding argument has a value not in that set or enumerated type). However, the checks are not exhaustive; for example, if the function expects an argument of 0 or 1, and 2 is passed, the syntax checkers will not report an error.

A runtime checking system provides this type of support. By adding appropriate assertions into the program, one can direct actions to be taken if something unexpected occurs. The problem with such checking mechanisms is that one must understand what the error conditions are before the checker can be instructed to watch for them. In some cases (as with the flag argument, which must be either 0 or 1), the error condition is obvious; with others, it may be far more subtle.

Several of the more common programming errors in programs written for the UNIX environment are summarized in Table 7. A database along the lines of the security manual for the system calls and library functions [17] would enable a tool such as a "security *lint*" to check for many improper validation problems. It would have other uses as well; it would help programmers to write robust code which did not involve security issues.

At runtime, the return value of functions should be checked. A very common error in UNIX programs is to forget to check the return value of a system function, or not to validate the reason for the failure of a system call. For example, an asynchronous read can fail because there is no

Table 7. Some programming tips for avoiding the improper validation flaw

- Use enumerated types instead of integers or macros whenever possible
- Check all arguments for illegal or unexpected values
- When handling errors, do not make assumptions you cannot verify; if in doubt, stop.
- Check the return values of all functions and system calls for errors.
- If your compiler does range checking, use indices and not pointers to reference array elements so the range checking can be used.
- Use library functions that do error checking whenever possible (*e.g., fgets* rather than *gets*)

information to be read, but unless the error code is checked, the failure will appear the same as a failure to read because the file does not exist.

As with the other flaws, detecting problems is greatly aided by the use of an abstraction. The abstraction operations operate upon the objects in specific ways, and those “specific ways” impose constraints on legal values. So the abstraction guides the analyst or programmer to the legal values and operations, and the tools (such as the *security lint*) would validate that only legal values and operations occur.

Another, more pernicious form of improper validation is the ability to masquerade as another. Detecting such a masquerade is beyond the scope of this work; the interested reader is referred to work on intrusion detection systems [19]. However, a failure to select an adequate one-way transform for reusable passwords is an excellent example of a flaw of improper validation.

4.7. Improper Indivisibility

This category involves operations which need to be atomic but are interruptible. At the user or program level, the best example is of file locking. On many UNIX systems, all locks are advisory, and so non-cooperating programs can interfere with one another. Hence, programs which do not respect the locking conventions can breach security.

Detecting problems of this nature requires abstraction again; one must determine which sequences of commands or calls need to be executed atomically. A prime example is the *mknod* followed by *chown* discussed in the *mkdir* attack. Conceptually, the creation of the directory should be by the person asking for it, not by some other user and then transferred. The latter scheme was adopted because the UNIX system then did not have a “create directory” system call. Because of the flaw this introduced, such a system call was added.

Note the relationship between this and category 1c, the change of name flaw; in some sense, these categories overlap, because the improper change of names or privileges often arises due to inadequate indivisibility. As many systems do not provide mechanisms for user-level atomicity (that is, once the sequence of indivisible instructions is started, no other user program may run until that sequence finished) because of the ease of a denial of service attack, mechanisms to prevent flaws of this nature are beyond the users or programmers; they require kernel-level modification. The best that can be done is to use an abstraction to determine what those indivisible operations should be, and use the protection system to enforce them.

Indivisibility occurs when what should be an atomic operation is interrupted, perhaps by a system crash (leaving the system in an inconsistent state) or a telephone being hung up during a login sequence (leaving some parts of the login sequence completed and others not). The key is that the system (or data being accessed) is in an inconsistent state; so, detection mechanisms require knowledge of the nature of consistent states. Then one can check for these flaws by looking for states which are not consistent; for example, UNIX accounts with no corresponding home directories, inconsistent transaction databases, and so forth. Unfortunately, prevention requires kernel-level modifications, since only the kernel can guarantee that a process or action will not be interrupted by another process or action.

Another example, again from the UNIX environment, will bring this point home. The password file is a database containing information about user accounts, including data used to authenticate users. If a user's account is not listed in that file, the user cannot log in. The password file has a fixed name (/etc/passwd). Updating the password file requires that a copy of it be made with the changes, and then the existing password file be replaced with the copy. Older versions of the UNIX system had no atomic file replacement command, so this was done by first deleting the current password file, and then binding the new password file to the old one's name. Often a system would crash after the removing but before the binding; then systems staff would invoke a monitor program, build a password file with one entry – the superuser – log in using that account, and then bind the new password file to the old password file's name. At that point, the system could be rebooted again, normally. The security threat is of course a denial of service attack; the fix was to add an atomic file replacement system call, so the kernel could guarantee that either the old file or the new file would be bound to the name at all times.

4.8. Improper Sequencing

Improper sequencing refers to the incorrect ordering of operations. The best example of this involves two programs writing to the same database. Even if the writes are atomic, the result is indeterminate because the Bernstein conditions[1] are violated.

An excellent example of this is the implementation of the password changing program in many UNIX systems. As was discussed above, the program creates a temporary password file and copies the contents of the old one into it, making the necessary changes as it is copied. If two password changing programs ran simultaneously, whichever bound the name of the password file to its temporary file last would have its changes take effect. The changes made by the other program would be ignored. (The solution was to add a lock file, which prevented the second program from running. Again, though, note the second program stopped because it honored a convention, not because the kernel blocked it.)

A second example is the implementation of the *mkdir* program discussed in Section 3.1. Treating directory creation, deletion, and change of ownership as writes to the parent directory, the problem is again that two processes are trying to write to the same location in a file (*i.e.*, same name of a file in the directory) at once. Again, this violates the Bernstein conditions and causes a security problem.

This type of flaw can be found much in the way improper changes can be found: one needs to determine the sequence of events (here, function and system calls) that can cause an improper serialization flaw to occur, and given that sequence of system and function calls, one can use the testing methods described above to locate such problems. Again, the testing work is preliminary, but has been successful in small cases, and there is every reason to believe it can be generalized to multiple processes.

4.9. Improper Choice of Operand or Operation

Because the choice of operand or operation admits so many possibilities, rather than to take the approach of detecting improper choices, let us approach the problem by detecting all choices and eliminating proper ones. Again, this means we must work from the abstraction on down to the implementation.

1. The Bernstein conditions disallow reading and writing, or two processes writing, simultaneously [23]

A very famous bug arose in some versions of UNIX UUCP when the shell’s built-in operators were augmented to include the in-line command evaluation operator `‘`. The *uuxqt* program, which received commands from the remote program *uux*, would scan the user-supplied command looking for command dividers such as pipes, and check each command to be sure the UUCP system configuration allowed that command to be executed. Since it did not know about the in-line command evaluation operator, it assumed the next word was an argument to the previous command rather than a command itself. It would pass the command to a shell, which would then execute the command. For example, the sequence

```
uux ‘remote!rmail ‘mail local!bishop < /etc/passwd‘ local!bishop
```

would enable bishop to acquire a copy of the remote system’s password file whether or not the UUCP system were configured to allow the remote user to use the *mail* command.

Signs that there may be an instance of this flaw are erroneous types (either in program arguments, function or system call arguments, or in return values); error codes (because this may leave the system in an unsafe state); and anomalous responses, such as “unfairly” favoring one set of users over another (unfair is in quotes because this is a matter of policy, not inherent to the system).

This type of flaw can arise in one of two ways: an abstraction operation may be chosen incorrectly, or the implementation may be poorly (incorrectly) chosen. In the former case, one must analyze the design of the abstraction and its operations to determine what the correct operation is, and under what conditions such operation will occur (or can fail); then, one must ensure the implementation adequately enforces those constraints. In the latter case, by going from implementation to abstraction, one can see what the abstraction operations are (as opposed to what they should be) and analyze those for correctness and fairness in light of the site security policy.

4.10. Summary

On looking at the methods of detection and prevention for all these flaws, the notion of “abstraction” stands out. It is vital to understand our use of this term: we do not mean a formal model (although if feasible that may be used). We mean instead an abstraction in the object-oriented sense: a collection of smaller parts and operations lumped together with well defined interfaces providing the only access to the internal representation. All too often, systems and subsystems are thrown together with little or no thought to how they will interact with other systems or subsystems; thus, as more security is needed, the modularity (if any) of the original entity is lost. The abstraction no longer reflects reality.

So, our suggestion is to look at the implementation itself, which exists, and abstract based upon that. What are the goals of the data being accessed? How can it be accessed? What interfaces are used? Frame these as abstract operations accessing an abstract object. Then ask what operations should be allowed upon that abstract entity. If the abstract operations that are allowed does not correspond to the set of operations that should be allowed, security problems arise. Further, if the implementation of the abstraction allows access through mechanisms other than the abstract operations (for example, direct editing of a database rather than updates), several types of vulnerabilities exist. It is left to the reader, and the security analyst, to determine what those flaws might be.

5. Conclusion

The goal of this work is to argue that a coherent and accurate taxonomy of system vulnerabilities can aid in the detection of exploitation or those flaws, as well as provide guidance in ameliorating or eliminating them.

The work presented herein can best be described as very promising. Because only a few vulnerabilities were classified, we have presented merely an argument that the taxonomy of vulnerabilities is complete. One area for future work is the analysis of other vulnerabilities, their placement in this taxonomy, and whether or not the taxonomy must be expanded to include new vulnerabilities.

Related to this is the correlation of the axes. Currently, we are classifying security flaws based upon 5 axes. But suppose certain types of flaws required 2 processes, and the rest only 1. Then one axis is redundant. Of the flaws classified above, those in vulnerability classes 1c (improper change) and 3b (improper synchronization) all use 2 processes, and the rest 1 process. Is this always true? Analysis of vulnerabilities may reveal other correlations.

As a side point, if the flaws tend to concentrate in some set of classes, this suggests either that the flaws tend to be in those categories, or that we are unaware of flaws in other categories; and in either case, the hunt for flaws in the sparsely populated classes should be intensified. If the flaws tend to occur during development, then more effort in the detection of those flaws must be made during development. Finally, the place where the flaws were first publicized gives guidance about where to look for vulnerabilities information; the author had no idea that so many vulnerabilities came from the USENET.

The detection and prevention work is also at an embryonic stage. Testing techniques, abstraction techniques, and an implementation guide to the security-related properties of each program (or set of programs) are central to effective detection and prevention. In particular, the ability to determine the abstractions that a system is implementing is crucial; once this is done, one needs to check that the abstraction is correctly implemented and properly isolates the implementation details – rather like a reference monitor. The conclusion from this approach is obvious, and somewhat surprising: the same techniques that are used to formally verify the design of a secure system may be applied to existing system. Going from implementation to abstraction is the only additional step.

In practise, formal methods are unlikely to be used as they are still not yet well developed enough, and the burden of abstracting to the degree necessary to use them may be quite high. However, this provides an additional impetus for their study. It also argues for an "informal" formal analysis (that is, analysis and argument rather than abstraction to a mathematical design and proof) for several reasons.

First, understanding the security of the existing system requires an understanding of how all its components work,. and how they are intended to work. All too often, system maintainers and developers patch or build on existing structures without considering the abstraction of the part they are maintaining or developing. This causes major security problems, as we saw with the NIS password example, because often the wrong tools are modified.

Secondly, it forces the managers (purchasers, users, *etc*.) to understand their security needs and how effectively the system can support those needs. For example, as the UNIX operating system made its way from the research laboratory into the commercial world, one major complaint about it was its lack of security mechanisms such as IBM, Burroughs, and other systems provided. Part of the complaints stemmed from an ignorance of the UNIX system, but far more from the misunderstanding that the UNIX system was not designed to be secure [20]. Commercial vendors began to add on security mechanisms for their clients, or third parties began to supply security mechanisms, as the UNIX community absorbed the complaints and tried to deal with them, but, as [21] points out, retrofitted mechanisms are never as good as those designed in from the start. Problems still arise when firms purchase UNIX systems not understanding the type of security those systems provide.

Finally, it enables one to analyze realistically proposed security solutions. If a new mechanism is proposed, the analysts determining what should be recommended can take their solution and analyze it in light of the above classification scheme for vulnerabilities and the abstractions of the system. They can make a more realistic estimate of the effectiveness of the mechanism upon detection of attacks and prevention of security incidents. A guide to where problems have historically been the most likely to occur, and the relationship between vulnerabilities and their risks (how many users they may affect, etc.) may be gleaned from statistics such as the rough statistics presented in section 3.11. This may provide some help in where to look first for problems.

Those statistics are useful in a non-technical way. They provide some ideas about sources for vulnerabilities: bulletin board systems, the USENET, and other non-academic channels. Interestingly enough, the emergency response teams do not publish descriptions of vulnerabilities (although some do publish notices – not details – of the vulnerabilities when the fixes are available); but others are much less recent. What exactly is the legal liability for posting a script for exploiting a vulnerability to a public place such as the USENET? That has not been decided. It undoubtedly will be.

The role of auditing is another issue. In this paper we have not discussed auditing in detail, primarily because of a lack of data. However, with the proper view, auditing can play a major role in the detection of attacks exploiting some of the vulnerabilities above. The key is to determine the signature of the flaws, and then look for commands that contain those characteristic patterns. For example, in the UNIX system, one signature would be an *open* system call followed by a *chown* system call; this indicates a possible improper change (category 1c) flaw. So, instead of looking for the specific high-level attack, one need only look for the low-level exploitation.

This ties into the notion of goal-oriented auditing and logging, which is an extension of the work of [22]. Using that approach, one determines first what one will use the auditing mechanisms to look for, and then builds in the audit mechanism to make the appropriate log entries. If logging is already provided (as is the case here), this refinement process indicates exactly what to look for.

We have outlined very rough mechanisms for both detection and prevention; these mechanisms need to be refined, and in some cases tools crafted, to aid the security analyst in his or her task. We need to analyze audit trails to determine what audit entries will indicate the attack has been launched; we can do this in part by using goal oriented auditing and logging, where we proceed from the end result we want to what the logs need to record. This future research is one of the most exciting areas in the field of computer security.

Acknowledgments. My most sincere thanks to Kevin Zeise and Scott Waddell, for very useful discussions of the framework presented here, and for their incisive comments on previous versions of this draft; also to Toney Jennings, for his unfailing support for this work. Tom Longstaff provided the name *security lint* for the tool described in section 4.6.

This work was funded by grant TDS-94-140 from Trident Data Systems to the University of California at Davis.

6. References

[1] Dan Farmer and Wietse Venema, "Improving the Security of Your Site by Breaking Into It," USENET posting (Dec. 1993)

[2] R. Bisbey II and D. Hollingsworth, "Protection Analysis Project Final Report," ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute (May, 1978); *cited in* [3]

[3] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi, "A Taxonomy of Computer Security Flaws with Examples," *submitted for publication*.

[4] Peter G. Neumann, "Computer System Security Evaluation," *1978 National Computer Conference Proceedings* (*AFIPS Conference Proceedings* **47**), pp. 1087–1095 (June 1978).

[5] C. R. Attanasio, P. W. Markstein, and R. Phillips, "Penetrating an Operating System: a Study of VM/370 Integrity," IBM Systems Journal **15**(1), International Business Machines Corp., pp. 102–106 (1979).

[6] B. Hebbard, P. Grosso, T. Baldridge, C. Chan, D. Fishman, P. Goshgarian, T. Hilton, J. Hoshen, K. Hoult, G. Huntley, M. Stolarchuk, and L. Warner, "A Penetration Analysis of the Michigan Terminal System," *Operating Systems Reviews* **14**(1) pp. 7–20 (Jan. 1980)

[7] A. L. Wilkinson, D. H. Anderson, D. P. Chang, Lee Hock Hin, A. J. Mayo, I. T. Viney, R. Williams, and W. Wright, "A Penetration Analysis of a Burroughs Large System," *Operating Systems Review* **15**(1) pp. 14–25 (Jan. 1981).

[8] Richard R. Linde, "Operating System Penetration," *1975 National Computer Conference Proceedings* (*AFIPS Conference Proceedings* **44**), pp. 361–368 (May 1975).

[9] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security Analysis and Enhancements of Computer Operating Systems," NBSIR 76–1041, Institute for Computer Sciences and Technology, National Bureau of Standards (Apr. 1976)

[10] Matt Bishop, "Security Problems with the UNIX Operating System," *unpublished* (1983).

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

[12] Dorothy E. Denning, "An Intrusion Detection Model," Technical Report CSL-149, Com-

puter Science Laboratory, SRI International (Nov. 1985).

[13] Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, Inc. (1987).

[14] K. Thompson, “Reflections on Trusting Trust,” *Communications of the ACM* **27**(8) pp. 761–763 (Aug. 1984).

[15] A. V. Discolo, “4.2 BSD Unix Security,” *unpublished* (1985).

[16] Shari Lawrence Pfleeger, *Software Engineering: The Production of Quality Software* (second edition), Macmillan Publishing Company, New York (1991).

[17] Matt Bishop, “Analyzing the Security of an Existing Computer System”, *1986 Proceedings of the Fall Joint Computer Conference*, pp. 1115-1119 (Nov. 1986)

[18] George Fink and Karl Levitt, “Property-based testing of privileged programs,” *Proceedings of the Tenth Annual Computer Security Applications Conference*, to appear.

[19] Dorothy Denning, “An Intrusion-Detection Model,” *Proceedings of the 1986 IEEE Symposium on Security and Privacy* pp. 118-131 (Apr. 1986).

[20] Dennis Ritchie, “On the Security of UNIX,” in *UNIX System Manager’s Manual, 4.3 Berkeley Software Distribution, Virtual VAX–11 Version*, Computer Science Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, pp. 17:1-3 (Apr. 1986)

[21] J. Saltzer and M. Schroeder, “The Protection of Information in Computer Systems,” *Proceedings of the IEEE* **63**(9)pp. 1278-1308 (Sep. 1975).

[22] Daniel Bonyun, “The Role of a Well-Defined Auditing Process in the Enforcement of Privacy Policy and Data Security,” *Proceedings of the 1981 Symposium on Security and Privacy* pp. 19-25 (Apr. 1981).

[23] Mamoru Maekawa, Arthur E. Oldehoeft, and Rodney R. Oldehoeft, *Operating Systems Advanced Concepts*, Benjamin/Cummings Publishing Company (1987)

[24] Matt Bishop, “A Race Condition Scanner,” in preparation.