Testing Protection in

Two Web Access Control Examples

Fred Cohen and Pat Leary
Sandia National Laboratories
November, 1996

0.1 Background and Introduction

We believe that protection systems are often implemented without adequate testing. One of the side effects of inadequate testing is the large number of protection failures we see in day-to-day operations of systems exposed to hostile environments like the Internet. The literature in information protection has few protection testing examples, and of those examples, still fewer provide helpful guidance in performing definitive tests on real systems. [8] [4] [3] [7] [9] [1] Because of the lack of available guidance in the information protection literature, and in order to address testing issues in situations requiring high assurance, we have taken an approach based on devising methodologies similar to those in common use in computer engineering [2] and software testing. [6]

0.1.1 Two Web-based Protection Testing Examples

The two applications we examined were Web servers augmented to include access controls. Two access control methods were tested; Unix-based access controls, and custom subject/object access controls.

- Operating system protection is often preferred for high assurance because it is based on the same underlying mechanisms used to protect the application itself from subversion. If operating system protection can be violated, then the application is not safe, nor is any special-purpose protection mechanisms the application may apply. In addition, operating system protection is far more widely used and stressed than application protection. As an example, Unix-based access controls are generally more trusted than Netscape's Web server access controls.

  In the Unix-based Web access control system, a uniform set of User identities are provided across all Unix platforms. A user is given a user number (e.g., *12345*) and name (e.g., *abcdef*) on an organizational basis and has the same user identification number and name on all platforms within the organization. Organizational security servers (e.g., Kerberos or X.509 certificate servers) then provide cross-platform and application-level authentication to any service wishing to use the central capability.

- The subject/object model of access control was identified as a key component of information security quite some time ago and has been in widespread use for over 25 years. [5] In order to provide understandable access controls over areas within secure Web servers, an implementation of a subject/object-based access control mechanism was implemented.

  In the subject/object Web access control system, subjects are identified by the same organizational security servers (e.g., Kerberos or X.509 certificate servers), objects are Unix directory areas, and access is controlled by an access control file stored on the Web server.

0.1.2 Some Background on Protection Testing

While the basic methods used in hardware and software testing are often valid in the protection testing arena, some of the main lines of thinking about testing in general do not apply to protection testing in particular. Two of these lines are; (1) the line based on assumptions about recurrence rates of detected faults, and (2) the line based on the assumption that systems under test are of substantial size. Specifically:

- In the hardware and software testing literature, the goal is often stated in terms of reducing the frequency of failures to below some rate. For example, in hardware testing, the dominant goal is to achieve a high mean-time-to-failure of the overall system, while in software testing, the goal is to reduce the failure rates to a level where the average user rarely experiences a failure. It is generally assumed that once a fault is found, it will not be exercised as often because people will avoid it.

  For example, if the spell checker in a popular slide presentation system causes system crashes when used on large files, users eventually learn not to do spell checking on these files. This means that if often exercised faults are removed, program functionality is adequate for most users, and products are usable. In a highly competitive software market such as the one that exists today, testing delays time to market. But completely untested systems are unusable in practice. There is therefore a balance between time to market and testing which is optimized when the number of failures found in normal operation is below the tolerance threshold of the users. The large amount of poorly tested software in the market today has the effect of lowering user expectations, which in turn allows even less complete testing. On the other hand, the increased complexity of software means that more testing is required in order to achieve the same failure rates. Companies in the software business are well aware of these issues and do mathematical analysis to help them optimize the amount of testing they do relative to the return on investment.

  Unfortunately, in information protection, this model does not accurately reflect the reality. Attackers seek out faults instead of avoiding them and put forth a great deal of time and effort to find ways of creating desired failures through those faults so as to achieve their goal as attackers. Once a fault is found, it is likely to be exercised with increasing frequency over time until it is fixed. One fault is often exploited in order to create other faults which are planted in order to allow subsequent reentry or other harmful side effects. The economics of attack are such that a single fault can lead to unlimited compromise.

- Another common assumption made in hardware and software testing is that the size of the hardware device or software package that must be tested is too large to attain very high coverage except under very limited fault models. Exhaustive search is almost never feasible, and coverage of data-dependent faults or race conditions is very hard to do. In timesharing systems and multiprocessor environments, even in systems requiring extremely high levels of assurance, it is virtually impossible to do enough testing to assure high coverage of all faults.

  In information protection, this assumption does not always have to be made. The most secure systems are secure because they use a very small hardware and/or software base that provides protection through the use of a well-defined protection mechanism. For example, the security kernel of a relatively secure operating system can be written only a few hundred lines of Pascal, and a Web server with provable security properties has been written in 80 lines of C. In most cases, the security relevant portion of a system can be implemented in a well-defined, small, and highly testable module.

The combination of higher assurance requirements and smaller size make protection testing a very different exercise than general hardware and software testing. The economic tradeoffs are quite different, the types of tests that are most revealing are different, and the goal is different.

0.2 Unix-Based Web Access Controls

Users accessing the system under test (SUT) are granted access to files based on; (1) their Kerberos or X.500 authenticated user identification number (UID), (2) the group identification numbers associated with that UID (GIDs), (3) the name (FILENAME) of the file they are attempting to access, (4) the information node associated with that file (INODE), (5) the protection settings (SETTINGS) stored in that INODE (Read, Write, and eXecute access by each of Owner, Group, and World r,w,xxo,g,w), (6) the ownership of INODE (OWNER), (7) the group ownership of INODE (GROUP), (8) the OWNER, GROUP, and SETTINGS, for all directories in the path between the root of the filesystem and INODE (PATH CONDITIONS), and (9) the contents of the */etc/passwd* (PASSWD) and */etc/group* (GROUPS) files used to associate users with UIDs and groups under Unix. The access control decisions are made by a program (ACCESS) which was modified only with respect to program interface to facilitate testing.

0.2.1 The Input Space

The variables involved in the access control decision are then; UID, GIDs, FILENAME, INODE, SETTINGS, OWNER, GROUP, PATH CONDITIONS, PASSWD, GROUPS. In the SUT, there are 65534 possible values for UID, 65534 possible values for each GID and a maximum of 15 simultaneous GIDs, 256^{256} possible values for FILENAME, 65534 possible values for INODE, 512 possible values for SETTINGS, 65534 possible values for each of OWNER and GROUP, the product of these values for each directory in PATH CONDITIONS, and an essentially unlimited number of variations on the PASSWD and GROUPS file contents. Numerically, that comes to a total of $5 * 10^8 21$ possible input combinations for each element in the path, ignoring the file content issues. This is clearly far too many combinations to test exhaustively.

By way of practicality, in testing ACCESS, it was found that, on a 200Mhz processor with a 10ms disk drive and a substantial disk cache, 512 tests could be performed in about 30 seconds. This comes to about 1.5 million tests per day. Since tests are independent of each other, testing is trivially parallelized, but even with 100 computers operating in parallel, 150 million tests per day is the best that can be achieved. It is also important to note that protection system operation may vary with operating system version, hardware, and configuration-specific information. For that reason, we can only use nearly identical systems for valid testing of this sort.

0.2.2 An Approach

It seems clear that any tests we perform will exercise only a very small portion of the input space. It is therefore critical that we find revealing fault models in order to attain substantial coverage within reasonable runtimes.

The simplest assumption, and one that gains a great deal of reduction in complexity, is the assertion that permanent faults in the ACCESS program result from values associated with one variable only. That is, we can assume that there are no interactions between variables. If this were true, we would only have to try one example for each value of each variable, or $65534 * 5 + 512 + 256 * 256 = 393728$ tests. This test can easily be completed in only 5 hours per directory level, or one day for all 5-deep directory structures.

Unfortunately, this assumption is certainly not valid in all cases. For example, we know that protection settings work differently for the *root* user (UID=0) than for other users and that GID interacts with SETTINGS and GROUP to determine accessibility only if the interaction between UID, USER, and SETTINGS is not decisive in this instance.

Despite the general invalidity of this assumption over this entire problem space, it may be valid in some subcases. For example, we may assume that *read* access is independent of *write* access and *execute* access, and thus develop tests which ignore all but *read* access. This reduces the problem space by a factor of 64, but if our assumption turns out to be invalid, our results may be misleading.

One way to deal with the challenge of making such assumptions is to perform testing to validate the assumptions assuming independence of this assumption from some other assumption. For example, we may test the assumption that *read* access is independent of *write* access by assuming that this aspect of the access control decision is independent of pathnames. We can then test the assumption of *read* independence from *write* on one pathname by exhaustion. We might also want to try to verify that the independence from pathname assumption is correct by trying a subset of the settings of *read* and *write* access over a substantial a number of different pathnames, or perhaps by exhausting the *read*, *write* SETTING space for each of a few hundred different pathnames.

A different approach taken in many cases in computer engineering is to create conditions for testing a particular variable by setting up an environment in which that variable is the key to the decision and then trying to exhaust the basis for that decision.

0.2.3 The Protection Challenge

Unix file read protection is based on a set of 9 protection bits, the owner and group identities associated with the particular file, and the effective nd real user identity and group identities of the user attempting the access. Different implementations of Unix handle group access differently, the order of evaluating user, group, and world privileges differs from system to system, and there have even been time lag problems associated with cached protection information in networked access control systems. [4]

There are no approved or universal methods for determining accessibility of a file under Unix other than logging in as the user under consideration and trying to access the file of interest.

In a network setting, this is complex, time consuming, and difficult to program properly. As an alternative, we devised a program that checked access by running as *root*, forking a process, setting group privileges to contain all groups specified in the information contained in the */etc/group* file, setting the user ID according to the user ID contained in the */etc/passwd* file, and attempting to open the file for read access.

0.2.4 The Testing Methodology

While this would seem to be a sure way to determine whether or not a user could read a file, we wanted to find a way to test this security-critical program to make certain that it functioned as we believed it should.

In order for the test to be valid, we decided that it should contain no common code with the proposed program. By having two independent programs performing the same task, we felt it would be less likely that some common failure would result in both being wrong. We further decided that a theoretical test, wherein we wrote a program that provided the theoretically correct answer, would be less useful than a test that actually tried to access real files while logged in as the real users and set to the real groups.

In essence then, we decided to take two independent methods for getting the same result, one performing the actual function and one performing the quicker analysis function to be used in the implementation, and compare the actual capability to the predicted capability. If they always agreed, we would have a reasonable assurance that the analytical capability was operating correctly.

Fault Model 1

In keeping with the common methods of testing used on computer hardware, we adopted a fault model based on permanent stuck-at faults. We assume that any fault in the algorithm (i.e., a difference in the output of the algorithm and the true operation of the system) would recur every time if the same conditions were applied. We further assumed that the only factors in making an access control determination for this fault model were:

> The set of 9 file protection bits (each of read, write and execute for each of owner, group, and world) the owner and group identities associated with the particular file, and the effective and real user identity and group identities of the user attempting the access.

We pause to note that testing for all 9 protection bits (512 settings), for all ownerships of the files (up to 65534 of them), for all group ownerships of the files (up to 65534 of them), for all user identities that could attempt access (up to 65534 of them) for all groups combinations that those users could have (65534^n of them for Unix systems permitting n simultaneous groups) would require $512 * 65534^{n+2}$ test cases. The value for n is typically in the range of 15 for Unix systems supporting multiple simultaneous groups, which brings us to a total of about $9 * 10^7 4$ tests. In this test, we assume that file names are not meaningful in making any protection decisions.

Practicality limits us here, so we make additional fault assumptions in order to design a feasible test set. In particular, we believe and note that, with the exception of some special cases, the difference between any particular user and any other particular user is unimportant to the operation of the mechanisms under consideration. We have examined typical Unix protection decision code and the code used in the decision system under test and believe that, with the exception of a small list of known exceptions, there is no difference.

Operating system differences vary from version to version, but typically include differences between the *root* user (user 0) and other users, users with user identity numbers below 100 (or in some systems 500 or 512), group 0, groups with group numbers below 100 (or 500 or 512 on some systems), negative user numbers, and negative group numbers.

Similarly, we believe that different sets of group permissions do not interact other than the case where a user is in a group or not in a group that is the same as the group of the file being considered. This eliminates the need to test for all possible group combinations.

If these assumptions are good, then effective testing becomes far more feasible. In particular, the number of tests moves to something more like 512 permission settings, 20 ownerships of the files, 20 group ownerships of the files, 20 user identities that could attempt access, and 100 group combinations that those users could have. This then comes to only $512 * 20 * 20 * 20 * 100$ tests, or 409,600,000 test cases. In a quick performance test, it was found that, on the system being experimented with, 512 tests took about 35 seconds, which means that on a single computer, the entire test set could be run in 8,000 hours, or about 333 days.

Optional Assumption Set 1

A further assumption that would appear to make sense for file access controls under Unix would be to ignore the write and execute file protection bits for analyzing read permissions. While it may be possible to exploit execute or write permission to get read access to a file, this is beyond the scope of the system currently being considered, and we will ignore it. This assumption leads to only 8 access control settings instead of 256, or a 64-fold reduction in time - reducing the time to test to only about 5.2 days.

Optional Assumption Set 2

While this assumption seems reasonable, it seems to us a risky assumption in that, if we are wrong, improper access may result. For that reason, a different test can be used to partially test exhaustively for all protections settings ... is used for testing out all combinations of users and groups. In particular, we assume that either the owner, group, or world protection bits are actually used in any particular protection decision, and therefore, that we need not test all combinations across this boundary.

If this assumption is right, we can try all combinations of user settings with group and world access set to disallow all access, all combinations of group settings with owner and world access set to disallow all access, and all combinations of world settings with owner and group access set to disallow all access. This comes to 24 access control settings, which allows the test to run in about 15.6 days.

Although this time may seem too great for such a test, there is a parallel processing facility available at our test site that includes 10 computers of each of several sorts. Since each of the the protection tests are independent of all the others, it is trivial to partition the problem for execution on parallel machines. Results should be identical so long as those machines are identical in operating system version to the system under test.

Fault Model 2

The astute reader will notice that in a real environment, the set of all directory permissions, owners, and groups between the particular file and the root of the Unix directory tree might also be involved in a proper protection decision.

As in Fault Model 1, we adopted a fault model based on permanent stuck-at faults. We assumed that decisions based on directory structure are independent of decisions based on file access, so that once file access controls were confirmed, we could independently test directory access controls. We again assume that any fault in the algorithm (i.e., a difference in the output of the algorithm and the true operation of the system) would recur every time if the same conditions were applied. We further assumed that the only factors in making an access control determination for this fault model were:

> The set of 9 directory protection bits (each of read, write and execute for each of owner, group, and world) the owner and group identities associated with the particular file, and the effective and real user identity and group identities of the user attempting the access.

0.2.5 The Test Method

Testing was performed by isolating the protection-relevant code and customizing a small software test-bed to exercise the code as desired. Some of the software used for testing is included in the appendices to demonstrate particulars. Environmental conditions required for proper operation of the protection-relevant code are generated by setting appropriate environment variables prior to entry.

In order to increase the likelihood of finding faults, a grey box approach was used in testing. The code under test in this case was written in Perl and the Perl interpreter was treated, more or less, as a black box. The Perl code used to implement protection was treated as a white box. We examined each line of Perl code in terms of the input variables that could effect it and results coming from it, and tried to create input conditions that could cause unanticipated behavior.

This approach led to some important efficiencies, but also leaves a great deal of the space uncovered. For example, if a particular input variable is not used in a particular line of Perl code, values for that variable don't have to spanned in order to detect unanticipated behaviors in the examined line of code.

On the other hand, this ignores the potential for subtle interactions between variables that might vary depending on the internal operation of the Perl interpreter. For example, if a variable used to store more than 1K of data were stored differently than a variable storing less than 1K of information and the storage allocator had a boundary condition error with data of exactly 1K in length, and a 1K data value for one variable could effect the value of another variable of less than 1K in length, this fault in the Perl interpreter would probably not be revealed by the testing techniques we used, even though it could result in misoperation of the protection program.

0.2.6 Test Results

When we first started testing, we found several errors in the program that had originally passed simple functional tests. The functional tests had included trying the program on a few examples. (e.g., setting some different protections for files owned by one or two users and verifying by inspection that results were correct)

As we went to more thorough tests, we found more errors, both in our testing algorithm and in the program it was intended to test. For example, we were surprised when, after running successfully for user identification numbers 0 and 1, the test revealed errors on user number 2. We didn't think there should be any difference between user numbers 1 and 2, and indeed there should not have been. An error in the way we were testing turned out to be responsible.

Increasing rigor yielded still more errors. In order to sweep different dimensions of the test space quickly, we used assumption set 2 and quickly found that our access control routine permitted access when files were not readable if the user number being tested was not a user number in the password file. The error was traced to a system call that returned a null pointer when a UID didn't exist. This in turn was incorrectly interpreted as a 0 by another routine which used that as the UID of the user requesting access. This in turn generated a through reexamination of the program for other similar possibilities.

This particular failure was especially important in the application under test because it is a remotely accessed system using authenticated user IDs associated with an X.500 or Kerberos server. Since most authenticated UIDs will not exist on the system being accessed and the error allowed access when it should have been denied, it could have granted unlimited access only to unauthorized users. Similarly, a user who had somehow broken into the system might have attained access using such a user ID. Another case where this might happen would be when a setUID program is loaded in vendor software with a UID that is not locally defined. This is quite common, for example, when software is installed using *tar*.

One question that has to be answered for each application is whether to grant such a user *world* access or whether another level of *universe* access should be defined for undefined remote users.

In our next round of tests, we decided to exercise the FILENAME space by keeping PATH CONDITIONS, GROUP, OWNER, SETTINGS, UID, and GIDs constant. We ran the *try2* test program listed in the appendices to exercise some typical FILENAME variations intended to stress interpretation methods used in many Unix applications. We quickly came across several errors in the way parameters were passed between program elements, some of which could have caused a user to gain unlimited access to the server without triggering any alarms or leaving any audit information reflective of the attack.

At this point, we stopped testing the Unix-based access control system and went on to another method that was in use in several internal applications.

0.3 The Subject/Object Web Access Control System

Users accessing the system under test (SUT) are granted access to files based on; (1) their Kerberos or X.500 authenticated user identification number (UID), (2) the name of the area of the file system (DIRECTORY) they are attempting to access, and the content of the authorized subject list contained in the CONTROL file associated with the DIRECTORY. Administrators accessing the system under test (SUT) are granted access to administrative files (ADMIN) based on an administrative access control setting in the CONTROL file.

The variables involved in the access control decision are then; UID, FILENAME, INODE, CONTROL, CINODE, SETTINGS, CSETTINGS, OWNER, GROUP, PUID, PGIDs, and PATH CONDITIONS. In the SUT, there are 65534 possible values for UID, GUID, and PGID and a maximum of 15 simultaneous PGIDs, 256^{256} possible values for FILENAME and CONTROL, 65534 possible values for CINODE and INODE, 512 possible values for SETTINGS and CSETTINGS, 65534 possible values for each of OWNER and GROUP, and the product of these values for each directory in PATH CONDITIONS. As in the earlier case, the number of possible input combinations for each element in the path is too large for meaningful consideration and, as above, this is clearly far too many combinations to test exhaustively.

Unlike the previous example, this SUT also provides a graphical interface (GTSADMIN) for controlling the CONTROL file, and that too was considered in our testing regimeΦe. In particular, the program that alters the CONTROL file has inputs consisting of an arbitrary number of bytes of data (UPDATE) sent from the remote Web browser and the CONTROL file.

0.3.1 How We Tested

The Subject/Object Web Access Control System has two components; the administrative component that helps manage the access control lists through a Web-based graphical user interface, and the access control component that allows or prevents access based on the access control list. These components act independently, except as they interact through the CONTROL file, and were tested separately.

The administrative functions were tested by simulating the Web environment and generating inputs to the administrative program. The access control decisions were tested by simulating the Web environment and generating inputs to the control program. As in the previous example, tests were performed using gray box techniques where the Perl programs were white boxes and the Perl interpreter and Web environment were black boxes.

0.3.2 Testing Administrative Functions

As a first step in testing the administrative functions, we separated the interface issues involving the use of the World-Wide-Web as a means for administration from the issue of what was done with the information provided by the Web interface. By examining them separately, we were able to provide for automated testing. When a fault was detected in one module, we verified that the fault could become a failure by testing the combined components.

In one of our first attempts to separate functions, we emulated the Web-based calls to GT-SADMIN by writing a script that set up appropriate environmental variables and simulated a Web-based *get* incoming from an administrator. Much to our surprise, this immediately produced a CONTROL file that would allow no further administrative control by any user. When we examined the code, we found that any input other than *GET* (all upper case at the beginning of a line and followed by a separator) by any authorized administrator would remove all administrator access. This was tracked to the following (approximate) conditional expression:

```
if (REQUEST_METHOD eq 'GET') Post_Authorized;
else Process_Authorized;
```

Any action by an administrator other than *GET* results in the automatic assumption that this is an update to the access control file and that the input includes all of the new settings to this file. There is no other error checking in the administrative process whatsoever. It also turns out that there are some Web-based browsers that produce *get* instead of *GET* in their requests, and any such browser wielded by any authorized administrator would immediately produce this denial of services situation.

Testing that spanned UID values revealed that UID values not listed in CONTROL are sent the message *Denied-unauthorized* while users without administrative access but who are listed in CONTROL are sent the message *Not-authorized*. This would allow an attacker to differentiate their status without attempting a normal access.

Based on known browser characteristics and the method used to update CONTROL, it was theorized that a Trojan horse in a Web page could cause a legitimate administrator to arbitrarily change CONTROL without knowing that the change had been made. Previous testing demonstrated that this attack should work but we have not yet had the time to test this on the live system. Because this flaw is fairly obvious, repairs are being considered even without demonstrating the specific fault. It is believed that a user capable of this action would be able to grant themselves access, take or alter information as desired, remove themselves from the list, and leave no obvious trace of their activities.

This last comment bears some additional discussion. During the testing process, we noticed that no audit records are generated by the protection mechanisms in place at this time. This introduces obvious issues of being unable to assess damage after an attack or detect an attack through audit analysis.

As faults were found, the repair process was started. Since repairs might induce different faults or remove existing faults, we suspended testing during repairs. As of this writing, we have several suspicions about the last version of the software that have not yet been tested. They are:

- Suspicion: A Web administrator may be able to access the whole server by sending preformatted access control file contents as data fields in an administrative update. This would mean that any administrator could cross need-to-know boundaries, and perhaps gain unlimited access to the server.

- Suspicion: We suspect there may be a limit on the size of the access control file based on the maximum size that can be sent to the administrative interface by the Web browser. Near this limit, control file contents may become corrupt and arbitrary behavior may result.

- Suspicion: A non-existent control file may produce undesired results. No checks are made, and memory contents prior to access may affect resultant control decisions.

Many other faults may exist, but tests have been suspended for now pending available time, resources, and repairs of existing faults.

0.3.3 Testing Access Control Decisions

Based on code examination and knowledge of common Web script faults, we suspected that passing pathnames containing .. would allow access to areas outside of the normal Web server's access areas. Testing confirmed that we could retrieve the password file by requesting files with names like *../../../../etc/passwd* where the number of *../* entries is varied until the password file is revealed. The same procedure will produce any file on the system if a proper path can be discovered.

While the particular version of the particular server in use in the experimental system was configured to prevent such filenames, other servers, other versions of this server, and different configurations of this server were found to not provide this protection. No documentation was provided to assure that proper configuration was used, and no tests were performed to assure that the incoming request didn't include such a name.

We also note that since the pathname is passed as an argument to the access control program, this means that the server passes different data than is input from the user to the server. We don't know what else might be altered in the passing of arguments and did not have time to test it.

The control file was found to use the *up-arrow* character to separate fields. It turned out that the use of this character in names of users or files could lead to incorrect decisions and file information. This was discovered in an exhaustive test of one-though-four character user and file names.

There is no checking to prevent matches if one user name is a precursive substring of another user name. For example, a user *joe* could access information available to a user *joel* This was detected in an exhaustive multicharacter test of UID values.

Exhaustive tests of multi-byte inputs also produces results that character codes 10 and 0 were treated as end-of-string characters in FILENAME and UID inputs. This is to be expected for byte-value 0 because this value is commonly used as an end-of-string value in so-called ASCIIZ strings. We don't know what is done with the remaining input string, however, if it is not treated as part of the string for memory deallocation purposes and it is treated as part of a string for memory allocation purposes, this produces a possible memory allocation problem that might be exploited to cause an uninitialized string to take on an improper value. The byte value 10 is the character code for *linefeed* which is the *newline* character in Unix. This should be a valid byte in a filename but apparently is not treated that way by this version of Perl. Again, we are unsure of the possible side effects of this fault.

Another side effect of the short-length input space exhaustion test was the possibility of an unnamed user (for example a user whose user name corresponds to the *space* character) being mistaken for a legitimate user if a *space* entry in the access control file exists. For example, if an administrator accidentally added a user with the *space* character as the user's name, it would appear to be an empty entry in the management interface while producing access for a user who could get an authentication error to produce a *space* as a a side effect.

One of the standard tests based on known flaws in server-based programs is buffer overflows leading to errant program behavior. In our long-input tests we found that long query sizes caused dramatic service impacts. For example, a 1024 byte filename caused the load average to dramatically increase, response time for the whole system got very poor, and other undetected side effects may also have occurred. While the Web server in use at this site would effectively limit this length to 2048 bytes, it is disconcerting that such a performance degradation should result from such simple input. It also appears that arguments to the Perl script may be lost in long input situations, and this could result in incorrect requests being processed.

Another common flaw, particularly in Perl scripts, is the use of remote input without syntax checking. The particular Perl script under examination uses FILENAME without any checking, introducing the possibility that Perl could be induced to perform illicit functions based on special characters in FILENAME. It turned out that in the particular script under test there is apparently no such fault. Upon discussions with the authors, we found that the lack of this fault was purely an accident of coding and was not due to intentional diligence. Other versions of this program formed by modifying it to add or alter function were found to have this fault.

Examination revealed that unknown file extensions are presented to browsers as being of *html* type. We believe this is not a valid assumption, but we see no obvious protection implications in terms of server security.

As part of a test to exercise all program branches, tests were performed on files that existed and were accessible, files that did not exist and were not accessible, files that existed and were not accessible, and files that did not exist but would have been accessible if they did exist. The tests revealed that non-existent files were reported to legitimate users as empty html files. This is not a correct result, but has no obvious protection implications.

Finally, we believe that it may be possible to exploit a known TCP/IP option for setting remote environment variables so as to forge the remotely specified REMOTE_USER variable. If this could be combined with a denial of service attack on the Kerberos authentication facility, this might cause a wrong REMOTE_USER value to be used in access control decisions. If this could be done once, it could be used to grant administrator access to an attacker. We were unable to test this idea due to a lack of time and resources.

0.4 Summary, Conclusion, and Further Work

This paper described an approach taken to performing protection tests on two Web servers. It was an early and a crude attempt at protection testing, and yet we believe it was quite revealing in several ways:

- On average, we were able to identify one catastrophic fault for every four hours of testing we performed. Considering that these protection mechanisms had already passed common functional tests, this indicates that the functional tests were not revealing relative to protection requirements.
- We used several techniques commonly used in hardware and software testing, but we used them in ways they are not usually used in these fields. This would indicate that some of the assumptions upon which functional testing is based do not apply to protection testing, but that the basic techniques designed to reveal faults are effective if properly applied to the task of revealing protection faults.
- We were unable to perform thorough testing because of a lack of time and resources, however, it is clear that the return on time spent testing was excellent in terms of vulnerability detection and removal.

Perhaps the most interesting conclusion we draw from our experience is unrelated to the testing results. The programmer who was maintaining the tested code identified many potential faults each time we brought out a fault by testing. It appears that programmers who have a sincere desire to find faults and fix them and who are exposed to these sorts of tests come to understand more about protection and how to program with protection taken into account. It is also a somewhat humbling experience to have such flaws found.

We also conclude that more protection testing and research into protection testing may be helpful in finding and removing protection faults from programs. Clearly, further work is called for.

Bibliography

[1] M. Bishop and M Dilger, *Checking for Race Conditions in File Access*, [This paper describes an analytical test of software to detect possible race conditions that could lead to security failures.]

[2] M. Breuer and A. Friedman *Testing...*, [Breuer's book on testing]

[3] M. Chung, N. Puketza, R.A. Olsson, B. Mukherjee, *Simulating Concurrent Intrusions for Testing Intrusion Detection Systems: Parallelizing Intrusions.* Proc. of the 1995 National Information Systems Security Conference. Baltimore, Maryland, October 10-13, 1995, pp. 173-183. [This paper shows a method for parallizing attack scripts so as to test intrusion detection systems against variations on known attacks. Results indicate that some intrusion detection systems produce substantially different results when attack patterns are varried.]

[4] F. Cohen and S. Mishra *Experiments on the Impact of Computer Viruses on Modern Computer Networks*, IFIP-TC11 Computers and Security, 1994 [This paper describes testing of protection settings in LAN environments to determine efficacy of network protection against computer viruses. It includes examples of complete tests of Unix protection settings in a local area network.]

[5] M. Harrison, W. Ruzzo, and J. Ullman, *Protection in Operating Systems.* CACM 19 no. 8 (August 1976):461–471. [This paper introduces the first formal mathematical model of protection in computer systems and forms the basis for the subject/object model of computer security. It also proves that determining the protection effects of a given configuration is, in general, undecidable.]

[6] M. Lyu, editor *Handbook of Software Reliability Engineering*, [This book is a collection of excellent articles that detail the state of the art circa 1995 in reliable software design. It includes much of the current mathematical knowledge, many examples software testing, and case studies. It is an excellent reference work in this field.]

[7] P. Moyer and E. Schultz, *A Systematic Methodology for Firewall Penetration Testing*, SRI Consulting. [This internal document describes a tiger-team approach for testing of firewalls based on *attacks real network intruders use.* The basic approach is to perform functional testing by trying techniques known to be used by intruders and that exemplify the protections provided by the firewall.]

[8] C. Pfleeger, S. Pfleeger, and M. Theofanos *A Methodology for Penetration Testing*, [This paper describes a method for White-box testing of a system based on flaw hypotheses and iteration.]

[9] N. Puketza, K. Zhaung, M. Chung, B. Mukherjee, and R. Olsson, *A Methodology for Testing Intrusion Detection Systems*, Accepted by IEEE and an extended version of work presented at 17th NCSC Conference, 1994. [This paper presents a methodology for testing intrusion detection systems against certain classes of known attacks under no-load and loaded conditions and includes quantitative results showing substantial practical limitations of intrusion detection systems.]

.1 Program Listings

.1.1 The Test Pattern

This test program was modified for different variations on settings for the *f*, *g*, *u*, *i*, *j*, and *k* variables. In the example below, we selected a subset of owners (*f*), groups (*g*), and users (*u*), and tested only for access control settings with the *read* bit set in each combination of possible positions (–r, -r-, -rr, r–, r-r, rr-, rrr). This models fault assumption 2. For fault assumption 1, we use the full range of (000...777).

```
for f in 0 1 2 -1 -2 99 100 101 1024 2048 4096 65534
        do
        for g in  0 1 2 -1 -2 99 100 101 2048 4096 65534
        do
        echo "Testing files owned by $f in group $g"
        for u in 0 1 2 3 99 100 101 256 512 1024 2048 4096 65534
                do
                echo "  Access by user $u"
                for i in 0 4 # 0 1 2 3 4 5 6 7
                        do
                        for j in 0 4 # 0 1 2 3 4 5 6 7
                                do
                                for k in 0 4 # 0 1 2 3 4 5 6 7
                                        do
                                        try $f $g $u $i$j$k
done    done    done    done    done    done
```

.1.2 The try Program

This program tries access for the particular configuration requested by the testing program above. The technique involved creating a file which has a name identical to its contents and its protection setting.

```
f=$1;g=$2;u=$3;n=$4
# echo "Trying f=$f g=$g u=$u n=$n"
echo "$n" > $n
chmod $n $n;chown $f $n;chgrp $g $n
a=‘./access $u $n‘
chown $u ./cat;chgrp $u ./cat
chmod 6755 ./cat
b=‘./cat $n 2>/dev/null‘
if test Z"$a" != Z"$b"
        then
        echo "theory: $a != practice: $b (o=$f g=$g u=$u n=$n)"
# else
#       echo "$a = $b"
fi
rm -f $n
```

.1.3 The try2 Program

This program tries different filenames, stressing non-alphanumeric characters that may appear in filenames but are sometimes mishandled by programming languages because of syntactic components of their environment.

```
o=$1;g=$2;u=$3;n=$4
# echo "Trying o=$o g=$g u=$u n=$n"
for fa in a b 0 1 A B  \; \' \" \- \[ \] \{ \} \( \) \* \& \^ \% \$ \# \@ \! \~ \`
        do
        for fb in a b 0 1 A B  \; \' \" \- \[ \] \{ \} \( \) \* \& \^ \% \$ \# \@ \! \~ \`
                do
                for fc in a b 0 1 A B  \; \' \" \- \[ \] \{ \} \( \) \* \& \^ \% \$ \# \@ \! \~ \`
                        do
                        for fd in a b 0 1 A B  \; \' \" \- \[ \] \{ \} \( \) \* \& \^ \% \$ \# \@ \! \~ \`
                                do
                                for fe in a b 0 1 A B  \; \' \" \- \[ \] \{ \} \( \) \* \& \^ \% \$ \# \@ \! \~ \`
                                        do
f="$fa$fb$fc$fd$fe"
echo "$f" > $f
echo "$f"
chmod $n "$f";chown $o "$f";chgrp $g "$f"
a=`./access $u "$f"`
chown $u ./cat
chgrp $u ./cat
chmod 6755 ./cat
b=`./cat "$f" 2>/dev/null`
if test Z"$a" != Z"$b"
        then
        echo "theory: $a != practice: $b (o=$o g=$g u=$u n=$n f=$f)"
# else
#       echo "$a = $b"
fi
rm -f $f
        done    done    done    done    done
```