A Critical Analysis of Vulnerability Taxonomies

Matt Bishop and David Bailey

CSE-96-11

September 1996

A Critical Analysis of Vulnerability Taxonomies

Matt Bishop[1]

Dave Bailey[2]

1.0 Introduction

Computer vulnerabilities seem to be omnipresent. In every system fielded, programming errors, configuration errors, and operation errors have allowed unauthorized users to enter systems, or authorized users to take unauthorized actions. Efforts to eliminate the flaws have failed miserably; indeed, sometimes attempts to patch a vulnerability have increased the danger. Further, designers and implementers rarely learn from the mistakes of others, in part because these security holes are so rarely documented in the open literature.

A *taxonomy* is a system of classification allowing one to uniquely identify something. The best known example, the science of systematics, classifies animals and plants into groups showing the relationship between each. Further, the classification is *unique*, so two of the same animal will always be classified with the same groups. That is, if one considers the hierarchy to be a tree structure and uniquely numbers each branch, each species of animal or plant is uniquely identified by a 6-tuple (kingdoms, phylums, classes, orders, family, genus).

A taxonomy for security vulnerabilities should provide the same benefits. The specific goals of such a taxonomy are to provide a historical record of the vulnerabilities in a form that system designers and implementers can use to anticipate flaws in their systems; to describe the vulnerabilities in a form useful for detection; to show common characteristics in related flaws for prevention and elimination; and to enable a security monitor to detect exploitation (or attempted exploitation) of the flaws. A taxonomy similar to the biological classification of plants and animals will do these.

Such a taxonomy allows one to classify each vulnerability as a unique ordered tuple. This is essential to detecting new vulnerabilities. Perhaps more importantly, it allows us to determine how many instances of a larger class of flaws are known, which in turn suggests where efforts to reduce or eliminate the flaws should be focused. It also allows us

1. Author's address: Department of Computer Science, University of California at Davis, Davis, CA 95616-8562; electronic mail address: bishop@cs.ucdavis.edu. This work was supported through a contractual arrangement with the United States Air Force.
2. Author's address: Galaxy Computer Services, Inc., P. O. Box 21069, Albuquerque, NM 87154; electronic mail address: daveb@gcsi.com. This work was supported by the Office of INFOSEC Computer Security through award MDA904-95-C-2107.

to characterize conditions under which the flaw arises, suggesting ways to detect new instances of the flaw.

In this work we distinguish between a vulnerability and an *attack*. An attack is a specific method to exploit a vulnerability; that is, the attack is the program (command script, JCL, *etc.*) or the social method (bamboozlement, lying, misleading, *etc.*) used to take advantage of the vulnerability to achieve the desired end. Drawing on the analogy with systematics, given the right taxonomy, the vulnerability would be the genus; the attack, the specific creature (although in practise, the attack can be taxonomized further).

In the 1970s, two major studies attempted to taxonomize security flaws. One, the RISOS study [1], focused on flaws in operating systems; the other, the Program Analysis (PA) study [4], included both operating systems and programs. Interestingly enough, the taxonomies both presented were similar, in that the classes of flaws could be mapped to one another. Since then, other studies have based their taxonomies upon these results [3][5]. However, the classifications defined in these studies are not taxonomies in the sense that we have used the word, for they fail to define classification schemes that identify a unique category for each vulnerability.

Aslam's recent study [2] approached classification slightly differently, through software fault analysis. A decision procedure determines into which class a soft ware fault is placed. Even so, it suffers from flaws similar to those of the PA and RISOS studies.

The next section contains a precise definition of *taxonomy*, as well as a review of the PA, RISOS, and Aslam classification schema. The third section shows that two security flaws may be taxonomized in multiple ways under all of these schemes. The paper concludes with some observations on taxonomies and some ideas on how to develop a more precise taxonomy.

2.0 Vulnerabilities, Attacks, and Taxonomies

A computer system is composed of *states* describing the current configuration of the entities that make up the computer system. The system computes through the application of *state transitions* that change the state of the system. All states reachable from a given initial state using a set of state transitions fall into the class of *authorized* states or the class of *unauthorized* states, and all state transitions are *authorized* or *unauthorized*, as defined by a security policy. In this paper, the definitions of these classes and transitions is considered axiomatic.

A *vulnerable* state is an authorized state from which an unauthorized state can be reached using authorized state transitions. A *compromised state* is the state so reached. An *attack* is a sequence of authorized state transitions which end in a compromised state. By definition, an attack begins in a vulnerable state.

A *vulnerability* is a characterization of a vulnerable state which distinguishes it from all non-vulnerable states. If generic, the vulnerability may characterize many vulnerable states; if specific, it may characterize only one. For example, a UNIX system with the password file world writable is in a vulnerable state, but the vulnerability may be stated as either the password file being writable (specific) or the system's protection domain being incorrectly set (generic).

Given a computable function with domain the set of vulnerable states and range a finite set of integers, a *decision procedure* refers to the application of the function to a specific vulnerable state. A *taxonomy* is a sequence of decision procedures which classifies each state as exactly one tuple.

Conceptually, one may think of a taxonomy as a tree. Each interior node represents a discriminating *property* distinguishing one class of vulnerabilities from another, and each of the branches between the node and its children represent a possible output of the decision procedure for that classification question. The leaves represent classifications of the vulnerabilities, and the corresponding tuple is the sequence of outputs from the decision procedures at each node encountered in the path from the root to the leaf.

Uniqueness of vulnerability classification requires that distinct vulnerabilities have distinct tuples. The use of tuples as classification mechanisms allows identification of similar classes of vulnerabilities, through common ancestor paths.

It also provides a mechanism to determine if the taxonomy is incomplete. Should a decision procedure require a result not in the function's range, the taxonomy must be extended to include that value in the function's range. As an example from systematics, if the decision procedure is to determine "body covering," the outputs would be labelled "fur," "hair," and "scales." If the body covering of a creature ("birds") does not correspond to one of these outputs, the schema is incomplete and must be expanded by adding a new output ("feathers"). Finally, the discriminating properties will differ along different paths through the tree; again using systematics, if the creature is a bird, asking details about the placement and hardness of its scales is irrelevant, just like asking about the color of feathers on a fish.

This type of taxonomy captures that (1) each different vulnerability has a single, unique corresponding tuple, and (2) there exists a well-defined procedure for deriving a tuples corresponding to a vulnerability.

2.1 The Program Analysis Study

Neumann's presentation [6] of this study organizes the nine classes of flaws to show the connections between the major classes and subclasses of flaws:

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.

Placing this into the framework of a taxonomy, some vulnerabilities have 1-tuples (class 2 or 4) and some have 2-tuples (class 1 or 3). But the breadth of these categories argues against uniqueness holding; for example, tuples for the “time of check to time of use” flaw are 1c, improper change, and 3a, improper indivisibility. In fact, this ambiguity lies at the heart of the problem with this system, but the problem is much more acute than mere ambiguity.

2.2 The RISOS Study

The RISOS (Research Into Secure Operating Systems) study defines seven classes of security flaws:

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.

Here, all vulnerabilities have a 1-tuple. Like the PA study, some ambiguity between the classes indicates that one vulnerability may have multiple tuples; for example, if one passes a pointer to an address in supervisor space to a kernel routine which then changes it, the tuple of the flaw is 1 (incomplete parameter validation) and 6 (violable prohibition/limit). Again, this is a symptom of the problem with using these classes as a taxonomy.

2.3 Aslam's Taxonomy

This taxonomy wad developed to organize vulnerability data being stored in a database. Consequently it is far more detailed than the other two, but it focuses specifically on UNIX faults at the implementation level:

1. Operational fault (configuration error)
    - 1a. Object installed with incorrect permissions
    - 1b. Utility installed in the wrong place
    - 1c. Utility installed with incorrect setup parameters
2. Environment fault
3. Coding fault
    - 3a. Condition validation error
        - 3a1. Failure to handle exceptions
        - 3a2. Input validation error
            - 3a2a. Field value correlation error
            - 3a2b. Syntax error
            - 3a2c. Type & number of input fields
            - 3a2d. Missing input
            - 3a2e. Extraneous input
        - 3a3. Origin validation error
        - 3a4. Access rights validation error

        3a5. Boundary condition error

    3b. Synchronization error

        3b1. Improper or inadequate serialization error

        3b2. Race condition error

The taxonomy of this study is more precise than those of the PA and RISOS studies. It provides more depth for classifying implementation-level flaws (“faults” in the language of this study). This appears to meet our requirements at the implementation level; the taxonomy clearly lacks the high-level categories to classify design errors. In fact, it suffers from a more severe problem.

2.4 First Flaw: *xterm* log file flaw

The program *xterm* is a program that emulates a terminal under the X11 window system. For reasons not relevant to this discussion, it must run as the omnipotent user *root* on UNIX systems. It enables the user to log all input and output to a file. If the file does not exist, *xterm* creates the log file and makes it owned by the user; if the file already exists, *xterm* checks that the user can write to it before opening the file. As any *root* process can write to any file on the system, the extra check is necessary to prevent a user from having *xterm* append log output to (say) the system password file, and gain privileges by altering that file.

Suppose the user wishes to log to an existing file. The following code fragment opens the file for writing:

```
if (access(“/usr/tom/X”, W_OK) == 0){
    fd = open(“/usr/tom/X”, O_WRONLY|O_APPEND);
```

The semantics of the UNIX operating system cause the name of the file to be loosely bound to the data object it represents, and the binding is asserted each time the name is used. If the data object corresponding to `/usr/tom/X` changes after the *access* but before the *open*, the *open* will not open to the file checked by *access*. So during that interval an attacker deletes the file and links a system file (such as the password file) to the name of the deleted file. Then *xterm* appends logging output to the password file. At this point, the user can create a *root* account without a password and gain *root* privileges. Figure 1 summarizes this.

2.5 Second Flaw: *fingerd* buffer overrun flaw

The Internet worm of 1988 [7] publicized this flaw, but it continues to recur, most recently in implementations of browsers for the World Wide Web and some versions of the SMTP agent *sendmail*. The *finger* protocol obtains information about the users of a remote system. The client program, called *finger*, contacts a server, called *fingerd*, on the remote system, and sends a name of at most 512 characters. The server reads the name, and returns the relevant information.

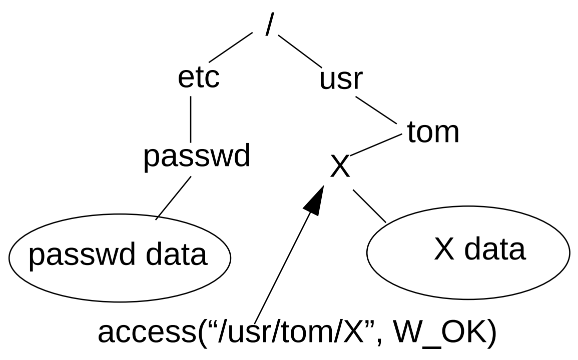

Figure 1a.

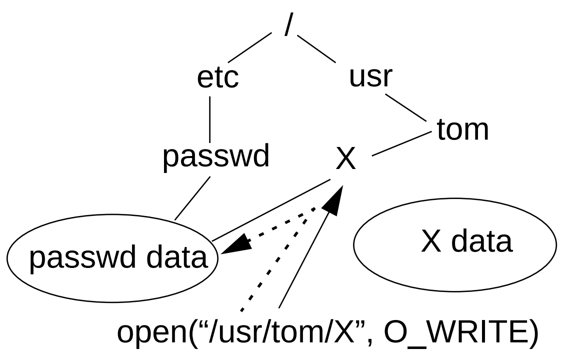

Figure 1b.

Figure 1. Figure 1a shows the state of the system at the time of the *access* system call; the solid arrow indicates the *access* refers to "/usr/tom/X". Both "/usr/tom/X" and "/etc/passwd" name distinct objects. However, before the process makes its *open* system call, "/usr/tom/X" is deleted and a direct alias (hard link) for "/etc/passwd" is created, and is named "/usr/tom/X". Then the *open* accesses the data associated with "/etc/passwd" when it opens "/usr/tom/X", since "/usr/tom/X" and "/etc/passwd" now refer to the same file. Figure 1b shows this, with the dashed arrow indicating which data is actually read and the solid arrow indicating the name given to *open*.

But the server does not check the length of the name that *finger* sends, and because the storage space for the name is allocated on the stack, directly above the return address for the I/O routine. The attacker writes a small program (in machine code) to obtain a command interpreter, and pads it to 512 bytes. He or she then sets the next 24 bytes to return to the input buffer instead of to the rightful caller (the main routine, in this case). The entire 536 byte buffer is sent to the daemon. The first 512 bytes go in the input storage array, and the excess 24 bytes overwrite the stack locations in which the caller's return address and status word are stored. The input routine returns to the code to spawn the command interpreter. The attacker now has access to the system. See Figure 2.

2.6 Segue

These flaws were chosen because the underlying problems are typical of security flaws. Further, both types exist on many different systems, and are frequent relative to the number of security flaws found. Fortunately, they are not always exploited, and can occur without causing security problems. Unfortunately, they pose enough security problems to warrant further study.

3.0 Discussion of the Classification Problem

Two issues central to understanding the problems with the three taxonomies are: what is the point of view of the flaw, and what is the level of abstraction?

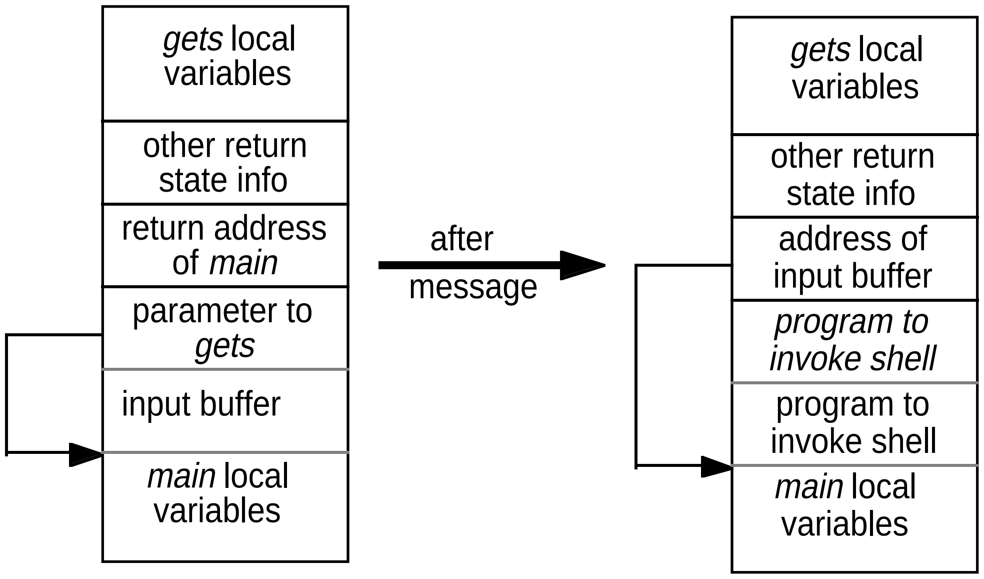

Figure 2. The picture to the left is the stack frame of *fingerd* when input is to be read. The figure on the right is that same stack after the bogus input is stored. The input string overwrites the input buffer and parameter to *gets*, allowing a return to the contents of the input buffer.

Both flaws discussed above depend upon the interaction of two processes: the trusted process (*xterm* or *fingerd*) and a second process (the *attacker*). For the *xterm* flaw, the attacker deletes the existing log file and inserts a link to the password file; for the *fingerd* flaw, the attacker writes a name which is too long. Further, the processes use operating system services to communicate. So, three processes are involved: the flawed process, the attacker process, and the operating system service routines. The view of the flaw when considered from the perspective of any of these may differ from the view when considered from the perspective of the other two. For example, from the point of view of the flawed process, the flaw may be an incomplete validation of a parameter, because the process does not adequately check the parameter it passes to the operating system via a system call. From the point of view of the operating system, however, the flaw may be a violable prohibition/limit, because the parameter may refer to an address outside the process' space. Which classification is appropriate?

Levels of abstraction muddy this issue more. At the lowest level, the flaw may be (say) an inconsistent parameter validation, because successive system calls do not check that the argument refers to the same object. At a higher level, this may be characterized as a race condition, or asynchronous-validation/inadequate-serialization problem. At an even higher level, it may be seen as an exploitable logic error, because a resource (object) can be deleted while in use.

The levels of abstraction are defined differently for every system, and this contributes to the ambiguity. (For example, the THE system has 6; PSOS has 15.) In what follows, the "higher" the level, the more abstract it is, without implying precisely where in the abstraction hierarchy either level occurs. Only the relationship, not the distance, of the levels is important in this context.

We now expand upon these questions using our two sample flaws.

3.1 The *xterm* log file flaw

We begin with the PA taxonomy. From the point of view of the *xterm* process, the flaw is clearly a type 1c flaw, as the problem is that between the time of check (*access*) to time of use (*open*) the referent of the name changes. However, with respect to the attacker process, the flaw is of type 1e, because something (in this case the binding between the name and the referent) is being deleted improperly. And from the operating system's point of view, the flaw is a type 3a flaw, as the opening of the file should atomically check that the access is allowed.

Reconsider the problem at a higher level of abstraction from the point of view of the operating system. At this level, a directory object is seen simply as an object; deletion and creation of files in the directory are semantically equivalent to writing in the directory, and obtaining file status and opening a file require that the directory be read. In this case, the flaw may be seen as a violation of the Bernstein conditions, which means that the flaw is one of improper sequencing, or type 3b.

At the abstraction level corresponding to design, the attacking process should not be able to write into the directory in the first place, leading to a characterization of the flaw as type 1a, improper initial protection. This is not a valid characterization at the implementation level, since both the attacking process and the *xterm* are being executed by the same user, and the semantics of the implementation of the UNIX operating system require that both processes be able to access the same objects in the same way.

At the implementation level, with respect to the *xterm* process and the RISOS taxonomy, the *xterm* flaw is clearly of type 4 since the file access is checked and then opened non-atomically. From the point of view of the attacker, the ability to delete the file makes the flaw type 7, an exploitable logic error, as well as type 6, since the attacker is manipulating a binding in the system's domain. And from the operating system's point of view, the flaw is a type 2 flaw, inconsistent parameter validation, because the access check and open use the same parameters, but the objects they refer to are different and this is not checked.

Interestingly, moving up in the hierarchy of abstractions, the flaw may be characterized as a violation of the Bernstein conditions, or improper sequencing; this is the same as a type 4 flaw, or the non-atomicity of an operation which should be atomic. So the process view prevails.

At the design level, that a write is allowed where it should not be is a flaw of type 5, inadequate identification/authorization/authentication, because the resource (the containing directory) is not adequately protected. Again, due to the nature of the protection model of the UNIX operating system, this would not be a valid characterization at the implementation level.

Thus, this single flaw has several different characterizations. At the implementation level, depending on the classifier's point of view, the *xterm* flaw can be classified in 3 different ways. Trying to abstract the underlying principles under one taxonomy places the flaw is a fourth class, and under the other one view (the *xterm* process view) prevails. Moving up to the design level, a completely different classification is needed. Clearly, this fails to meet our definition of taxonomy, so the ambiguity in the PA classifications make it difficult to use to classify flaws for our purposes.

Finally, consider Aslam's taxonomy. The selection criteria for fault classification [2] places the flaw in class 1a from the point of view of the attacking program (object installed with incorrect permissions, because the attacking program can delete the file), in class 3a4 from the point of view of the *xterm* program (access rights validation error, as *xterm* does not properly validate the file at the time of access), and class 3b1 from the point of view of the operating system (improper or inadequate serialization error, as the deletion and creation should not be interspersed between the access and *open*). As an aside, absent the explicit decision procedure, the flaw could also have been placed in class 3b2, race conditions. So this taxonomy satisfies the decision procedure criteria, but not the uniqueness one.

That this ambiguity of classification is not a unique characteristic of one flaw is apparent when we study the second flaw, that of *fingerd*.

3.2 *fingerd* buffer overrun flaw

With respect to the *fingerd* process and the PA taxonomy, the buffer overflow flaw is clearly a type 2 flaw, as the problem is not checking parameters, leading to addressing memory not in its memory space by referencing through an out-of-bounds pointer value. However, with respect to the attacker process (the *finger* program), the flaw is of type 4, because an operand (the data written onto the connection) is improper (specifically, too long, and arguably not what *fingerd* is to be given). And from the operating system's point of view, the flaw is a type 1b flaw, because the user is allowed to write directly into what should be in the process' space (the return address), and execute what should be treated as data only. Note this last is also an architectural problem.

Moving still higher in the layers of abstraction, the storage space of the return address is a variable or an object. From the operating system point of view, this makes the flaw be type 1c, because a parameter – specifically, the return address – changes unexpectedly. From the *fingerd* point of view, though, the more abstract issue is the execution of data (the input); this is improper validation, specifically not validating the type of the instructions being executed. So the flaw is a type 2 flaw.

At the highest level, the system is changing a security-related value in memory, and is executing data that should not be executable. Hence this ia again a type 1a flaw. But this is not a valid characterization at the implementation level, because the architectural design of the system requires the return address to be stored on the stack, just as the input buffer is allocated on the stack; and as the hardware supporting the UNIX operating system does not have per-word protection (instead, it is per page or per segment), the system requires that the process be able to write to, and read from, its stack.

With respect to the *fingerd* process using the RISOS taxonomy, the buffer overflow flaw is clearly a type 1 flaw, as the problem is not checking parameters, allowing the buffer to overflow. However, with respect to the *finger* process, the flaw is of type 6, because the limit on input data to be sent can be ignored (violated). And from the operating system's point of view, the flaw is a type 5 flaw, because the user is allowed to write directly to what should be in the process' space (the return address), and execute what should be treated as data only.

Moving still higher, the storage space of the return address is a variable or an object. From the operating system point of view, this makes the flaw be type 4, because a parameter – specifically, the return address – changes unexpectedly. From the *fingerd* point of view, though, the more abstract issue is the execution of data (the input); this is improper validation, specifically not validating the type of the instructions being executed. So the flaw is a type 5 flaw.

At the highest level, this is again a type 5 flaw, because the system is changing a security-related value in memory, and is executing data that should not be executable. Again, due to the nature of the protection model of the UNIX operating system, this would not be a valid characterization at the implementation level.

Finally, under Aslam's taxonomy, the flaw is in class 3a5 from the point of view of the attacking program (boundary condition error, as the limit on input data can be ignored), in class 3a5 from the point of view of the *xterm* program (boundary condition error, as the process writes beyond a valid address boundary), and class 2 from the point of view of the operating system (environment fault, as the error occurs when the program is executed on a specific machine, specifically a stack-based machine). As an aside, absent the explicit decision procedure, the flaw could also have been placed in class 3a4, access rights validation error, as the code executed in the input buffer should be data only; and the return address is outside the process' protection domain, yet is altered by it. So again, this taxonomy satisfies the decision procedure criteria, but not the uniqueness one.

The RISOS classifications are somewhat more consistent among the levels of abstraction, since the improper authorization classification runs through the layers of abstraction. However, point of view plays a role here, as that classification applies to the operating system's point of view at two levels, and to the process view between them. This, again, vitiates the usefulness of the classification scheme for our purposes.

3.3 Summary of Analysis

The flaw classification is not consistent among different levels of abstraction. Ideally, the flaw should be classified the same at all levels (possibly with more refinement at lower levels). This problem is ameliorated somewhat by the overlap of the flaw classifications, because as one refines the flaws, the flaws may shift class. But the classes themselves should be distinct; they are not, leading to this problem.

The point of view is also a problem. The point of view should not affect the class into which the flaw falls; but as the examples show, it clearly does. So, can we use this as a tool for classification – that is, classify flaws based on the triple of the classes that they fall under? The problem is the classes are not partitions; they overlap, and so it is often not clear which class should be used for a component of the triple.

In short, the two examples demonstrate why the PA, RISOS, and Aslam classifications do not meet our need for taxonomies: they meet neither the uniqueness nor the well-defined decision procedure requirement.

4.0 Conclusion

The PA and RISOS studies give a high-level view of the basic types of flaws, and with some refinement might serve admirably as high-level classes in a taxonomy. But these studies do not present a taxonomy. Aslam's study gives a low-level view of flaws, and his classification scheme is tailored towards a particular operating system. The lack of levels of abstraction is a serious problem. More serious is that discriminatory criteria are orthogonal in the taxonomy when the orthogonality is not apparent. As an example, a flaw can be classified as a race condition (3b2), or as an object installed with incorrect permissions (1a), but not both. Yet these two together characterize a large class of race conditions.

Discriminating properties, as embodied in the decision functions, determine classification. From the above, such properties may occur anywhere in the taxonomic tree, and indeed may occur multiple times at different points in the classification procedure. For example, "incorrect permissions" may be the discriminating property at a node 2 away from the root for race conditions, but 3 away from the root for unauthorized access. This eliminates the problem of unnecessary orthogonality in the taxonomy.

The discriminating properties will be as important as the vulnerabilities they classify. For example, suppose a particular security vulnerability was first found in OS/360. Would it be likely to recur in the system at hand even though the hardware and software are completely different? If yes, then the vulnerability would be an instance of a "genus" of vulnerabilities; an example of such a genus would be the time of check to time of use flaw, which occurs in many different systems.

The role of attack classifications also merits some discussion. Although attack classification and vulnerability classification are often considered the same problem, this paper considers an attack to be an exploitation of a vulnerability. The distinction allows one to separate the mechanism of the exploitation from the underlying vulnerability; for example, two different attacks may use two different programs to exploit the same vulnerability, if the two different programs have the same programming flaw. Now, one can view a vulnerability as a containing class (such as genus is in the biological taxonomy) and attacks as elements of that class. This unifies the two into a complete and consistent taxonomy, much as Landwehr *et al.* tried to do in [5].

The game of Animal suggests an approach to building a taxonomy. Given two flaws, some (discriminating) question captures the difference between them. Then, for the next flaw, either it matches one of the earlier ones found, or some other (discriminating) question(s) distinguish it from the other two flaws. Proceeding inductively, it is clear a taxonomy can be constructed. The relationship between levels in the taxonomic tree is not clear; can the discriminatory properties be grouped so that like properties are on the same level? How can "like" properties be characterized?

The open research issues can be summarized quite simply. First, although we rejected two classification schemes as unsuitable for taxonomies, can a new taxonomy use their discriminatory questions? Indeed, are their questions even at the same level in the taxonomy? Secondly, what discriminatory questions should it have? Can the levels of abstraction be mirrored by classifications at nodes close to the root, so that distance from the root implies a lower level of abstraction, or must the levels be intermingled with implementation questions? Finally, what are appropriate top-level classifications? More experience in this area is needed.

5.0 References

[1] 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)

[2] T. Aslam, "A Taxonomy of Security Faults in the UNIX Operating System," Master of Science thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907 (1995).

[3] M. Bishop, "A Taxonomy of UNIX System and Network Vulnerabilities," Technical Report 95-10, Department of Computer Science, University of California at Davis, Davis, CA (1995).

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

[5] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A Taxonomy of Computer Program Security Flaws," *Computing Surveys* **26**(3) pp. 211–255 (Sep. 1994).

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

[7] D. Seeley, “A Tour of the Worm,” *Proceedings of the 1989 Winter USENIX Technical Conference* pp. 287–304 (Jan. 1989).