Vulnerabilities Analysis

Matt Bishop
Department of Computer Science
University of California at Davis
One Shields Ave.
Davis, CA 95616-8562

Abstract

This note presents a new model for classifying vulnerabilities in computer systems. The model is structurally different than earlier models, It decomposes vulnerabilities into small parts, called "primitive conditions." Our hypothesis is that by examining systems for these conditions, we can detect vulnerabilities. By preventing these conditions from holding, we can prevent vulnerabilities from occurring, even if we do not know that the vulnerability exists. A formal basis for this model is presented. An informal, experimental method of validation for non- secure systems is described. If the model accurately describes existing systems, it guides the development of tools to analyze systems for vulnerabilities.

1. Introduction and Problem Statement

Errors in computer systems and programs are called *bugs*. Many studies have analyzed the origins of bugs, how to decrease the number of bugs in a system, and how to test for (informally verify) and prove (formally verify) the lack of bugs in a program. Tools such as bounds-checking compilers, dynamic debuggers, and profilers let programmers check for specific bugs in the program. Methodologies such as stepwise refinement breaks the process of design and implementation into a set of steps that simplify the refinement of the problem to a solution. But even with all these techniques and technologies, bugs still exist.

Bugs that enable users to violate the site security policy are called *vulnerabilities* or *security holes* (or *holes* for short). 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.

Security holes arise from inconsistencies or errors in design, in implementation, in operation, and in maintenance. Inconsistencies occur when the requirements of the security policy (an axiomatic definition of the term "secure," developed from external factors that do not concern us here) are phrased poorly, misunderstood, or not interpreted consistently. For example, suppose a policy states that "electronic mail will be confidential." Does this mean that the sender’s address and the recipient’s address are to be confidential? The designer of the transport mechanism may assume so, and expect the user agent to encode the sender and recipient names. But the designer of the user agent may expect the transport layer to encode the sender’s address and recipient’s address after pulling out the source and destination addresses. The failure to protect these addresses may violate the security policy, causing a vulnerability.

The theme of vulnerabilities analysis is to devise a classification, or set of classifications, that enable the analyst to abstract the information desired from a set of vulnerabilities. This information may be a set of signatures, for intrusion detection; a set of environment conditions necessary for an attacker to exploit the vulnerability; a set of coding characteristics to aid in the scanning of code; or other data. The specific data used to classify vulnerabilities depends upon the specific goals of the classification. This explains why multiple classification schemes are extant. Each serves the needs of the community or communities to which its classifier belongs.

The problem of interest is to discover these vulnerabilities before attackers can exploit them. We seek a classification scheme with the following properties:

1. Similar vulnerabilities are classified similarly. For example, all vulnerabilities arising from race conditions should be grouped together. However, we do *not* require that they be distinct from other vulnerabilities. For example, a vulnerability involving a race condition may require untrusted users having specific access permissions on files or directories. Hence it should also be grouped with a condition for improper or dangerous file access permissions.
2. Classification should be primitive. Determining whether a vulnerability falls into a class requires a "yes" or "no" answer. This means each class has exactly *one* property. For example, the question "does the vulnerability arise from a coding fault or an environental fault" is ambiguous; the answer could be either, or both. For our scheme, this question would be two distinct questions: "does a coding fault contribute to the vulnerability" and "does an environmental fault contribute to the vulnerability." Both can be answered "yes" or "no" and there is no ambiguity to the answers.
3. Classification terms should be well-defined. For example, does a "coding fault" arise from an improperly configured environment? One can argue that the program should have checked the environment, and therefore an "environmental fault" is simply an alternate manifestation of a "coding fault." So, the term "coding fault" is not a valid classification term.
4. Classification should be based on the code, environment, or other technical details. This means that the social cause of the vulnerability (malicious or simply erroneous, for example) are not valid classes. This requirement eliminates the speculation about motives for the hole. While valid for some classification systems, this information can be very difficult to establish and will not help us discover new vulnerabilities.
5. Vulnerabilities may fall into multiple classes. Because a vulnerability can rarely be characterized in exactly one way, a realistic classification scheme must take the multiple characteristics causing vulnerabilities into account. This allows some structural redundency in that different vulnerabilities may lie in the same class; but as indicated in 1, above, we expect (and indeed desire) this overlap.

The remainder of the paper is organized as follows. The next section reviews earlier vulnerability classification schemes, gives examples of vulnerabilities, and demonstrates they do not meet our requirements. We then consider a scheme for analyzing the implementation of a system with a proveably secure *design*. The fourth section analyzes this scheme to determine whether the result is applicable to systems that do not have a proveably secure design. We discuss experiments, and present several examples of the requisite analysis. We then discuss how to develop tools to detect vulnerabilities, or to prevent their exploitation (or both). We conclude with some open research issues and a summary.

The next section reviews the goals and characteristics of previous classification schemes, and explains why they are inadequate for our needs.

2. Prior Work

In the 1970s, two major studies classified security flaws. One, the RISOS study [1], focused on flaws in operating systems; the other, the Program Analysis (PA) study [2], included both operating

systems and programs. Interestingly enough, the classification schemes both presented were similar, in that the classes of flaws could be mapped to one another. Since then, other studies have based their classification schemes upon these results [3, 4]. However, the RISOS and PA schemes do not take the level of abstraction or point of view into account. Further, their classes are broadly defined, and non-primitive and overlapping (by the above requirements).

Aslam's recent study [5], as extended by Krsul [6], 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.

This section contains a review of the PA, RISOS, and Aslam classification schema. We then show that two security flaws may be classified in multiple ways under all of these schemes.

2.1. RISOS

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.

Some ambiguity between the classes indicates that one vulnerability may have multiple classifications; 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). This is a symptom of the problem with using these classes as a taxonomy.

2.2. PA

Neumann's presentation [4] 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)
    a. 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";
    b. 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

c. improper change – the “time-of-check to time-of-use” flaw; changing a parameter unexpectedly;

d. improper naming – allowing two different objects to have the same name, resulting in confusion over which is referenced;

e. 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;

   a. improper indivisibility – interrupting atomic operations (e.g. locking); cache inconsistency

   b. 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.

The problem with this classification is the breadth of the categories. Like the PA classification, vulnerabilities may fall into multiple classes, and the same vulnerability falls into different classes depending on the level of abstraction at which the analysis is conducted.

2.3. Aslam's model and Krsul's work

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

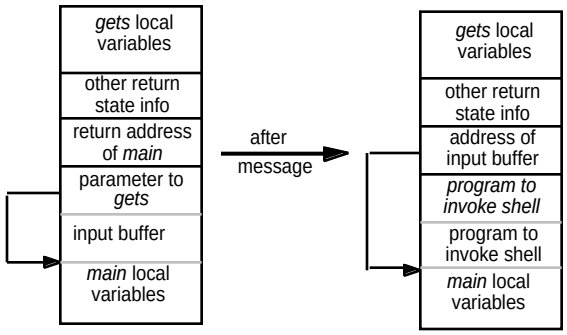
The classification scheme of this study is more precise than those of the PA and RISOS studies. It provides more depth for classifying implementation-level flaws ("faults" in the language of this study). This appears to meet our requirements at the implementation level; the scheme clearly lacks the high-level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

2.4. Problems with using these models

To motivate our discussion of problems, consider a buffer overflow attack.

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.

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.



This flaw depends upon the interaction of two processes: the trusted process (*fingerd*) and a second process (the attacker). 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 [8]; PSOS has 15 [9].) 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.

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 isw 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.

4.4. Summary of Problems with Previous Work

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.

3. Relationship of Vulnerabilities to Specifications

A *formal top-level specification* (or FTLS) is a mathematical description of a computer system. It states specific conditions that the system design and implementation must meet. Ideally, these specifications are shown to be consistent with (a mathematical statement of) the security policy that the system is to enforce. Systems such as PSOS, KSOS, and Gemini adopted this approach. The DoD Trusted Computer System Evaluation Criteria requires a formal design proved to conform to a FTLS.

Assuming the proofs are correct, the security of such systems breaks down at the level of code verification. Proving that the implementation of the system is correct and conforms to the design (implying, at least by transitivity, that the implementation correctly conforms to the FTLS) is beyond our current capabilities. Were it not, the most straightforward approach would be to express the constraints of the FTLS as mathematical invariants, and verify mathematically that the implementation did not invalidate the invariants at any time.

An informal version of this last step uses formal testing. Property-based testing checks to see that a program does not violate a given set of specifications [10-12]. What follows is akin to this process, but assumes that specifications are not initially available.

4. Vulnerability Analysis

Given that existing systems do not have FTLS, the above is an exercise in verifying systems with a proveably secure design. Very few systems meet this criteria. This section discusses the generalization of the technique to the much larger class of systems that are not proveably secure, or ideed even secure.

We should note that a general procedure to locate *all* vulnerabilities in *arbitrary* systems is an undecidable problem, by Rice's Theorem. So, we must focus on specific system details.

4.1. Informal approaches

The FTLS describes the security constraints of the system. Systems without a FTLS have a security policy, which may either be formal (written and specific) or informal (unwritten or ambiguous). In either case, certain rules emerge from the policy as security constraints. For example:

- A user must be authenticated to a new identity before changing to that identity.
- Data entered into a buffer may not be written beyond the end of the buffer.
- The user *bishop* may alter passwords of other users as well as her own.

The first two are part of an implicit security policy, because they are general to many sites. The third is an example of a site-specific policy rule that identifies a single user as special.

System vulnerabilities violate some set of these rules. In the formal methods, we mathematically determine what these rules are (from the FTLS) and determine formally if a program violates them. In this informal method, we determine what these rules are from the (explicit or implicit) security policy and determine experimentally if a program violates them. Unfortunately, many aspects of the system's requirements are implicit, such as the first two of the above three rules.

Our procedure is to identify the implicit and explicit rules that make up a system security policy. From this, we can derive a set of *characteristics* that identify properties of non-secure programs — specifically, vulnerabilities. We include conditions needed for these properties to result in a security breach.

An alternate view is that vulnerabilities arise as a combination of specific conditions of the environment and specific properties of the program. That is, characteristics can be deduced from the vulnerabilities themselves.This is akin to deducing the security policy of a system from an oracle that states whether actions are "secure" or "non-secure." It is however far more realistic than an attempt to deduce the implicit security policy from the way a system works. Most people will agree on what violates security. But they will disagree on whether an abstract statement of "secure," or a list of requirements for something to be considered "secure," is correct.

Definition: A *characteristic* of a vulnerability is a condition that must hold for the vulnerability to exist.

We represent each vulnerability by its set of characteristics, called the *characteristic set*. We classify vulnerabilities by the characteristics in their sets. This meets our first criterion, because similar vulnerabilities will be classified similarly due to the characteristics under consideration being in both their characteristic sets. It also meets our fifth criterion, as a single vulmerability may fall into multiple classes. The remaining criteria require a definition of the specific characteristics.

Definition: A set of characteristics is *sound* if the characteristics are independent; that is, given any subset of the set of characteristics, none of the elements in the complement can be derived from the subset.

For our purposes, we require that a characteristic set be sound. This simplifies classification and lead to minimal-sized characteristic sets.

Definition: A set of characteristics *C* is *complete with respect to a system CS* if the characteristic sets of all vulnerabilities of a system *CS* are composed of elements of *C*.

Finally, the characteristic set associated with the vulnerability that contains as few characteristics as possible is *minimal*.

Hypothesis. Every vulnerability has a unique, sound characteristic set of minimal size.

Call this set the *basic characteristic set* of the vulnerability.

4.2. Prevention of vulnerabilities, exploiting vulnerabilities

Let *CS* be a system with some set of vulnerabilites $V = \{ v_1, \ldots, v_n \}$. Let $C = \{ c_1, \ldots, c_k \}$ be a complete set of characteristics with respect to *CS*. Thus, each vulnerabilitiy v_i's characteristic set is a selection from the elements of *C*.

Let two vulnerabilities v_a and v_b have basic characteristic sets $C(v_a)$ and $C(v_b)$, respectively. If $C(v_a) \cap C(v_b) \neq$, both vulnerabilities have a common characteristic. As they are elements of basic characteristic sets, if this property fails to hold, vulnerabilities v_a and v_b will both be unexploitable because the conditions to exploit them do not exist.

Determining the similarity of vulnerabilities now becomes an exercise in set intersection. Consider two vulnerabilities, and define distance as follows:

Definition: Two vulnerabilities v_a and v_b are *n* units apart if $| C(v_a) \cap C(v_b) | = n$.

This suggests a procedure to locate vulnerabilities: look for characteristics. When detected, the condition described by the characteristic must be negated. Then not only the single vulnerability under consideration, but *all* vulnerabilities with the same characteristic, are non-exploitable.

We expand on this approach in the next section. First, we review our hypothesis and what we want to show.

4.3. Hypotheses

The hypotheses we wish to test are as follows:

- We can determine the basic characteristic set for any vulnerability.

  This will give us a set of characteristic conditions that cause the vulnerability to be exploitable.

- We can determine a set of characteristics for a set of vulnerabilities

  If we can, then we can generate characteristics by examining known vulnerabilities for a system. We can then use these characteristics to examine the system for more vulnerabilities, and proceed iteratively.

- The size of a complete set of characteristics for a system is smaller than the set of vulnerabilities.

  Intuition suggests that this is obvious, as the set of vulnerabilities is similar to the power set of the set of characteristics. However, not every combination of characteristics creates a vulnerability. Does this mean that a very few characteristics create a large number of vulnerabilities? Further, by defining characteristics appropriately, we may create a template for a large set of characteristics; an example of this will be given in the next section. In this case, analysis of a system for any characteristic matching the template may uncover many different instances of the template characteristic.

- Each characteristic suggests a tool to analyze the system or system programs to determine if the condition exists.

  If true, the implication is that a set of tools can be designed to look for, and close, vulnerabilities, even before thoise vulnerabilities are known to exist! This hypothesis suggests that the characteristics must be quite detailed (to enable automated or manual search).

5. Using the Hypothesis

Assuming our hypotheses are correct, we can use the analysis to eliminate vulnerabilities. We simply look for situations in which the characteristic conditions hold, and negate them. Some reflections on this follow.

Consider the issue of level of abstraction. One low-level characteristic might be "the **PATH** variable is not reset when spawning a subcommand." This can be checked easily and automatically. At a slightly higher level, the characteristic might read "the environment in which the subcommand will execute is not reset to a safe value before spawning the subcommand." This is broader, yet also susceptible to mechanical checking. At an even higher level, "The protection domain of a subprocess is not properly initialized" is a design flaw, yet clearly suibsumes the previous two. Thus, levels of abstraction are absorbed into the characteristics. (The next section contains more examples of this.) The characteristics are inclusive of lower-level characteristics, so there is a clear containment relationship. We hypothesize this relationship holds for complete sets of characteristics at different levels of abstraction. If so, this solves the first problem with the previous classification schemes.

Point of view is a bit trickier. Currently, we believe it must be an attribute of the characteristic. In essence, every vulnerability will then have 3 different characteristic sets, one for each of the three points of view. However, this area clearly needs more study.

6. Examples

In what follows, we present three vulnerabilities and discuss their breakdown. Throughout, we assume a UNIX system environment.

6.1. Data and Stack Buffer Overflows

The *fingerd* attack discussed earlier is an example of this. The breakdown is as follows:

C1. Failure to check bounds when copying data into a buffer.

C2. Failure to prevent the user from altering the return address.

C3. Failure to check that the input data was of the correct form (user name or network address).

C4. Failure to check the type of the words being executed (data loaded, not instructions).

Invalidating any of these conditions prevents an attacker from exploiting this vulnerability:

C1'. If the attacker cannot overflow the bounds, the control flow will continue in the text (instruction) space and not shift to the loaded data.

C2'. If the return address cannot be altered, then even if the input overflows the bounds, the control flow will resume at the correct place.

C3'. As neither a user name nor a network address is a valid sequence of machine instructions on most UNIX systems, this would cause a program crash and not a security breach.

C4'. If the system cannot execute data, the return into the stack will cause a fault. (Some vendore have implemented this negation, so data on the stack cannot be executed. However, data in the heap can be, leaving them vulnerable to attack.)

The sensitivity of characteristics to hosts is clear from C3 and C4. On some systems, ASCII text would be valid instructions; on these systems, condition C3 *always* holds. Thus, conditions C1, C2, and C4 would be relevant. On some systems, C4 is negated, so an attack like *fingerd* would not succeed. But some vendoirs approximate the negation of C4 by determining type from the location of the words in memory. Words on the stack are data and may not be executed. Words elsewhere may be instructions and so can be executed. If the words are data words not in the stack, then C4 holds.

Race conditions involving UNIX file accesses

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);
```

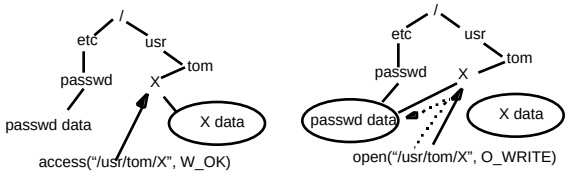


Figure 1a. Figure 1b.

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.

The race condition has the following conditions:

C1. Within the process must be an interval between the checking of a condition based upon a file name and an opening of that file using the file name and based on the file satisfying the condition.

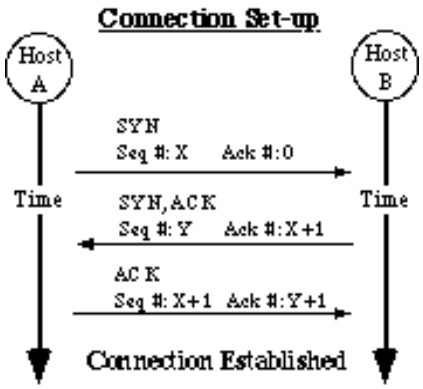
C2. An attacker can alter the binding of the file object to the name.

Characteristic C1 is very specific to file accesses. A more general version of this characteristic is to delimit the "window" of opportunity by any check on the file's attributes and by any action taken upon the file and based upon the prior check. However, generalizing C2 requires looking at general objects, not just file objects, or altering bindings to properties in addition to names. The specificity of C1, and its less specific version, demonstrate the levels of abstraction.

As with buffer overflows, the race condition is not exploitable if either characteristic does not hold. See Bishop and Dilger's paper [13] for a detailed discussion of race conditions arising from file accesses.

6.3. Internet address spoofing

Initiating a TCP connection has three steps, as illustrated [14]. Host A sends a SYN to host B; the SYN contains a sequence number X. Host B responds with SYN/ACK, supplying its own sequence number Y and incrementing A's sequence number. Host A ends the handshake with an ACK containing the sequence number Y+1. IP spoofing refers to a host N sending B the TCP initiation sequence packets, but with A's IP address as source address. B must ensure that A ignores, or never receives, the second message for if A does receive it, A will send an RST packet and the connection will be reset.



The characteristics of this attack are:

C1. The sequence number from the victim is predictable.

C2. The victim will never receive a reply packet from the purported peer.

Given these characteristics, the host N can impersonate the purported peer A. B's SYN/ACK will never be received, and N can predict what Y will be. Hence B will think it is communicating with A.

If C1 is false, then N can prevent A from receiving traffic from B, but N will simply have to guess what Y is. If C2 is false, then the first reply packet that B receives from A will interrupt the connection and cause a state transition that will terminate the (one-way) connection between N and B.

Note that C2 is quite general. In the best known instance of this attack [15], the attacker flooded the ports of the purported peer to prevent it from receiving the SYN/ACK packet from the victim. As an illustration of point of view, from the point of view of the purported victim, the vulnerability is:

C3. Too many incoming messages can block access to all ports for some finite time interval.

However, from the attacker's point of view, this is subsumed in C2.

6.4. Summary

These three examples demonstrate the characteristics of three well-known vulnerabilities. These also demonstrate aspects of the characteristics and of techniques that can derive from them. We now consider how to implement tools based upon these mechanisms.

7. Tool Development

Central to any classification scheme is to what use it will be put. Our goal was to develop a classification scheme that could lead to tool development. Given the analysis leading to the characteristics composing the vulnerabilities, one need only build tools to look for the characteristics.

In this aspect the layers of abstraction become relevant. If the tools are to be automated, and run with little guidance, then the characteristics must be very precise and well-defined. If the tools are to be run with much human intervention and analysis, the characteristics may be much broader. For example, the characteristic "An attacker can alter the binding of the file object to the name ' (§6.2, characteristic C2) is simple to check by a recursive scan of permissions of the UNIX file system; but the characteristic "Too many incoming messages can block access to all ports for some finite time interval" (§6.3, C3) is more difficult. Although one can automate a check for finite resources, analyzing the time-out strategy is considerably more difficult.

8. Conclusions

The research questions of this discussion are:

- Does every vulnerability have a unique, sound characteristic set of minimal size?
- Does every system have a complete set of characteristics small enough to make scanning for them, as opposed to vulnerabilities, advantageous (quicker, more effective, *etc*)?
- How do we determine the basic characteristic set for any vulnerability?

The development questions are:

- How effective are tools that look for characteristics at locating previously unknown vulnerabilities?
- How general can we make the characteristics that the tools look for and still require minimal human intervention?

9. References

1. Abbott, R.P., *et al*., *Security Analysis and Enhancements of Computer Operating Systems*, . 1976, Institute for Computer Sciences and Technology, National Bureau of Standards: Gaithersburg, MD.
2. Bisbey, R. and D. Hollingsworth, *Protection Analysis Project Final Report*, . 1978, Information Sciences Institute, University of Southern California: Marina Del Rey, CA.
3. Landwehr, C.E., *et al.*, *A Taxonomy of Computer Program Security Flaws.* Computing Surveys, 1994. **26**(3): p. 211-255.
4. Neumann, P.G. *Computer System Security Evaluation*. in *National Computer Conference*. 1978.
5. Aslam, T., *A Taxonomy of Security Faults in the UNIX Operating System*, in *Department of Computer Sciences*. 1995, Purdue University: West Lafayette, IN 47097.
6. Krsul, I., *Software Vulnerability Analysis*, in *Department of Computer Sciences*. 1998, Purdue University: West Lafayette, IN 47097. p. 171.
7. Seeley, D. *A Tour of the Worm*. in *Winter USENIX Technical Conference.* 1989: USENIX Association.
8. Djikstra, E., *The Structure of the THE Multiprogramming System.* Communications of the ACM, 1968. **11**(5): p. 341-346.
9. Neumann, P., *et al.*, *A Provably Secure Operating System*, . 1975, Stanford Research Institute: Menlo Park, CA 94025. p. 327.
10. Ko, C., G. Fink, and K. Levitt. *Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring.* in *Tenth Annual Computer Security Applications Conference*. 1994.
11. Fink, G. and K. Levitt. *Property-Based Testing of Privileged Programs*. in *Tenth Annual Computer Security Applications Conference*. 1994.
12. Fink, G., *Discovering Security and Safety Flaws Using Property Based Testing*, in *Department of Computer Science*. 1996, University of California: Davis, CA 95616-8562.
13. Bishop, M. and M. Dilger, *Checking for Race Conditions in File Accesses.* Computing Systems, 1996. **9**(2): p. 131-152.

14. Heberlein, T. and M. Bishop. *Attack Class: Address Spoofing.* in *Nineteenth National Information Systems Security Conference*. 1996. Baltimore, MD.
15. Shimomure, T. and J. Markoff, *Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw--By the Man Who Did It*. 1996, New York, NY: Warner Books.