Vulnerabilities Versus Exploits

Computer Measurement Laboratory
info@cmlab.biz

A great deal of attention has been given in the computer security world to the notion of software vulnerabilities. At best, this focus is misguided. At worst, it is a real obstacle to the solution of the computer security problem.

There are many different definitions for the term vulnerability. They are all equally ambiguous from the standpoint of measurement. To disambiguate this issue let us turn to the field of software reliability. The fields of software reliability and software security share a great deal of commonality. In the field of reliability, a vulnerability would be called a software fault. When a fault is encountered at run time the software would fail. This failure event is represented by a departure from the specified behavior of a program.

To clarify this discussion some definitions would be in order. People make mistakes when they are recording their thoughts. These mistakes are called ***errors***. There are two types of errors. There are errors of omission wherein something that should have been stated was not. There are errors of commission. In this case, something was stated incorrectly.

The result of a designer/developer making an error is a ***fault***. This fault is the physical manifestation of an error. Faults come in different sizes. In some cases a simple token in the language of definition was incorrectly recorded, say a plus sign was written down when a minus sign should have been written instead. Alternatively, a whole predicate clause could have been screwed up. In which case multiple tokens would have been inappropriately recorded.

When a fault is encountered at execution time, we will say that the program has ***failed***. The failure event represents the moment when the fault was executed. The execution consequences of the failure may vary wildly. The program may crash immediately. It may slowly begin to decay, as is the case with a memory leak. Or, it may simply lose the capability of expressing one or more of its operations. Unless we are specifically watching for them, most failure events will go unnoticed for a very long time.

Let us now turn our attention to the etiology of the faults. They can occur at three distinct places in the software development process. A fault might be introduced at the requirements stage. This would be the case if the customer articulated the need for a red warning label to be posted and we inadvertently wrote down that a green warning label be posted. The program could be designed correctly and implemented correctly according to the design. But the moment that it showed a green display it would have failed because that was not what the customer wanted.

Alternatively, we could well have captured the customer's requirement for a red warning label during the requirements formulation. When it came to the design of the program, a designer might misinterpret or (what's worse) alter the specification to have the program produce a green warning label. The program could be written to implement this new design quite correctly. However, the moment that it produced a green warning label, the program will have failed.

Finally, the designer could well have implemented the red warning label requirement in the design of the program. When it came time to write the program, the programmer could have accidentally written the wrong numeric code for red into the program so that it would produce a green warning label instead of a red one.

The end result of all three of these different faults will be the same on the final program. It will fail. The failure event, however, will have three different etiologies. If we are to improve our development process, it will be vital to understand where and how the faults are being introduced.

In the last analysis it really doesn't matter how many faults there are in a program. What does matter is the fact that one or more of them might get executed during the normal course of the execution of a program. We may look at a program as if it were a road map. Each road on this map represents a possible execution path in the program. Based on his/her needs each user of the program will select a subset of these roads to travel.

If a fault lies on a back road or a cul-de-sac that is never traveled by a user, then the program will never fail for that user. Another user might choose to go down the road that contains the fault during his/her normal travels. They will always find the fault and the program will always fail for them. Thus, we can see that the user's perception of the reliability of a program will depend on the roads that they choose to travel and not on the fault burden itself.

When we test a program, we are really attempting to drive those roads that we think that the user will travel. If we have a good test program, our subset of tested roads will match that of our potential users very well. If the user's subset of roads is different from those that we have tested, they may well stumble across faults that we did not discover in test.

The important thing, here, is that we can actually measure the roads that a typical user travels. We can characterize the use of a program by the frequency with which each user traverses a particular road segment as they travel through the program. If we are watching we can know that a typical user is traveling through the map as we expect that they should. We can also know exactly when they begin to explore new territory.

Now we get to the essential differences between the study of reliability and the study of computer security. In the reliability context user of a program will select their subset of the roads from the program road map based on their needs. In the security context, however, the user is motivated to find and execute the faults (vulnerabilities). A hacker, for example, might well know of a fault (vulnerability) on a particular road segment based on his analysis of the source code. In which case, the hacker will cause the program to move to a road segment that is not part of a typical or certified execution pattern.

If we know how a program is normally used in the context in which it is deployed we can, in fact, certify a road map for its execution in this context. We can know that it will not fail in this context in that we have thoroughly tested in it this context. We can also easily monitor in real time the activities of users to insure that they do not depart from the certified use of the road network. A hacker will behave in a manner very different from a normal user. He will be instantly recognizable because of this. In that the hacker will vector his activity directly to the fault that he knows is in the system, he is readily identified. As result, he is easily defeated. No one can deliberately misuse or program if we are watching. It is just that simple.

Now let us return to the notion of the failure event. An ***exploit*** is a deliberate attempt by a hacker to execute a fault. In other words, an exploit is a failure event. In a reliability context, the failure event is an accidental event. In the security context, a failure event, exploit, is a deliberate event. The thing that both of these two different types of failure events have in common is that the user has departed from a standard or validated use of the road system.

No sensible engineer would ever consider the prospect of designing and building a defect free bridge, building or automobile. It would be at once virtually impossible and/or prohibitively expensive. Rather these items are designed to function well within a known range of reasonable defects (faults). That is what good engineering is all about.

The computer security literature, today, abounds with published attempts at building a vulnerability (fault) free program. This effort is indeed a fool's errand for a number of reasons. First and foremost among these reasons is that modern software systems are very large. They typically contain several millions lines of source code. As a result they are astonishingly complex. It is virtually impossible to build a road map for any of these programs much less probe for faults along all of these road segments. Programs also change very rapidly. It is not uncommon for 10% or more of the code base to change from one build to another. This also means that the road map is changing very rapidly as well. As new features are added to programs, brand new opportunities arise for new fault (vulnerability) types to be created in this new environment.

We can never know how many faults there are in a program. It is impossible and impractical to pretend that we might accomplish this task. It is really not even important to have this knowledge. There are a couple of things that are important to know, however. First, we need to build an accurate road map represent the actual or certified use of the program. Then can test the program for faults on this map and pretty well assure ourselves that it is reliable in this context.

Software vulnerabilities (faults), then, should not be a central focus in securing a program. We must learn to monitor the activities of the program when it is running. An exploit will measurably alter the nominal execution of a program. It can readily be detected and controlled. Software vulnerabilities can be noted when they occur and studies made as to their etiology so to improve our software development process. The immediate concern in computer security, however, should not be on the vulnerabilities but on the exploits.