Use of A Taxonomy of Security Faults*

Taimur Aslam, Ivan Krsul, and Eugene H. Spafford
COAST Laboratory
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{aslam,krsul,spaf}@cs.purdue.edu

Technical Report TR-96-051

September 4, 1996

Abstract

Security in computer systems is important so as to ensure reliable operation and to protect the integrity of stored information. Faults in the implementation of critical components can be exploited to breach security and penetrate a system. These faults must be identified, detected, and corrected to ensure reliability and safeguard against denial of service, unauthorized modification of data, or disclosure of information.

We define a classification of security faults in the Unix operating system. We state the criteria used to categorize the faults and present examples of the different fault types.

We present the design and implementation details of a prototype database to store vulnerability information collected from different sources. The data is organized according to our fault categories. The information in the database can be applied in static audit analysis of systems, intrusion detection, and fault detection. We also identify and describe software testing methods that should be effective in detecting different faults in our classification scheme.

*This paper to be presented at the 19th National Information Systems Security Conference, October 22-25, 1996, Baltimore, Maryland.

1 Introduction

Security of computer systems is important so as to maintain reliable operation and to protect the integrity and privacy of stored information.

In recent years we have seen the development of sophisticated vulnerability databases and vulnerability exploitation tools by the so-called "computer underground". Some of these tools are capable of automating the exploitation of vulnerabilities that were thought to require considerable expertise, including IP and DNS spoofing. These tools are freely and widely available, and pose a significant threat that cannot be ignored. The celebrated Kevin Mitnick is an example of a vandal who used such tools and databases to penetrate hundreds of computers before being caught [17]. Although Mitnick was an expert at exploiting VMS security holes, it is widely believed that his knowledge of Unix was limited and that he was provided, by a source unknown, with ready-made tools of considerable complexity [30].

With the widespread use of computers, and increased computer knowledge in the hands of people whose objective is to obtain access to unauthorized systems and resources, it is no longer possible or desirable to implement security through obscurity [16].

To ensure that computer systems are secure against malicious attacks we need to analyze and understand the characteristics of faults that can subvert security mechanisms. A classification scheme can aid in the understanding of faults that cause security breaches by categorizing faults and grouping faults that share common characteristics.

2 Related Work

Existing fault classification schemes are not suitable for data organization because they do not clearly specify the selection criteria used. This can lead to ambiguities and result in a fault being classified in more than one category.

The Protection Analysis (PA) Project conducted research on protection errors in operating systems during the mid-1970s. The group published a series of papers, each of which described a specific type of protection error and presented techniques for finding those errors. The proposed detection techniques were based on pattern-directed evaluation methods, and used formalized patterns to search for corresponding errors [13]. The results of the study were intended for use by personnel working in the evaluation or enhancement of the security of operating systems [10].

The objective of this study was to enable anyone with little or no knowledge about computer security to discover security errors in the system by using the pattern-directed approach. However, these method could not be automated easily and their database of faults was never published. The final report of the PA project proposed four representative categories of faults. These were designed to group faults based on their syntactic structure and are too broad to be used for effective data organization.

The RISOS project was a study of computer security and privacy conducted in the mid-1970s [6]. The project was aimed at understanding security problems in existing operating systems and to suggest ways to enhance their security. The systems whose security features were studied included IBM's OS/MVT, UNIVAC's 1100 Series operating system, and Bolt Beranek and Newman's TENEX system for the PDP-10. The main contribution of the study was a classification of integrity flaws found in the operating systems studied.

The fault categories proposed in the RISOS project are general enough to classify faults from several operating systems, but the generality of the fault categories prevents fine-grain classification and can lead to ambiguities, classifying the same fault in more than one category.

Carl Landwehr et al. [24] published a collection of security flaws in different operating systems and classified each flaw according to its genesis, or the time it was introduced into the system, or the section of code where each flaw was introduced. The taxonomy proposed, unfortunately, is difficult to use for unambiguous classification because the categories are too generic and because it does not specify a clear classification criteria.

Brian Marick [25] published a survey of software fault studies from the software engineering literature. Most of the studies reported faults that were discovered in production quality software. Although the results of the study are insightful, the classification scheme provided is not suitable for data organization and unambiguous classification.

Although classical software testing techniques are not strictly concerned with a taxonomy of software flaws, we must pay close attention to them because fault classification schemes must classify faults detected using these methods.

Boundary Condition Errors:
Boundary Value Analysis (BVA)can be used to design test cases for functional testing of modules. BVA ensures that the test cases exercise the boundary conditions that can expose boundary condition errors [26]. In addition to functional testing, mutation testing can also be used to detect boundary conditions by designing appropriate language dependent mutants [7, 12, 31, 14].

Domain analysis can be applied to detect boundary condition errors. Domain analysis has been studied with two variables and examined with three variables [19, 5]. The main disadvantage of domain testing is that it can only be applied to a small number of variables as the difficulty of selecting test cases becomes increasingly complex. In an experiment by Howden, path analysis revealed the existence of one out of three path selection errors [18].

Input validation Errors: These errors result when a functional module fails to properly validate the input it accepts from another module or another process. Failure to validate the input may cause the module accepting input to fail or it may indirectly cause another interacting module to fail.

Syntax testing can be used to verify that functional modules that accept input from other processes or modules do not fail when presented with ill-formatted input.

Path analysis and testing can be applied to detect scenarios where a certain execution path may be chosen based on the input. In an experiment conducted by Howden, path testing revealed the existence of nine out of twelve computation errors.

Access Validation Errors: Path analysis can be used to detect errors that result from incorrectly specified condition constructs. Branch and Relational Operator testing (BRO) is a test case design techniques that can aid in the design of test cases that can expose access validation errors.

Failure to Handle Exceptional Condition Errors: A security breach can be caused if a system fails to handle an exceptional condition. This can include unanticipated return codes, and failure events.

Static analysis techniques such as inspection of design documents, code walk-throughs, and formal verification of critical sections can be used to ensure that a system can gracefully handle any unanticipated event. Path analysis testing can also be performed on small critical sections of code to ensure that all possible execution paths are examined. This can reveal problems that may not have been anticipated by the designers or overlooked because of complexity.

Environment Errors: These errors are dependent on the operational environment, which makes them difficult to detect [31]. It is possible that these vulnerabilities manifest themselves only when the software is run on a particular machine, under a particular operating system, or a particular configuration.

Spafford [31] used mutation testing to uncover problems with integer overflow and underflow. Mutation testing can be used to design test cases that exercise a specific set of inputs unique to the run-time environment. Path analysis and testing can also be applied to sections of the code to ensure that all possible inputs are examined.

Synchronization Errors: These are introduced because of the existence of a timing window between two operations or faults that result from improper or inadequate serialization of operations. One possible sequence of actions that may lead to a synchronization fault can be characterized as [22]:

1. A process acquires access to an object to perform some operation.
2. The process's notion of the object changes indirectly.
3. The process performs the operation on the object.

Mutation testing can be used to detect synchronization faults in a program. To detect faults that are introduced by a timing window between two operations, a `trap_on_execution` mutant can be placed between these two operations. The mutant terminates execution of the program if certain specified conditions are not satisfied. For instance, a timing window between the access permission checks and the actual logging in `xterm` could be exploited to compromise security [3]. A mutant for this vulnerability could be designed that terminated execution thus killing the mutant, if the access checks had been completed. This mutant could be placed between the access checks and the logging to detect the race condition.

Mutants can also be designed to detect improper serialization operations. Consider a set of n statement that must be executed sequentially to ensure correct operation. We assume that the statements do not contain any instructions that break the sequential lock-step execution. We can design $(n! - 1)$ mutants that rearrange the order of the n execution statements. These mutants are killed when the mutated program produces a different result than the original program.

Configuration Errors: These may result when software is adapted to new environments or from a failure to adhere to the security policy. Configuration errors consist of faults introduced after software has been developed and are faults introduced during the maintenance phase of the software life-cycle.

A static audit analysis of a system can reveal a majority of configuration errors. Among the various software testing techniques discussed, static analysis is the most effective in detecting configuration errors. The static audit of a system can be automated by using static audit tools such as COPS [15] and Tiger [29] that search a system for known avenues of penetration.

3 Fault Classification Scheme

From the work presented in the previous section, and from our experience working with security faults, we developed a taxonomy of security faults that is more appropriate for data organization. We broadly classify faults as either coding faults or emergent faults. Although personnel, communication, physical, and operations security also play an essential role in the reliable operation of computer systems, we focus on faults that are embodied in the software.

Coding faults are comprised of faults that were introduced during software development. These faults could have been introduced because of errors in programming logic, missing or incorrect requirements, or design errors [28, 32, 27, 9, 20].

Emergent faults result from improper installation of software, unexpected integration incompatibilities, and when when a programmer fails to completely understand the limitations of the run-time modules. Emergent faults are essentially those where the software performs exactly according to specification, but still causes a fault. Most policy errors can be classified as emergent faults, as can be modular sofware where each module works perfectly but the integrated product does not.

For classification purposes, we abstract each implementation error to a level that will maintain the specific characteristics yet hide the implementation details. This approach is beneficial when classifying faults from more than one programming language.

Our taxonomy of faults is comprised of the following categories:

Coding Faults

- Synchronization errors.
- Condition validation errors.

Emergent Faults

- Configuration errors.
- Environment faults.

3.1 Synchronization Errors

In our taxonomy a fault classifies as a synchronization error if:

- A fault can be exploited because of a timing window between two operations.
- A fault results from improper serialization of operations.

For example, a vulnerability was found in many versions of the `xterm` program which, if exploited, allowed users to create and delete arbitrary files in the system. If `xterm` operated as a setuid or setgid process, then a race condition between the access check permissions to the logging file and the logging itself allowed users to replace any arbitrary file with the logging file [3]. The following code illustrates how the vulnerability would be exploited.

```
# create a FIFO file and name it foo
mknod foo p
# start logging to foo
xterm -lf foo
# rename file foo to junk
mv foo junk
# create a symbolic link to password file
ln -s /etc/passwd foo
# open other end of FIFO
cat junk
```

This error occurs because of a timing window that exists between the time access permissions of the logging file are checked and the time actual logging is started. This timing window could be exploited by creating a symbolic link from the logging file to a target file in the system. As `xterm` runs setuid root, this could be used to create new files or destroy existing files in the system.

3.2 Condition Validation Errors

Conditions are usually specified as a conditional construct in the implementation language. An expression corresponding to the condition is evaluated and an execution path is chosen based on the outcome of the condition. In this discussion, we assume that an operation is allowed to proceed only if the condition evaluated to true. A condition validation error occurs if:

- A condition is missing. This allows an operation to proceed regardless of the outcome of the condition expression.
- A condition is incorrectly specified. Execution of the program would proceed along an alternate path, allowing an operation to proceed regardless of the outcome of the condition expression, completely invalidating the check.
- A predicate in the condition expression is missing. This would evaluate the condition incorrectly and allow the alternate execution path to be chosen.

Condition errors are coding faults that occur because a programmer misunderstood the requirements or made a logic error when the condition was specified.

In our taxonomy, a fault classifies as a condition error if one of the following conditions is missing or not specified correctly:

Check for limits. Before an operation can proceed, the system must ensure that it can allocate the required resources without causing starvation or deadlocks. For input/output operations, the system must also ensure that a user/process does not read or write beyond its address boundaries.

Check for access rights. The system must ensure that a user/process can only access an object in its access domain. The mechanics of this check would differ among different systems depending on how access control mechanisms are implemented.

Check for valid input. Any routines that accept input directly from a user or from another routine must check for the validity of input. This includes checks for:

- Field-value correlation.
- Syntax.
- Type and number of parameters or input fields.
- Missing input fields or delimiters.
- Extraneous input fields or parameters.

Failure to properly validate input may indirectly cause other functional modules to fail and cause the system to behave in an unexpected manner.

Check for the origin of a subject. In this context, subject refers to a user/process, host, and shared data objects. The system must authenticate the subject's identity to prevent against identity compromise attacks.

In Unix, `/etc/exports` specifies a lists of trusted remote hosts that are allowed to mount the file system. In SunOS 4.1.x, if a host entry in the file was longer than 256 characters, or if the number of hosts exceeded the cache capacity, a buffer overflow allowed any non-trusted host to mount the file system [4]. This allowed unauthorized users read and write access to all files on a system. This error occurred because the system failed to check that it had read more than 256 characters or that it had exhausted the cache capacity.

Another example is the `uux` utility in Unix. This utility allows users to remotely execute a limited set of commands. A flaw in the parsing of the command line allowed remote users to execute arbitrary commands on the system [11]. The command line to be executed was received by the remote system, and parsed to see if the commands in the line were among the set of commands that could be executed. `uux` read the first word of the line, and skipped characters until a delimiter character (;,^, |) was read. `uux` would continue this way until the end of the line was read. However, two delimiters (&, ‘) were missing from the set, so a command following these characters would never be checked before being executed. For example, a user could execute any command by executing the following sequence.

```
uux remote_machine ! rmail anything & command
```

In `uux` the command after the “`&`” character would not be checked before being executed. This allowed users to execute unauthorized commands on a remote system. This error occurred because `uux` failed to check for the missing delimiters.

3.3 Configuration Errors

The configuration of a system consists of the software and hardware resources. In our taxonomy, a fault can be classified as a configuration error if:

- A program/utility is installed in the wrong place.
- A program/utility is installed with incorrect setup parameters.
- A secondary storage object or program is installed with incorrect permissions.

For example, at some sites the `tftp` daemon was enabled in such a way that it allowed any user on the Internet to access any file on the machine running `tftp`. This flaw qualifies as a configuration error in our taxonomy because `tftp` was not properly installed. `tftp` should have been enabled such that access to the file system was restricted via the `chroot` command [1, 2].

3.4 Environment Faults

Environment faults are introduced when specifications are translated to code but sufficient attention is not paid to the run-time environment. Environmental faults can also occur when different modules interact in an unanticipated manner. Independently the modules may function according to specifications but an error occurs when they are subjected to a specific set of inputs in a particular configuration environment.

For example, the `exec` system call overlays a new process image over an old one. The new image is constructed from an executable object file or a data file containing commands for an interpreter. When an interpreter file is executed, the arguments specified in the `exec` call are passed to the interpreter. Most interpreters take "`-i`" as an argument to start an interactive shell.

In SunOS version 3.2 and earlier, any user could create an interactive shell by creating a link with the name "`-i`" to a setuid shell script. `exec` passed "`-i`" as an argument to the shell interpreter that started an interactive shell. Both the `exec` system call and the shell interpreter worked according to specifications. The error resulted from an interaction between the shell interpreter and the `exec` call that had not been considered.

4 Selection Criteria

For each of the classifications described in our taxonomy, it should be possible to design a decision process that would help us classify faults automatically and unambigously. Many such decision processes are possible and we present a selection criteria that can be used to classify security faults into different categories to distinctly classify each fault.

For each fault category we present a series of questions that are used to determine membership in a specific category. An affirmative answer to a question in that series qualifies the fault to be classified in the corresponding category.

4.1 Condition Validation Errors

The following sets of questions can be used to determine if a fault can be classified as a condition validation error.

Boundary Condition Errors

- Did the error occur when a process attempted to read or write beyond a valid address boundary?
- Did the error occur when a system resource was exhausted?
- Did the error result from an overflow of a static-sized data structure?

Access Validation Errors

- Did the error occur when a subject invoked an operation on an object outside its access domain?
- Did the error occur as a result of reading or writing to/from a file or device outside a subject's access domain?

Origin Validation Errors

- Did the error result when an object accepted input from an unauthorized subject?
- Did the error result because the system failed to properly or completely authenticate a subject?

Input Validation Errors

- Did the error occur because a program failed to recognize syntactically incorrect input?
- Did the error result when a module accepted extraneous input fields?
- Did the error result when a module did not handle missing input fields?
- Did the error result because of a field-value correlation error?

Failure to Handle Exceptional Conditions

- Did the error manifest itself because the system failed to handle an exceptional condition, generated by a functional module, device, or user input?

4.2 Synchronization Errors

This section presents the criteria that can be used to decide if a fault can be classified as a synchronization error.

Race Condition Errors

- Is the error exploited during a timing window between two operations?

Serialization Errors

- Did the error result from inadequate or improper serialization of operations?

Atomicity Errors

- Did the error occur when partially-modified data structures were observed by another process?
- Did the error occur because the code terminated with data only partially modified as part of some operation that should have been atomic?

4.3 Environment Errors

This section presents a series of questions that be used to decide if a fault can be classified as an environment error.

- Does the error result from an interaction in a specific environment between functionally correct modules?
- Does the error occur only when a program is executed on a specific machine, under a particular configuration?
- Does the error occur because the operational environment is different from what the software was designed for?

4.4 Configuration Errors

The following questions can be used to determine if a fault can be classified as a configuration error.

- Did the error result because a system utility was installed with incorrect setup parameters?
- Did the error occur by exploiting a system utility that was installed in the wrong place?
- Did the error occur because access permissions were incorrectly set on a utility such that it violated the security policy?

5 Applications of Fault Taxonomy

In this section, we present some applications of our fault classification scheme. In addition, we also identified some testing techniques that may be used to systematically detect those faults.

5.1 Vulnerability Database

Landwehr et al.[24] observe that the history of software failure has been mostly undocumented and knowing how systems have failed can help us design better systems that are less prone to failure. The design of a vulnerability database is one step in that direction.

The database could serve as a repository of vulnerability information collected from different sources, could be organized to allow useful queries to be performed on the data, and could provide useful information to system designers in identifying areas of weaknesses in the design, requirements, or implementation of software. The database could also be used to maintain vendor patch information, vendor and response team advisories, and catalog the patches applied in response to those advisories. This information would be helpful to system administrators maintaining legacy systems.

Taimur Aslam designed and built a prototype vulnerability database [8] to explore the usefulness of the classification scheme presented in this paper. Our vulnerability database is based on a relational schema model that consists of both physical and conceptual entities. These entities are represented as relations (tables) in the model. Relational algebra defines the operations that can be performed on the the relations. It also defines a set of basis functions such that any query in the relational model can be specified only in terms of these functions. The basis functions in the relational model are: SELECT, PROJECT, UNION, DIFFERENCE, and CARTESIAN PRODUCT.

The database was populated with vulnerability information from several sources and proved a useful resource in the development of intrusion detection patterns for the COAST intrusion detection system IDIOT [22, 23, 21].

6 Future Work

It needs to be determined whether our classification scheme needs to be enhanced to encompass other operating systems. Many modern systems are based on a software architecture that is different from that of Unix. These include micro-kernels, object-oriented, and distributed operating systems. If needed, our classification scheme can be easily expanded because the criteria used for the taxonomy does not rely on implementation details and is designed to encompass general characteristics of a fault. Also, our existing categories can be extended to include any news faults that cannot be classified into the existing categories, should any be found.

The COAST vulnerability database also needs to be extended with more vulnerabilities. The database currently has over 80 significant faults, largely from variants of the UNIX operating system. We have data to extend the collection to almost 150 cataloged faults. Once this is complete, we intend to evaluate the structure and use of the database for some of our original research goals: building static audit tools, guiding software design and testing, and enhancing incident response capabilities.

7 Conclusion

In this paper we presented a fault classification scheme that helps in the unambiguous classification of security faults that is suitable for data organization and processing. A database of vulnerabilities using this classification was implemented and is being used to aid in the production of tools that detect and prevent computer break-ins. The classification scheme has contributed to the understanding of computer security faults that cause security breaches.

References

[1] CERT advisory CA-91:18. Computer Emergency Response Team Advisory, 1991.

[2] CERT advisory CA-91:19. Computer Emergency Response Team Advisory, 1991.

[3] CERT advisory CA-93:17. Computer Emergency Response Team Advisory, 1993.

[4] CERT advisory CA-94:02. Computer Emergency Response Team Advisory, 1994.

[5] DeMillo R. A, Hocking E. D, and Meritt M. J. A Comparison of Some Reliable Test Data Generation Procedures. Technical report, Georgia Institue of Technology, 1981.

[6] R.P. Abbott et al. Security Analysis and Enhancements of Computer Operating Systems. Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[7] H. Agrawal, R. DeMillo, R. Hathaway, and et al. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.

[8] Taimur Aslam. A taxonomy of security faults in the unix operating system. Master's thesis, Purdue University, 1995.

[9] Boris Beizer. *Software Testing Techniques.* Electrical Engineering/Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.

[10] Richard Bibsey, Gerald Popek, and Jim Carlstead. Inconsistency of a single data value over time. Technical report, Information Sciences Institute,University of Southern California, December 1975.

[11] Matt Bishop. Analyzing the Security of an Existing Computer System. IEEE Fall Joint Computer Conference, November 1986.

[12] T.A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, May 1980.

[13] Jim Carlstead, Richard Bibsey II, and Gerald Popek. Pattern-directed protection evaluation. Technical report, Information Sciences Institue,University of Southern California, June 1975.

[14] Richard A. DeMillo and Aditya P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software. Technical report, Software Engineering Research Center, Purdue University , SERC-TR-92-P, March 1991.

[15] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Software Engineering Research Center, Purdue University, September 1991.

[16] Simson Garfinkel and Eugene Spafford. *Practical Unix and Internet Security.* O'Reily and Associates, second edition, 1996.

[17] Katie Hafner and John Markoff. *Cyberpunk: Outlaws and Hackers on the Computer Frontier.* Touchstone, 1992.

[18] W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–214, 1976.

[19] White L. J and Cohen E. K. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[20] D.E. Knuth. The Errors of TeX. *Software Practice and Experience*, 19(7):607–685, 1989.

[21] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Purdue University, 1995.

[22] Sandeep Kumar and Eugene Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *17th National Computer Security Conference*, 1994.

[23] Sandeep Kumar and Eugene H. Spafford. A software architecture to support misuse intrusion detection. Technical Report CSD-TR-95-009, Purdue University, 1995.

[24] Carl Landwher et al. A taxonomy of computer program security flaws. Technical report, Naval Research Laboratory, November 1993.

[25] Brian Marick. A survey of software fault surveys. Technical Report UIUCDCS-R-90-1651, University of Illinois at Urbana-Champaign, December 1990.

[26] G. Myers. *The Art of Software Testing.* Wiley, 1979.

[27] D. Potier, J.L. Albin, R. Ferrol, and A. Bilodeau. Experiments with Computer Software Complexity and Reliability. In *Proceedings of the 6th International Conference on Software Engineering*, pages 94–103. IEEE Press, 1982.

[28] Raymond J. Rubey. Quantitative Aspects of Software Validation. *SIGPLAN Notices*, SE-5(3):276–286, May 1975.

[29] David R. Safford, Douglas Lee Schales, and David K. Hess. The TAMU security package. In Edward Dehart, editor, *Proceedings of the Security IV Conference*, pages 91–118, 1993.

[30] Tsutomu Shimomura and John Markoff. *Takedown.* Hyperion Books, 1996.

[31] Eugene H. Spafford. Extending Mutation Testing to Find Environmental Bugs. *Software Practice and Principle*, 20(2):181–189, Feb 1990.

[32] David M. Weiss and Victor R. Basili. Evaluating Software development by Analysis of Changes: Some Data from the Software Engineering Laboratory. *IEEE Transactions on Software Engineering*, SE-11(2):3–11, February 1985.