



**GFI LANguard N.S.S. v8 - New version out now!**

Voted Favorite Commercial Security Tool for two consecutive years by NMAP users.

Download free 30-day trial today! - [www.gfi.com/lannetscan/](http://www.gfi.com/lannetscan/)

## NCSC-TG-023: Bright Orange book

A Guide to Understanding Security Testing and Test Documentation in Trusted Systems

- Published: **Oct 16, 2002**
- Updated: **Oct 16, 2002**
- Section: [Network Security Library :: NCSC&DoD Rainbow series](#)
- Author: [The Editor](#)
- Company: [WindowSecurity.com](#)
- Rating: **2/5 - 1 Votes**



NCSC-TG-023

VERSION-1

NATIONAL COMPUTER SECURITY CENTER

A GUIDE TO

UNDERSTANDING

SECURITY TESTING

AND

TEST DOCUMENTATION

IN

TRUSTED SYSTEMS

July 1993

Approved for Public Release:

Distribution Unlimited.

NCSC-TG-023

Library No. S-232.561

Version-1

FOREWORD

The National Computer Security Center is issuing A Guide to Understanding Security Testing

and Test Documentation in Trusted Systems as part of the "Rainbow Series" of documents our Technical Guidelines Program produces. In the Rainbow Series, we discuss in detail the features of the Department of Defense Trusted Computer System Evaluation Criteria (DoD 5200.28-STD) and provide guidance for meeting each requirement. The National Computer Security Center, through its Trusted Product Evaluation Program, evaluates the security features of commercially produced computer systems. Together, these programs ensure that users are capable of protecting their important data with trusted computer systems.

The specific guidelines in this document provide a set of good practices related to security testing and the development of test documentation. This technical guideline has been written to help the vendor and evaluator community understand what deliverables are required for test documentation, as well as the level of detail required of security testing at all classes in the Trusted Computer System Evaluation Criteria.

As the Director, National Computer Security Center, Invite your suggestions for revision to this technical guideline. We plan to review this document as the need arises.

National Computer Security Center

Attention: Chief, Standard, Criteria and Guidelines Division

9800 Savage Road

Fort George G. Meade, MD 20755-6000

Patrick R. Gallagher, Jr.                      January, 1994

Director

National Computer Security Center

#### ACKNOWLEDGMENTS

Special recognition and acknowledgment for his contributions to this document are extended to Virgil D. Gligor, University of Maryland, as primary author of this document.

Special thanks are extended to those who enthusiastically gave of their time and technical expertise in reviewing this guideline and providing valuable comments and suggestions. The assistance of C. Sekar Chandrasekaran, IBM and Charles Bonneau, Honeywell Federal Systems, in the preparation of the examples presented in this guideline is gratefully acknowledged.

Special recognition is extended to MAJ James P. Gordon, U.S. Army, and Leon Neufeld as National Computer Security Center project managers for this guideline.

#### TABLE OF CONTENTS

|                        |     |
|------------------------|-----|
| FOREWORD               | i   |
| ACKNOWLEDGMENTS        | iii |
| 1. INTRODUCTION        | 1   |
| 1.1 PURPOSE            | 1   |
| 1.2 SCOPE              | 1   |
| 1.3 CONTROL OBJECTIVES | 2   |

|                                                               |    |
|---------------------------------------------------------------|----|
| 2. SECURITY TESTING OVERVIEW                                  | 3  |
| 2.1 OBJECTIVES                                                | 3  |
| 2.2 PURPOSE                                                   | 3  |
| 2.3 PROCESS                                                   | 4  |
| 2.3.1 System Analysis                                         | 4  |
| 2.3.2 Functional Testing                                      | 4  |
| 2.3.3 Security Testing                                        | 5  |
| 2.4 SUPPORTING DOCUMENTATION                                  | 5  |
| 2.5 TEST TEAM COMPOSITION                                     | 6  |
| 2.6 TEST SITE                                                 | 17 |
| 3. SECURITY TESTING - APPROACHES, DOCUMENTATION, AND EXAMPLES | 8  |
| 3.1 TESTING PHILOSOPHY                                        | 8  |
| 3.2 TEST AUTOMATION                                           | 9  |
| 3.3 TESTING APPROACHES                                        | 11 |
| 3.3.1 Monolithic (Black-Box) Testing                          | 11 |
| 3.3.2 Functional-Synthesis (White-Box) Testing                | 13 |
| 3.3.3 Gray-Box Testing                                        | 25 |
| 3.4 RELATIONSHIP WITH THE TCSEC SECURITY TESTING REQUIREMENTS | 18 |
| 3.5 SECURITY TEST DOCUMENTATION                               | 21 |
| 3.5.1 Overview                                                | 21 |
| 3.5.2 Test Plan                                               | 22 |
| 3.5.2.1 Test Conditions                                       | 22 |
| 3.5.2.2 Test Data                                             | 24 |
| 3.5.2.3 Coverage Analysis                                     | 25 |
| 3.5.3 Test Procedures                                         | 27 |
| 3.5.4 Test Programs                                           | 27 |
| 3.5.5 Test Log                                                | 28 |
| 3.5.6 Test Report                                             | 28 |

|                                                                                          |    |
|------------------------------------------------------------------------------------------|----|
| 3.6 SECURITY TESTING OF PROCESSORS' HARDWARE/FIRMWARE PROTECTION MECHANISMS              | 28 |
| 3.6.1 The Need for Hardware/Firmware Security Testing                                    | 29 |
| 3.6.2 Explicit TCSEC Requirements for Hardware Security Testing                          | 30 |
| 3.6.3 Hardware Security Testing vs. System Integrity Testing                             | 31 |
| 3.6.4 Goals, Philosophy, and Approaches to Hardware Security Testing                     | 31 |
| 3.6.5 Test Conditions, Data, and Coverage Analysis for Hardware Security Testing         | 32 |
| 3.6.5.1 Test Conditions for Isolation and Noncircumventability Testing                   | 32 |
| 3.6.5.2 Text Conditions for Policy-Relevant Processor Instructions                       | 33 |
| 3.6.5.3 Tests Conditions for Generic Security Flaws                                      | 33 |
| 3.6.6 Relationship between Hardware/Firmware Security Testing and the TCSEC Requirements | 34 |
| 3.7 TEST PLAN EXAMPLES                                                                   | 36 |
| 3.7.1 Example of a Test Plan for "Access"                                                | 37 |
| 3.7.1.1 Test Conditions for Mandatory Access Control of "Access"                         | 38 |
| 3.7.1.2 Test Data for MAC Tests                                                          | 38 |
| 3.7.1.3 Coverage Analysis                                                                | 39 |
| 3.7.2 Example of a Test Plan for "Open"                                                  | 43 |
| 3.7.2.1 Test Conditions for "Open"                                                       | 43 |
| 3.7.2.2 Test Data for the Access Graph Dependency Condition                              | 44 |
| 3.7.2.3 Coverage Analysis                                                                | 46 |
| 3.7.3 Examples of a Test Plan for "Read"                                                 | 46 |
| 3.7.3.1 Test Conditions for "Read"                                                       | 47 |
| 3.7.3.2 Test Data for the Access-Check Dependency Condition                              | 47 |
| 3.7.3.3 Coverage Analysis                                                                | 51 |
| 3.7.4 Examples of Kernel Isolation Test Plans                                            | 51 |
| 3.7.4.1 Test Conditions                                                                  | 51 |
| 3.7.4.2 Test Data                                                                        | 51 |
| 3.7.4.3 Coverage Analysis                                                                | 53 |
| 3.7.5 Examples of Reduction of Cyclic Test Dependencies                                  | 54 |

|                                                                    |    |
|--------------------------------------------------------------------|----|
| 3.7.6 Example of Test Plans for Hardware/Firmware Security Testing | 57 |
| 3.7.6.1 Test Conditions for the Ring Crossing Mechanism            | 58 |
| 3.7.6.2 Test Data                                                  | 58 |
| 3.7.6.3 Coverage Analysis                                          | 60 |
| 3.7.7 Relationship with the TCSEC Requirements                     | 62 |
| 4. COVERT CHANNEL TESTING                                          | 66 |
| 4.1 COVERT CHANNEL TEST PLANS                                      | 66 |
| 4.2 AN EXAMPLE OF A COVERT CHANNEL TEST PLAN                       | 67 |
| 4.2.1 Test Plan for the Upgraded Directory Channel                 | 67 |
| 4.2.1.1 Test Condition                                             | 68 |
| 4.2.1.2 Test Data                                                  | 68 |
| 4.2.1.3 Coverage Analysis                                          | 70 |
| 4.2.2 Test Programs                                                | 70 |
| 4.2.3 Test Results                                                 | 70 |
| 4.3 RELATIONSHIP WITH THE TCSEC REQUIREMENTS                       | 70 |
| 5. DOCUMENTATION OF SPECIFICATION-TO-CODE CORRESPONDENCE           | 72 |
| APPENDIX                                                           | 73 |
| 1 Specification-to-Code Correspondence                             | 73 |
| 2 Informal Methods for Specification-to-Code Correspondence        | 74 |
| 3 An Example of Specification-to-Code Correspondence               | 76 |
| GLOSSARY                                                           | 83 |
| REFERENCES                                                         | 90 |

## 1. INTRODUCTION

The National Computer Security Center (NCSC) encourages the widespread availability of trusted computer systems. In support of this goal the Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) was created as a metric against which computer systems could be evaluated. The NCSC published the TCSEC on 15 August 1983 as CSC-STD-001-83. In December 1985, the Department of Defense (DoD) adopted it, with a few changes, as a DoD Standard, DoD 5200.28-STD. [13] DoD Directive 5200.28, "Security Requirements for Automatic Data Processing (ADP) Systems," requires that the TCSEC be used throughout the DoD. The NCSC uses the TCSEC as a standard for evaluating the effectiveness of security controls built into ADP systems. The TCSEC is divided into four divisions: D, C, B, and A. These divisions are ordered in a hierarchical manner with the highest division (A) being reserved for systems providing the best

available level of assurance. Within divisions C and B there are a number of subdivisions known as classes. In turn, these classes are also ordered in a hierarchical manner to represent different levels of security.

## 1.1 PURPOSE

Security testing is a requirement for TCSEC classes C1 through A1. This testing determines that security features for a system are implemented as designed and that they are adequate for the specified level of trust. The TCSEC also requires test documentation to support the security testing of the security features of a system. The TCSEC evaluation process includes security testing and evaluation of test documentation of a system by an NCSC evaluation team. A Guide to Understanding Security Testing and Test Documentation for Trusted Systems will assist the operating system developers and vendors in the development of computer security testing and testing procedures. This guideline gives system developers and vendors suggestions and recommendations on how to develop testing and testing documentation that will be found acceptable by an NCSC Evaluation Team.

## 1.2 SCOPE

TCSEC classes C1 through A1 assurance is gained through security testing and the accompanying test documentation of the ADP system. Security testing and test documentation ensures that the security features of the system are implemented as designed and are adequate for an application environment. This guideline discusses the development of security testing and test documentation for system developers and vendors to prepare them for the evaluation process by the NCSC. This guideline addresses, in detail, various test methods and their applicability to security and accountability policy testing. The Trusted Computing Base (TCB) isolation, noncircumventability testing, processor testing, and covert channel testing methods are examples.

This document provides an in-depth guide to security testing. This includes the definitions, writing and documentation of the test plans for security and a brief discussion of the mapping between the formal top-level specification (FTLS) of a TCB and the TCB implementation specifications. This document also provides a standard format for test plans and test result presentation. Extensive documentation of security testing and specification-to-code correspondence arise both during a system evaluation and, more significantly, during a system life cycle. This guideline addresses evaluation testing, not life-cycle testing. This document complements the security testing guideline that appears in Section 10 of the TCSEC.

The scope and approach of this document is to assist the vendor in security testing and in particular functional testing. The vendor is responsible for functional testing, not penetration testing. If necessary, penetration testing is conducted by an NCSC evaluation team. The team collectively identifies penetration vulnerabilities of a system and rates them relative to ease of attack and difficulty of developing a hierarchy penetration scenario. Penetration testing is then conducted according to this hierarchy, with the most critical and easily executed attacks attempted first [17].

This guideline emphasizes the testing of systems to meet the requirements of the TCSEC. A Guide to Understanding Security Testing and Test Documentation for Trusted Systems does not address the testing of networks, subsystems, or new versions of evaluated computer system products. It only addresses the requirements of the TCSEC.

Information in this guideline derived from the requirements of the TCSEC is prefaced by the word "shall." Recommendations that are derived from commonly accepted good practices are

prefaced by the word "should." The guidance contained herein is intended to be used when conducting and documenting security functional testing of an operating system. The recommendations in this document are not to be construed as supplementary requirements to the TCSEC. The TCSEC is the only metric against which systems are to be evaluated.

Throughout this guideline there are examples, illustrations, or citations of test plan formats that have been used in commercial product development. The use of these examples, illustrations, and citations is not meant to imply that they contain the only acceptable test plan formats. The selection of these examples is based solely on their availability in computer security literature. Examples in this document are not to be construed as the only implementations that will satisfy the TCSEC requirements. The examples are suggestions of appropriate implementations.

### 1.3 CONTROL OBJECTIVES

The TCSEC and DoD 5200.28-M [14] provide the control objectives for security testing and documentation. Specifically these documents state the following:

"Component's Designated Approving Authorities, or their designees for this purpose . . . will assure: . . .

"4. Maintenance of documentation on operating systems (O/S) and all modifications thereto, and its retention for a sufficient period of time to enable tracing of security-related defects to their point of origin or inclusion in the system.

"5. Supervision, monitoring, and testing, as appropriate, of changes in an approved ADP System that could affect the security features of the system, so that a secure system is maintained.

"6. Proper disposition and correction of security deficiencies in all approved ADP Systems, and the effective use and disposition of system housekeeping or audit records, records of security violations or security-related system malfunctions, and records of tests of the security features of an ADP System.

"7. Conduct of competent system Security Testing and Evaluation (ST&E), timely review of system ST&E reports, and correction of deficiencies needed to support conditional or final approval or disapproval of an ADP system for the processing of classified information.

"8. Establishment, where appropriate, of a central ST&E coordination point for the maintenance of records of selected techniques, procedures, standards, and tests used in testing and evaluation of security features of ADP systems which may be suitable for validation and use by other Department of Defense components."

Section 5 of the TCSEC gives the following as the Assurance Control Objective:

"The third basic control objective is concerned with guaranteeing or providing confidence that the security policy has been implemented correctly and that the protection critical elements of the system do, indeed, accurately mediate and enforce the intent of that policy. By extension, assurance must include a guarantee that the trusted portion of the system works only as intended. To accomplish these objectives, two types of assurance are needed. They are life-cycle assurance and operational assurance.

"Life-cycle assurance refers to steps taken by an organization to ensure that the system is designed, developed, and maintained using formalized and rigorous controls and standards. Computer systems that process and store sensitive or classified information depend on the hardware and software to protect that information. It follows that the

hardware and software themselves must be protected against unauthorized changes that could cause protection mechanisms to malfunction or be bypassed completely. For this reason, trusted computer systems must be carefully evaluated and tested during the design and development phases and reevaluated whenever changes are made that could affect the integrity of the protection mechanisms. Only in this way can confidence be provided that the hardware and software interpretation of the security policy is maintained accurately and without distortion." [13]

## 2. SECURITY TESTING OVERVIEW

This section provides the objectives, purpose, and a brief overview of vendor and NCSC security testing. Test team composition, test site location, testing process, and system documentation are also discussed.

### 2.1 OBJECTIVES

The objectives of security testing are to uncover all design and implementation flaws that enable a user external to the TCB to violate security and accountability policy, isolation, and noncircumventability.

### 2.2 PURPOSE

Security testing involves determining (1) a system security mechanism adequacy for completeness and correctness and (2) the degree of consistency between system documentation and actual implementation. This is accomplished through a variety of assurance methods such as analysis of system design documentation, inspection of test documentation, and independent execution of functional testing and penetration testing.

### 2.3 PROCESS

A qualified NCSC team of experts is responsible for independently evaluating commercial products to determine if they satisfy TCSEC requirements. The NCSC is also responsible for maintaining a listing of evaluated products on the NCSC Evaluated Products List (EPL). To accomplish this mission, the NCSC Trusted Product Evaluation Program has been established to assist vendors in developing, testing, and evaluating trusted products for the EPL. Security testing is an integral part of the evaluation process as described in the Trusted Product Evaluations-A Guide For Vendors. [18]

#### 2.3.1 System Analysis

System analysis is used by the NCSC evaluation team to obtain a complete and in-depth understanding of the security mechanisms and operations of a vendor's product prior to conducting security testing. A vendor makes available to an NCSC team any information and training to support the NCSC team members in their understanding of the system to be tested. The NCSC team will become intimately familiar with a vendor's system under evaluation and will analyze the product design and implementation, relative to the TCSEC.

System candidates for TCSEC ratings B2 through A1 are subject to verification and covert channel analyses. Evaluation of these systems begins with the selection of a test configuration, evaluation of vendor security testing documentation, and preparation of an NCSC functional test plan.

#### 2.3.2 Functional Testing

Initial functional testing is conducted by the vendor and results are presented to the NCSC team.

The vendor should conduct extensive functional testing of its product during development, field testing, or both. Vendor testing should be conducted by procedures defined in a test plan.

Significant

events during testing should be placed in a test log. As testing proceeds sequentially through each

test case, the vendor team should identify flaws and deficiencies that will need to be corrected.

When a hardware or software change is made, the test procedure that uncovered the problem should then be repeated to validate that the problem has been corrected. Care should be taken to verify that

the change does not affect any previously tested procedure. These procedures also should be repeated

when there is concern that flaws or deficiencies exist. When the vendor team has corrected all functional problems and the team has analyzed and retested all corrections, a test report should be

written and made a part of the report for review by the NCSC test team prior to NCSC security testing.

The NCSC team is responsible for testing vendor test plans and reviewing vendor test documentation. The NCSC team will review the vendor's functional test plan to ensure it sufficiently

covers each identified security mechanism and explanation in sufficient depth to provide reasonable

assurance that the security features are implemented as designed and are adequate for an application

environment. The NCSC team conducts its own functional testing and, if appropriate, penetration testing after a vendor's functional testing has been completed.

A vendor's product must be free of design and implementation changes, and the documentation to support security testing must be completed before NCSC team functional testing. Functional security testing is conducted on C1 through A1 class systems and penetration testing on B2, B3, and A1 class systems. The NCSC team may choose to repeat any of the functional tests performed by the vendor and/or execute its own functional test. During testing by the NCSC team, the team informs the vendor of any test problems and provides the vendor with an opportunity to correct implementation flaws. If the system satisfies the functional test requirements, B2 and above candidates undergo penetration testing. During penetration testing the NCSC team collectively identifies penetration vulnerabilities in the system and rates them relative to ease of attack and

difficulty in developing a penetration hierarchy. Penetration testing is then conducted according to

this hierarchy with the most critical and most easily executed attacks attempted first [17].

The vendor

is given limited opportunity to correct any problems identified [17]. When opportunity to correct

implementation flaws has been provided and corrections have been retested, the NCSC team documents the test results. The test results are input which support a final rating, the publication of

the Final Report and the EPL entry.

### 2.3.3 Security Testing

Security testing is primarily the responsibility of the NCSC evaluation team. It is important to note, however, that vendors shall perform security testing on a product to be evaluated using NCSC

test methods and procedures. The reason for vendor security testing is two-fold: First, any TCB changes required as a result of design analysis or formal evaluation by the NCSC team will

require that the vendor (and subsequently the evaluation team) retest the TCB to ensure that its security properties are unaffected and the required changes fixed the test problems. Second, any new system release that affects the TCB must undergo either a reevaluation by the NCSC or a rating-maintenance evaluation by the vendor itself. If a rating maintenance is required, which is expected to be the case for the preponderant number of TCB changes, the security testing responsibility, including all the documentation evidence, becomes a vendor's responsibility-not just that of the NCSC evaluation team.

Furthermore, it is important to note that the system configuration provided to the evaluation team for security testing should be the same as that used by the vendor itself. This ensures that consistent test results are obtained. It also allows the evaluation team to examine the vendor test suite and to focus on areas deemed to be insufficiently tested. Identifying these areas will help speed the security testing of a product significantly. (An important implication of reusing the vendor's test suite is that security testing should yield repeatable results.)

When the evaluation team completes the security testing, the test results are shown to the vendor. If any TCB changes are required, the vendor shall correct or remove those flaws before TCB retesting by the NCSC team is performed.

## 2.4 SUPPORTING DOCUMENTATION

Vendor system documentation requirements will vary, and depending on the TCSEC class a candidate system will be evaluated for, it can consist of the following:

Security Features User's Guide. It describes the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another. This may be used to identify the protection mechanisms that need to be covered by test procedures and test cases.

Trusted Facility Manual. It describes the operation and administration of security features of the system and presents cautions about functions and privileges that should be controlled when running a secure facility. This may identify additional functions that need to be tested.

Design Documentation. It describes the philosophy of protection, TCB interfaces, security policy model, system architecture, TCB protection mechanisms, top level specifications, verification plan, hardware and software architecture, system configuration and administration, system programming guidelines, system library routines, programming languages, and other topics.

Covert Channel Analysis Documentation. It describes the determination and maximum bandwidth of each identified channel.

System Integrity Documentation. It describes the hardware and software features used to validate periodically the correct operation of the on-site hardware and firmware elements of the TCB.

Trusted Recovery Documentation. It describes procedures and mechanisms assuring that after an ADP system failure or other discontinuity, recovery is obtained without a protection compromise. Information describing procedures and mechanisms may also be found in the Trusted Facility Manual.

Test Documentation. It describes the test plan, test logs, test reports, test procedures, and test results and shows how the security mechanisms were functionally tested, covert channel bandwidth, and mapping between the FTLS and the TCB source code. Test documentation is used to document plans, tests, and results in support of validating and verifying the security testing effort.

## 2.5 TEST TEAM COMPOSITION

A vendor test team should be formed to conduct security testing. It is desirable for a vendor to provide as many members from its security testing team as possible to support the NCSC during its security testing. The reason for this is to maintain continuity and to minimize the need for retraining throughout the evaluation process. The size, education, and skills of the test team will vary depending on the size of the system and the class for which it is being evaluated. (See Chapter 10 of the TCSEC, "A Guideline on Security Testing.")

A vendor security testing team should be comprised of a team leader and two or more additional members depending on the evaluated class. In selecting personnel for the test team, it is important to assign individuals who have the ability to understand the hardware and software architecture of the system, as well as an appropriate level of experience in system testing. Engineers and scientists with backgrounds in electrical engineering, computer science and software engineering are ideal candidates for functional security testing. Prior experience with penetration techniques is important for penetration testing. A mathematics or logic background can be valuable in formal specifications involved in AI system evaluation.

The NCSC test team is formed using the guidance of Chapter 10, in the TCSEC, "A Guideline on Security Testing." This chapter specifies test team composition, qualifications and parameters.

Vendors may find these requirements useful recommendations for their teams.

## 2.6 TEST SITE

The location of a test site is a vendor responsibility. The vendor is to provide the test site. The evaluator's functional test site may be located at the same site at which the vendor conducted his functional testing. Proper hardware and software must be available for testing the configuration as well as appropriate documentation, personnel, and other resources which have a significant impact on the location of the test site.

## 3. SECURITY TESTING-APPROACHES, DOCUMENTATION, AND EXAMPLES

### 3.1 TESTING PHILOSOPHY

Operating systems that support multiple users require security mechanisms and policies that guard against unauthorized disclosure and modification of critical user data. The TCB is the principal operating system component that implements security mechanisms and policies that must itself be protected [13]. TCB protection is provided by a reference monitor mechanism whose data structures and code are isolated, noncircumventable, and small enough to be verifiable. The reference monitor ensures that the entire TCB is isolated and noncircumventable.

Although TCBs for different operating systems may contain different data structures and programs, they all share the isolation, noncircumventability, and verifiability properties that distinguish them from the rest of the operating system components. These properties imply that the security functional testing of an operating system TCB may require different methods from those commonly used in software testing for all security classes of the TCSEC.

Security testing should be done for TCBs that are configured and installed in a specific system and operate in a normal mode (as opposed to maintenance or test mode). Tests should be done using user-level programs that cannot read or write internal TCB data structures or programs. New data structures and programs should also not be added to a TCB for security testing purposes, and special TCB entry points that are unavailable to user programs should not be used. If a TCB is tested in the maintenance mode using programs that cannot be run at the user level, the security tests would be meaningless because assurance cannot be gained that the TCB performs user-level access control correctly. If user-level test programs could read, write or add internal TCB data structures and programs, as would be required by traditional instrumentation testing techniques, the TCB would lose its isolation properties. If user-level test programs could use special TCB entry points not normally available to users, the TCB would become circumventable in the normal mode of operation.

Security testing of operating system TCBs in the normal mode of operation using user-level test programs (which do not rely on breaching isolation and noncircumventability) should address the following problems of TCB verifiability through security testing: (1) Coverage Analysis, (2) Reduction of Cyclic Test Dependencies, (3) Test Environment Independence, and (4) Repeatability of Security Testing.

(1) Coverage Analysis. Security testing requires that precise, extensive test coverage be obtained during TCB testing. Test coverage analysis should be based on coverage of test conditions derived from the Descriptive Top-Level Specification (DTLS)/Formal Top-Level Specification (FTLS), the security and accountability model conditions, the TCB isolation and noncircumventability properties, and the individual TCB-primitive implementation. Without covering such test conditions, it would be impossible to claim reasonably that the tests cover specific security checks in a demonstrable way. Whenever both DTLS and FTLS and security and accountability models are unavailable or are not required, test conditions should be derived from documented protection philosophy and resource isolation requirements [13]. It would be impossible to reasonably claim that the implementation of a specific security check in a TCB primitive is correct without individual TCB-primitive coverage. In these checks a TCB primitive may deal differently with different parameters. In normal-mode testing, however, using user-level programs makes it difficult to guarantee significant coverage of TCB-primitive implementation while eliminating redundant tests

that appear when multiple TCB primitives share the same security checks (a common occurrence in TCB kernels).

The role of coverage analysis in the generation of test plans is discussed in Section 3.5.2, and illustrated in Sections 3.7.1.3-3.7.3.3.

(2) Reduction of Cyclic Test Dependencies. Comprehensive security testing suggests that cyclic test dependencies be reduced to a minimum or eliminated whenever possible. A cyclic test dependency exists between a test program for TCB primitive A and TCB primitive B if the test program for TCB primitive A invokes TCB primitive B, and the test program for TCB primitive B invokes TCB primitive A. The existence of cyclic test dependencies casts doubts on the level of assurance obtained by TCB tests. Cyclic test dependencies cause circular arguments and assumptions about test coverage and, consequently, the interpretation of the test results may be flawed. For example, the test program for TCB primitive A, which depends on the correct behavior of TCB primitive B, may not discover flaws in TCB primitive A because such flaws may be masked by the behavior of B, and vice versa. Thus, both the assumptions (1) that the TCB primitive B works correctly, which must be made in the test program for TCB primitive A, and (2) that TCB primitive A works correctly, which must be made in the test program for TCB primitive B, are incorrect. The elimination of cyclic test dependencies could be obtained only if the TCB is instrumented with additional code and data structures an impossibility if TCB isolation and noncircumventability are to be maintained in normal mode of operation.

An example of cyclic test dependencies, and of their removal, is provided in Section 3.7.5.

(3) Test Environment Independence. To minimize test program and test environment dependencies the following should be reinitialized for different TCB-primitive tests: user accounts, user groups, test objects, access privileges, and user security levels. Test environment initialization may require that the number of different test objects to be created and logins to be executed become very large. Therefore, in practice, complete TCB testing cannot be carried out manually. Testing should be automated whenever possible. Security test automation is discussed in Section 3.2.

(4) Repeatability of Security Testing. TCB verifiability through security testing requires that the results of each TCB-primitive test be repeatable. Without test repeatability it would be impossible to evaluate developers' TCB test suites independently of the TCB developers. Independent TCB testing may yield different outcomes from those expected if testing is not repeatable. Test repeatability by evaluation teams requires that test plans and procedures be documented in an accurate manner.

### 3.2 TEST AUTOMATION

The automation of the test procedures is one of the most important practical objectives of security testing. This objective is important for at least three reasons. First, the procedures for test environment initialization include a large number of repetitive steps that do not require operator intervention, and therefore, the manual performance of these steps may introduce avoidable errors in the test procedures. Second, the test procedures must be carried out repeatedly once for every system generation (e.g., system build) to ensure that security errors have not been introduced

during system maintenance. Repeated manual performance of the entire test suite may become a time consuming, error-prone activity. Third, availability of automated test suites enables evaluators to verify both the quality and extent of a vendor's test suite on an installed system in an expeditious manner. This significantly reduces the time required to evaluate that vendor's test suite.

The automation of most test procedures depends to a certain extent on the nature of the TCB interface under test. For example, for most TCB-primitive tests that require the same type of login, file system and directory initialization, it is possible to automate the tests by grouping test procedures in one or several user-level processes that are initiated by a single test-operator login. However, some TCB interfaces, such as the login and password change interfaces, must be tested from a user and administrator terminal. Similarly, the testing of the TCB interface primitives of B2 to A1 systems available to users only through trusted-path invocation requires terminal interaction with the test operator. Whenever security testing requires terminal interaction, test automation becomes a challenging objective.

Different approaches to test automation are possible. First, test designers may want to separate test procedures requiring terminal interaction (which are not usually automated), from those that do not require terminal interaction (which are readily amenable to automation). In this approach, the minimization of the number of test procedures that require terminal interaction is recommended.

Second, when test procedures requiring human-operator interaction cannot be avoided, test designers may want to connect a workstation to a terminal line and simulate the terminal activity of a human test operator on the workstation. This enables the complete automation of the test environment initialization and execution procedures, but not necessarily of the result identification and analysis procedure. This approach has been used in the testing of the Secure Xenix™ TCB. The commands issued by the test workstation that simulates the human-operator commands are illustrated in the appendix of reference [9].

Third, the expected outcome of each test should be represented in the same format as that assumed by the output of the TCB under test and should be placed in files of the workstation simulating a human test operator. The comparison between the outcome files and the test result files (transferred to the workstation upon test completion) can be performed using simple tools for file comparisons available in most current operating systems. The formatting of the outcome files in a way that allows their direct comparison with the test program output is a complex process. In practice, the order of the outcomes is determined only at the time the test programs are written, and sometimes only at execution time. Automated analysis of test results is seldomly done for this reason. To aid analysis of test results by human operators, the test result outputs can label and time-stamp each test. Intervention by a human test operator is also necessary in any case of mismatches between

obtained  
test results and expected outcomes.

An approach to automating security testing using Prolog is presented in reference [20].

### 3.3 TESTING APPROACHES

All approaches to security functional testing require the following four major steps: (1) the development of test plans (i.e., test conditions, test data including test outcomes, and test coverage analysis) and execution for each TCB primitive, (2) the definition of test procedures, (3) the development of test programs, and (4) the analysis of the test results. These steps are not independent of each other in all methods. Depending upon how these steps are performed in the context of security testing, three approaches can be identified: the monolithic (black-box) testing approach, the functional-synthesis (white-box) testing approach, and a combination of the two approaches called the gray-box testing approach.

In all approaches, the functions to be tested are the security-relevant functions of each TCB primitive that are visible to the TCB interface. The definition of these security functions is given by:

Classes C1 and C2. System documentation defining a system protection philosophy, mechanisms, and system interface operations (e.g., system calls).

Class B1. Informal interpretation of the (informal) security model and the system documentation.

Classes b2 and B3. Descriptive Top-Level Specifications (DTLSs) of the TCB and by the interpretation of the security model that is supposed to be implemented by the TCB functions.

Class A1. Formal Top-Level Specifications (FTLSs) of the TCB and by the interpretation of the security model that is supposed to be implemented by the TCB functions.

Thus, a definition of the correct security function exists for each TCB primitive of a system designed for a given security class. In TCB testing, major distinctions between the approaches discussed in the previous section appear in the areas of test plan generation (i.e., test condition, test data, and test coverage analysis). Further distinctions appear in the ability to eliminate redundant TCB-primitive tests without loss of coverage. This is important for TCB testing because a large number of access checks and access check sequences performed by TCB kernels are shared between different kernel primitives.

#### 3.3.1 Monolithic (Black-Box) Testing

The application of the monolithic testing approach to TCBs and to trusted processes is outlined in reference [2]. The salient features of this approach to TCB testing are the following: (1) the test condition selection is based on the TCSEC requirements and include discretionary and mandatory security, object reuse, labeling, accountability, and TCB isolation; (2) the test conditions for each TCB primitive should be generated from the chosen interpretation of each security function and primitive as defined above (for each security class). Very seldom is the relationship between the model interpretation and the generated test conditions, data, and programs shown explicitly (3 and

4]. Without such a relationship, it is difficult to argue coherently that all relevant security features of the given system are covered.

The test data selection must ensure test environment independence for unrelated tests or groups of tests (e.g., discretionary vs. mandatory tests). Environment independence requires, for example, that the subjects, objects, and access privileges used in unrelated tests or groups of tests must differ in all other tests or group of tests.

The test coverage analysis, which usually determines the extent of the testing for any TCB primitive, is used to delimit the number of test sets and programs. In the monolithic approach, the test data is usually chosen by boundary-value analysis. The test data places the test program directly above, or below, the extremes of a set of equivalent inputs and outputs. For example, a boundary is tested in the case of the "read" TCB call to a file by showing that (1) whenever a user has the read privilege for that file, the read TCB call succeeds; and (2) whenever the read privilege for that file is revoked, or whenever the file does not exist, the read TCB call fails. Similarly, a boundary is tested in the case of TCB-call parameter validation by showing that a TCB call with parameters passed by reference (1) succeeds whenever the reference points to an object in the caller's address space, and (2) fails whenever the reference points to an object in another address space (e.g., kernel space or other user spaces).

To test an individual boundary condition, all other related boundary conditions must be satisfied. For example, in the case of the "read" primitive above, the test call must not try to read beyond the limit of a file since the success/failure of not reading/reading beyond this limit represents a different, albeit related, boundary condition. The number of individual boundary tests for N related boundary conditions is of the order  $2N$  (since both successes and failures must be tested for each of the N conditions). Some examples of boundary-value analysis are provided in [2] for security testing, and in [5] and [6] for security-unrelated functional testing.

The monolithic testing approach has a number of practical advantages. It can always be used by both implementors and users (evaluators) of TCBs. No specific knowledge of implementation details is required because there is no requirement to break the TCB (e.g., kernel) isolation or to circumvent the TCB protection mechanism (to read, modify, or add to TCB code). Consequently, no special tools for performing monolithic testing are required. This is particularly useful in processor hardware testing when only descriptions of hardware/firmware implemented instructions, but no internal hardware/firmware design documents, are available.

The disadvantages of the monolithic approach are apparent. First, it is difficult to provide a precise coverage assessment for a set of TCB-primitive tests, even though the test selection may cover the

entire set of security features of the system. However, no coverage technique other than boundary-value analysis can be more adequate without TCB code analysis. Second, the elimination of redundant TCB-primitive tests without loss of coverage is possible only to a limited extent; i. e., in the case of access-check dependencies (discussed below) among TCB-primitive specifications. Third, in the context of TCB testing, the monolithic approach cannot cope with the problem of cyclic dependencies among test programs. Fourth, lack of TC code analysis precludes the possibility of distinguishing between design and implementation code errors in all but a few special cases. Also, it precludes the discovery of spurious code within the TCB—a necessary condition for Trojan Horse analysis.

In spite of these disadvantages, monolithic functional testing can be applied successfully to TCB primitives that implement simple security checks and share few of these checks (i. e., few or no redundant tests would exist). For example, many trusted processes have these characteristics, and thus this approach is adequate.

### 3.3.2 Functional-Synthesis (White-Box) Testing

Functional-synthesis-based testing requires the test of both functions implemented by each program (e. g., program of a TCB primitive) as a whole and functions implemented by internal parts of the program. The internal program parts correspond to the functional ideas used in building the program. Different forms of testing procedures are used depending upon different kinds of functional synthesis (e. g., control, algebraic, conditional, and iterative synthesis described in [1] and [7]). As pointed out in [9], only the control synthesis approach to functional testing is suitable for security testing.

In control synthesis, functions are represented as sequences of other functions. Each function in a sequence transforms an input state into an output state, which may be the input to another function.

Thus, a control synthesis graph is developed during program development and integration with nodes representing data states and arcs representing state transition functions. The data states are defined by the variables used in the program and represent the input to the state transition functions.

The assignment of program functions, procedures, and subroutines to the state transition functions of the graph is usually left to the individual programmer's judgment. Examples of how the control synthesis graphs are built during the program development and integration phase are given in [1] and [7].

The suitability of the control synthesis approach to TCB testing becomes apparent when one identifies the nodes of the control synthesis graph with the access checks within the TCB and the arcs with data states and outcomes of previous access checks. This representation, which is the dual of the traditional control synthesis graphs [9], produces a kernel access-check graph (ACG). This representation is useful because in TCB testing the primary access-check concerns are those of

(1) missing checks within a sequence of required checks, (2) wrong sequences of checks, and (3) faulty or incomplete access checks. (Many of the security problems identified in the Multics kernel design project existed because of these broad categories of inadequate access checks [8].) It is more suitable than the traditional control-synthesis graph because major portions of a TCB, namely the kernel, have comparatively few distinct access checks (and access-check sequences) and a large number of object types and access privileges that have the same access-check sequences for different TCB primitives [9]. (However, this approach is less advantageous in trusted process testing because trusted processes-unlike kernels-have many different access checks and few shared access sequences.) These objects cause the same data flow between access check functions and, therefore, are combined as graph arcs.

The above representation of the control synthesis graph has the advantage of allowing the reduction of the graph to the subset of kernel functions that are relevant to security testing. In contrast, a traditional graph would include (1) a large number of other functions (and, therefore, graph arcs), and (2) a large number of data states (and, therefore, graph nodes). This would be both inadequate and unnecessary. It would be inadequate because the presence of a large number of security-irrelevant functions (e.g., functions unrelated to security or accountability checks or to protection mechanisms) would obscure the role of the security-relevant ones, making test coverage analysis a complex and difficult task. It would be unnecessary because not only could security-irrelevant functions be eliminated from the graph but also the flows of different object types into the same access check function could be combined, making most object type-based security tests unnecessary.

Any TCB-primitive program can be synthesized at the time of TCB implementations as a graph of access-checking functions and data flow arcs. Many of the TCB-primitive programs share both arcs and nodes of the TCB graph. To build an access-check graph, one must identify all access-check functions, their inputs and outputs, and their sequencing. A typical input to an access-check function consists of an object identifier, object type and required access privileges. The output consists of the input to the next function (as defined above) and, in most cases, the outcome of the function check. The sequencing information for access-check functions consists of (1) the ordering of these functions, and (2) the number of arc traversals for each arc. An example of this is the sequencing of some access check functions that depend on the object types.

Test condition selection in the control-synthesis approach can be performed so that all the above access check concerns are satisfied. For example, test conditions must identify missing discretionary, mandatory, object reuse, privilege-call, and parameter validation checks (or parts of those checks). It also must identify access checks that are out of order, and faulty or incomplete checks, such as being able to truncate a file for which the modify privilege does not exist. The test conditions must

also be based on the security model interpretation to the same extent as that in the monolithic approach.

The test coverage in this approach also refers to the delimitation of the test data and programs for each TCB primitive. Because many of the access-check functions, and sequences of functions, are common to many of the kernel primitives (but not necessarily to trusted-process primitives), the synthesized kernel (TCB) graph is fairly small. Despite this the coverage analysis cannot rely on individual arc testing for covering the graph. The reason is that arc testing does not force the testing of access checks that correspond to combinations of arcs and thus it does not force coverage of all relevant sequences of security tests. Newer test coverage techniques for control synthesis graphs, such as data-flow testing [9, 10, and 11] provide coverage of arc combinations and thus are more appropriate than those using individual arc testing.

The properties of the functional-synthesis approach to TCB testing appear to be orthogonal to those of monolithic testing. Consider the disadvantages of functional-synthesis testing. It is not as readily usable as monolithic testing because of the lack of detailed knowledge of system internals. Also, it helps remove very few redundant tests whenever few access check sequences are shared by TCB primitives (as is the case with most trusted-process primitives).

Functional-synthesis-based testing, however, has a number of fundamental advantages. First, the coverage based on knowledge of internal program structure (i.e., code structure of a kernel primitive) can be more extensive than in the monolithic approach [1 and 7]. A fairly precise assessment of coverage can be made, and most of the redundant tests can be identified. Second, one can distinguish between TCB-primitive program failures and TCB-primitive design failures, something nearly impossible with monolithic testing. Third, this approach can help remove cyclic test dependencies. By removing all, or a large number of redundant tests, one removes most cyclic test dependencies (example of Section 3.7.5).

TCB code analysis becomes necessary whenever a graph synthesis is done after a TCB is built. Such analysis helps identify spurious control paths and code within a TCB—a necessary condition for Trojan Horse discovery. (In such a case, a better term for this approach would be functional-analysis-based testing.)

### 3.3.3 Gray-Box Testing

Two of the principal goals of security testing have been (1) the elimination of redundant tests through systematic test-condition selection and coverage analysis, and (2) the elimination of cyclic dependencies between the test programs. Other goals, such as test repeatability, which is also considered important, can be attained through the same means as those used for the other methods.

The elimination of redundant TCB-primitive tests is a worthwhile goal for the obvious reason that it reduces the amount of testing effort without loss of coverage. This allows one to determine a smaller nucleus of tests that must be carried out extensively. The overall TCB assurance may increase due to the judicious distribution of the test effort. The elimination of cyclic dependencies

among the TCB-primitive test programs is also a necessary goal because it helps establish a rigorous test order without making circular assumptions of the behavior of the TCB primitives. Added assurance is therefore gained.

To achieve the above goals, the gray-box testing approach combines monolithic testing with functional-synthesis-based testing in the test selection and coverage areas. This combination relies on the elimination of redundant tests through access-check dependency analysis afforded by monolithic testing. It also relies on the synthesis of the access-check graph from the TCB code as suggested by functional-synthesis-based testing (used for further elimination of redundant tests). The combination of these two testing methods generates a TCB-primitive test order that requires increasingly fewer test conditions and data without loss of coverage.

A significant number of test conditions and associated tests can be eliminated by the use of the access-check graph of TCB kernels. Recall that each kernel primitive may have a different access-check graph in principle. In practice, however, substantial parts of the graphs overlap. Consequently, if one of the graph paths is tested with sufficient coverage for a kernel primitive, then test conditions generated for a different kernel primitive whose graph overlaps with the first need only include the access checks specific to the latter kernel primitive. This is true because by the definition of the access-check graph, the commonality of paths means that the same access checks are performed in the same sequence, on the same types of objects and privileges, and with the same outcomes (e. g., success and failure returns). The specific access checks of a kernel primitive, however, must also show that the untested subpath(s) that has not been tested, of that kernel primitive, joins the tested path.

(A subset of the access-check and access-graph dependencies for the access, open, read, write, fcntl, ioctl, opensem, waltsem and slgsem primitives of Unix<sup>TM</sup>-like kernels are illustrated in Figures 1 and 2, pages 23 and 24. The use of these dependencies in the development of test plans, especially in coverage analysis, is illustrated in Sections 3.7.2.3 and 3.7.3.3; namely, in the test plans for access, open, and read. Note that the arcs shown in Figure 2, page 24 include neither complete flow-of-control information nor complete sets of object types, access-checks per call, and call outcome.)

### 3.4 RELATIONSHIP WITH THE TCSEC SECURITY TESTING REQUIREMENTS

The TCSEC security testing requirements and guidelines (i.e., Part 1 and Section 10 of the TCSEC) help define different approaches for security testing. They are particularly useful for test condition generation and test coverage. This section reviews these requirements in light of security testing approaches defined in Section 3.3.

Security Class C1

## Test Condition Generation

"The security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation." [TCSEC Part I, Section 2.1]

For this class of systems, the test conditions should be generated from the system documentation which includes the Security Features User's Guide (SFUG), the Trusted Facility Manual (TFM), the system reference manual describing each TCB primitive, and the design documentation defining the protection philosophy and its TCB implementation. Both the SFUG and the manual pages, for example, illustrate how the identification and authentication mechanisms work and whether a particular TCB primitive contains relevant security and accountability mechanisms. The Discretionary Access Control (DAC) and the identification and authentication conditions enforced by each primitive (if any) are used to define the test conditions of the test plans.

## Test Coverage

"Testing shall be done to assure that there are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCB." [TCSEC, Part I, Section 2.1]

"The team shall independently design and implement at least five system-specific tests in an attempt to circumvent the security mechanisms of the system." [TCSEC, Part II, Section 10]

The above TCSEC requirements and guidelines define the scope of security testing for this security class. Since each TCB primitive may include security-relevant mechanisms, security testing shall include at least five test conditions for each primitive. Furthermore, because source code analysis is neither required nor suggested for class C1 systems, monolithic functional testing (i.e., a black-box approach) with boundary-value coverage represents an adequate testing approach for this class. Boundary-value coverage of each test condition requires that at least two calls of each TCB primitive be made, one for the positive and one for the negative outcome of the condition. Such coverage may also require more than two calls per condition. Whenever a TCB primitive refers to multiple types of objects, each condition is repeated for each relevant type of object for both its positive and negative outcomes. A large number of test calls may be necessary for each TCB primitive because each test condition may in fact have multiple related conditions which should be tested independently of each other.

## Security Class C2

### Test Condition Generation

"Testing shall also include a search for obvious flaws that would allow violation of resource isolation, or that would permit unauthorized access to the audit and authentication data." [TCSEC, Part I, Section 2.2]

These added requirements refer only to new sources of test conditions, but not to a new testing approach nor to new coverage methods. The following new sources of test conditions should be considered:

(1) Resource isolation conditions. These test conditions refer to all TCB primitives that implement specific system resources (e.g., object types or system services). Test conditions for TCB primitives implementing services may differ from those for TCB primitives implementing different types of objects. Thus, new conditions may need to be

generated for TCB services. The mere repetition of test conditions defined for other TCB primitives may not be adequate for some services.

(2) Conditions for protection of audit and authentication data. Because both audit and authentication mechanisms and data are protected by the TCB, the test conditions for the protection of these mechanisms and their data are similar to those which show that the TCB protection mechanisms are tamperproof and noncircumventable. For example, these conditions show that neither privileged TCB primitives nor audit and user authentication files are accessible to regular users.

#### Test Coverage

Although class C1 test coverage already suggests that each test condition be covered for each type of object, coverage of resource-specific test conditions also requires that each test condition be covered for each type of service (whenever the test condition is relevant to a service). For example, the test conditions which show that direct access to a shared printer is denied to a user shall be repeated for a shared tape drive with appropriate modification of test data (i.e., test environments set up, test parameters and outcomes—namely, the test plan structure discussed in Section 3.5).

#### Security Class B1

##### Test Condition Generation

The objectives of security testing ". . . shall be: to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy enforced by the TCB; as well as to ensure that no subject (without authorization to do so) is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users." [TCSEC, Part I, Section 3.1]

The security testing requirements of class B1 are more extensive than those of both classes C1 and C2, both in test condition generation and in coverage analysis. The source of test conditions

referring to users' access to data includes the mandatory and discretionary policies implemented by the TCB. These policies are defined by an (informal) policy model whose interpretation within the TCB allows the derivation of test conditions for each TCB primitive. Although not explicitly stated in the TCSEC, it is generally expected that all relevant test conditions for classes C1 and C2

also would be used for a class B1 system.

##### Test Coverage

"All discovered flaws shall be removed or neutralized and the TCB retested to demonstrate that they have been eliminated and that new flaws have not been introduced." [TCSEC, Part I, Section 3.1]

"The team shall independently design and implement at least fifteen system specific tests in an attempt to circumvent the security mechanisms of the system." [TCSEC, Part II, Section 10]

Although the coverage analysis is still boundary-value analysis, security testing for class B1 systems suggests that at least fifteen test conditions be generated for each TCB primitive that contains security-relevant mechanisms to cover both mandatory and discretionary policy. In practice, however, a substantially higher number of test conditions is generated from interpretations

of the (informal) security model. The removal or the neutralization of found errors and the retesting of the TCB requires no additional types of coverage analysis.

## Security Class B2

### Test Condition Generation

"Testing shall demonstrate that the TCB implementation is consistent with the descriptive top-level specification." [TCSEC, Part I, Section 3.2]

The above requirement implies that both the test conditions and coverage analysis of class B2 systems are more extensive than those of class B1. In class B2 systems every access control and accountability mechanism documented in the DTLS (which must be complete as well as accurate) represents a source of test conditions. In principle the same types of test conditions would be generated for class B2 systems as for class B1 systems, because (1) in both classes the test conditions could be generated from interpretations of the security policy model (informal at B1 and formal at B2), and (2) in class B2 the DTLS includes precisely the interpretation of the security policy model. In practice this is not the case however, because security policy models do not model a substantial number of mechanisms that are, nevertheless, included in the DTLS of class B2 systems. (Recall that class B1 systems do not require a DTLS of the TCB interface.) The number and type of test conditions can therefore be substantially higher in a class B2 system than those in a class B1 system because the DTLS for each TCB primitive may contain additional types of mechanisms, such as those for trusted facility management.

### Test Coverage

It is not unusual to have a few individual test conditions for at least some of the TCB primitives. As suggested in the gray-box approach defined in the previous section, repeating these conditions for many of the TCB primitives to achieve uniform coverage can be both impractical and unnecessary. Particularly this is true when these primitives refer to the same object types and services. It is for this reason and because source-code analysis is required in class B2 systems to satisfy other requirements that the use of the gray-box testing approach is recommended for the parts of the TCB in which primitives share a substantial portion of their code. Note that the DTLS of any system does not necessarily provide any test conditions for demonstrating the tamperproofness and noncircumventability of the TCB. Such conditions should be generated separately.

## Security Class B3

### Test Condition Generation

The only difference between classes B2 and B3 requirements of security testing reflects the need to discover virtually all security policy flaws before the evaluation team conducts its security testing exercise. Thus, no additional test condition requirements appear for class B3 testing. Note that the DTLS does not necessarily provide any test conditions for demonstrating the TCB is tamperproof and noncircumventable as with class B2 systems. Such conditions should be generated separately.

## Test Coverage

"No design flaws and no more than a few correctable implementation flaws may be found during testing and there shall be reasonable confidence that few remain." [TCSEC, Part I, Section 3.3]

The above requirement suggests that a higher degree of confidence in coverage analysis is required for class B3 systems than for class B2 systems. It is for this reason that it is recommended the gray-box testing approach be used extensively for the entire TCB kernel, and data-flow coverage be used for all independent primitives of the kernel (namely, the gray-box method in Section 3.3 above).

## Security Class A1

The only differences between security testing requirements of classes B3 and A1 are (1) the test conditions shall be derived from the FTLS, and (2) the coverage analysis should include at least twenty-five test conditions for each TCB primitive implementing security functions. Neither requirement suggests that a different testing method than that recommended for class B3 systems is required.

## 3.5 SECURITY TEST DOCUMENTATION

This section discusses the structure of typical test plans, test logs, test programs, test procedures, and test reports. The description of the test procedures necessary to run the tests and to examine the test results is also addressed. The documentation structures presented are meant to provide the system developers with examples of good test documentation.

### 3.5.1 Overview

The work plan for system testing should describe how security testing will be conducted and should contain the following information:

- Test-system configuration for both hardware and software.
- Summary test requirements.
- Procedures for executing test cases.
- Step-by-step procedures for each test case.
- Expected results for each test step.
- Procedures for correcting flaws uncovered during testing.
- Expected audit information generated by each test case (if any).

See Section 3.7.7, "Relationship with the TCSEC Requirements."

### 3.5.2 Test Plan

Analysis and testing of mechanisms, assurances and/or documentation to support the TCSEC security testing requirements are accomplished through test plans. The test plans should be sufficiently complete to cover each identified security mechanism and should be conducted with sufficient depth to provide reasonable assurance that any bugs not found lie within the

acceptable

risk threshold for the class of the system being evaluated. A test plan consists of test conditions, test data, and coverage analysis.

### 3.5.2.1 Test Conditions

A test condition is a statement of a security-relevant constraint that must be satisfied by a TCB

primitive. Test conditions should be derived from the system's DTLS/FTLS, from the interpretation

of the security and accountability models (if any), from TCB isolation and noncircumventability properties, and from the specifications and implementation of the individual TCB primitive under test. If neither DTLS/FTLS nor models are required, then test conditions should be derived from the informal policy statements, protection philosophy and resource isolation requirements.

#### (1) Generation of Model or Policy-Relevant Test Conditions

This step suggests that a matrix of TCB primitives and the security model(s) or requirement components be built. Each entry in the matrix identifies the security relevance of each primitive (if

any) in a security model or requirement area and the relevant test conditions. For example, in the mandatory access control area of security policy, one should test the proper object labeling by the

TCB, the "compatibility" property of the user created objects, and the TCB implemented authorization rules for subject access to objects. One should also test that the security-level relationships are properly maintained by the TCB and that the mandatory access works independently of, and in conjunction with, the discretionary access control mechanism. In the discretionary access control area, one may include tests for proper user/group identifier selection,

proper user inclusion/exclusion, selective access distribution/revocation using the access control

list (ACL) mechanism, and access review.

Test conditions derived from TCB isolation and noncircumventability properties include conditions that verify (1) that TCB data structures are inaccessible to user level programs, (2) that

transfer of control to the TCB can take place only at specified entry points, which cannot be bypassed

by user-level programs, (3) that privileged entry points into the TCB cannot be used by user level

programs, and (4) that parameters passed by reference to the TCB are validated.

Test conditions derived from accountability policy include conditions that verify that user identification and authentication mechanisms operate properly. For example, they include conditions that verify that only sufficiently complex passwords can be chosen by any user, that the

password aging mechanism forces reuse at stated intervals, and so on. Other conditions of identification and authentication, such as those that verify that the user login level is dominated by

the user's maximum security level, should also be included. Furthermore, conditions that verify that the user commands included in the trusted path mechanism are unavailable to the user program

interface of the TCB should be used. Accountability test conditions that verify the correct operation

of the audit mechanisms should also be generated and used in security testing.

The security relevance of a TCB primitive can only be determined from the security policy,

accountability, and TCB isolation and noncircumventability requirements for classes B1 to A1, or from protection philosophy and resource isolation requirements for classes C1 and C2. Some TCB primitives are security irrelevant. For example, TCB primitives that never allow the flow of information across the boundaries of an accessible object are always security irrelevant and need not be tested with respect to the security or accountability policies. The limitation of information flow to user-accessible objects by the TCB primitives implementation, however, needs to be tested by TCB-primitive-specific tests. A general example of security-irrelevant TCB primitives is provided by those primitives which merely retrieve the status of user-owned processes at the security level of the user.

## (2) Generation of TCB-Primitive-Specific Test Conditions

The selection of test conditions used in security testing should be TCB-primitive-specific. This helps remove redundant test conditions and, at the same time, helps ensure that significant test coverage is obtained. For example, the analysis of TCB-primitive specifications to determine their access-check dependencies is required whenever the removal of redundant TCB-primitive tests is considered important. This analysis can be applied to all testing approaches. The specification of a TCB primitive A is access-check dependent on the specification of a TCB primitive B if a subset of the access checks needed in TCB primitive A are performed in TCB primitive B, and if a TCB call to primitive B always precedes a TCB call to primitive A (i.e., a call to TCB primitive A fails if the call to TCB primitive B has not been done or has not completed with a successful outcome). In case of such dependencies, it is sufficient to test TCB primitive B first and then to test only the access checks of TCB primitive A that are not performed in TCB primitive B. Of course, the existence of the access-check dependency must be verified through testing.

As an example of access-check dependency, consider the fork and the exit primitives of the Secure Xenix<sup>TM</sup> kernel. The exit primitive always terminates a process and sends a return code to the parent process. The mandatory access check that needs to be tested in exit is that the child's process security level equals that of the parent's process. However, the specifications of the exit primitive are access-check dependent on the specifications of the fork primitive (1) because an exit call succeeds only after a successfully completed fork call is done by some parent process, and (2) because the access check, that the child's process level always equals that of the parent's process level, is already performed during the fork call. In this case, no additional mandatory access test is needed for exit beyond that performed for fork. Similarly, the sigsem and the waitsem primitives of some Unix<sup>TM</sup> based kernels are access-check dependent on the opensem primitive, and no additional mandatory or discretionary access checks are necessary.

However, in the case of the read and the write primitives of Unix<sup>TM</sup> kernels, the specifications of which are also access-check dependent on both the mandatory and the discretionary checks of the open primitive, additional tests are necessary beyond those done for open. In the case of the read primitive one needs to test that files could only be read if they have been opened for reading, and that reading beyond the end of a file is impossible after one tests the dependency of read

on the specification of open. Additional tests are also needed for other primitives such as fcntl and loctl; their specifications are both mandatory and discretionary access-check dependent on the open primitives for files and devices. Note that in all of the above examples a large number of test conditions and associated tests are eliminated by using the notion of access check dependency of specifications because, in general, less test conditions are generated for access check dependency testing than for the security testing of the primitive itself.

The following examples are given in references [3] and [4]: (1) of the generation of such constraints from security models, (2) of the predicates, variables, and object types used in constraint definition, and (3) of the use of such constraints in test conditions for processor instructions (rather than for TCB primitives).

See Section 3.7.7, "Relationship with the TCSEC Requirements."

### 3.5.2.2 Test Data

"Test data" is defined as the set of specific objects and variables that must be used to demonstrate that a test condition is satisfied by a TCB primitive. The test data consist of the definition of the initialization data for the test environment, the test parameters for each TCB primitive, and the expected test outcomes. Test data generation is as important as test condition generation because it ensures that test conditions are exercised with appropriate coverage in the test programs, and that test environment independence is established whenever it is needed.

To understand the importance of test data generation consider the following example. Suppose that all mandatory tests must ensure that the "hierarchy" requirement of the mandatory policy interpretation must be tested for each TCB primitive. (Expansion on this subject, i.e., the nondecreasing security level requirement for the directory hierarchy can be found in [12].) What directory hierarchy should one set up for testing this requirement and at the same time argue that all possible directory hierarchies are covered for all tests? A simple analysis of this case shows that there are two different forms of upgraded directory creation that constitute an independent basis for all directory hierarchies (i.e., all hierarchies can be constructed by the operations used for one or the other of the two forms, or by combinations of these operations). The first form is illustrated in Figure 3a representing the case whereby each upgraded directory at a different level is upgraded from a single lower level (e.g., system low). The second form is illustrated in Figure 3b and represents the case whereby each directory at a certain level is upgraded from an immediately lower level. A similar example can be constructed to show that combinations of security level definitions used for mandatory policy testing cover all security level relationships.

Test data for TCB primitives should include several items such as the TCB primitive input data, TCB primitive return result and success/failure code, object hierarchy definition, security

level used

for each process/object, access privileges used, user identifiers, object types, and so on. This selection needs to be made on a test-by-test basis and on a primitive-by-primitive basis.

Whenever

environment independence is required, a different set of data is defined [2]. It is very helpful that

the naming scheme used for each data object helps identify the test that used that item.

Different

test environments can be easily identified in this way. Note that the test data selection should ensure

both coverage of model-relevant test conditions and coverage of the individual TCB primitives. This will be illustrated in an example in the next section.

See Section 3.7.7, "Relationship with the TCSEC Requirements."

### 3.5.2.3 Coverage Analysis

Test coverage analysis is performed in conjunction with the test selection phase of our approach.

Two classes of coverage analysis should be performed: model- or policy-dependent coverage and individual TCB primitive coverage.

#### (1) Model- or Policy-Dependent Coverage

In this class, one should demonstrate that the selected test conditions and data cover the interpretation of the security and accountability model and noncircumventability properties in all

areas identified by the matrix mentioned above. This is a comparatively simple task because model

coverage considerations drive the test condition and data selection. This kind of coverage includes

object type, object hierarchy, subject identification, access privilege, subject/object security level,

authorization check coverage, and so on. Model dependent coverage analysis relies, in general, on

boundary-value analysis.

#### (2) Individual TCB-Primitives Coverage

This kind of coverage includes boundary value analysis, data flow analysis of individual access-check graphs of TCB primitives, and coverage of dependencies. The examples of reference [2]

illustrate boundary-value analysis. Other forms of TCB-primitive coverage will be discussed in Section 3.7 of this guideline. For example, graph coverage analysis represents the determination that the test conditions and data exercise all the data flows for each TCB-primitive graph. This includes not only the traversal of all the graph access checks (i.e., nodes) but also of all the graph's

arcs and arc sequences required for each TCB primitive. (The example for access primitive of Unix<sup>TM</sup> kernels included in Section 3.7 explains this form of coverage. Data flow coverage is also

presented in [10] and [11] for security-unrelated test examples.)

Coverage analysis is both a qualitative and quantitative assessment of the extent to which the test

shows TCB-primitive compliance with the (1) design documentation, (2) resource isolation, (3) audit and authentication data protection, (4) security policy and accountability model conditions,

(5) DTLS/FTLS, as well as with those of the TCB isolation and noncircumventability properties. To achieve significant coverage, all security-relevant conditions derived from a TCB model and properties and DTLS/FTLS should be covered by a test, and each TCB-primitive test should cover

the implementation of its TCB primitive. For example, each TCB- primitive test should be performed for all independent object types operated upon by that TCB primitive and should test all independent security exceptions for each type of object.

See Section 3.7.7, "Relationship with the TCSEC Requirements."

### 3.5.3 Test Procedures

A key step in any test system is the generation of the test procedures (which are also known as "test scripts"). The major function of the test procedure is to ensure that an independent test operator or user is able to carry out the test and to obtain the same results as the test implementor. The procedure for each test should be explained in sufficient detail to enable repeatable testing. The test procedure should contain the following items to accomplish this:

(1) Environment Initialization Procedure. This procedure defines the login sequences and parameters, the commands for object and subject cleanup operations at all levels involved in the test, the choice of object names, the commands and parameters for object creation and initialization at the required levels, the required order of command execution, the initialization at the required levels, the initialization of different subject identifiers and access privileges (for the initialized objects) at all required levels, and the specification of the test program and command names and parameters used in the current test.

(2) Test Execution Procedure. The test procedure includes a description of the test execution from a terminal including the list of user commands, their input, and the expected terminal, printer, or file output.

(3) Result Identification Procedure. The test procedure should also identify the results file for a given test, or the criteria the test operator must use to find the results of each individual test in the test output file. The meaning of the results should also be provided.

See Section 3.7.7, "Relationship with the TCSEC Requirements."

Note: A system in which testing is fully automated eliminates the need for separate test procedure documentation. In such cases, the environment initialization procedures and the test execution procedures should be documented in the test data section of the test plans. Automated test operator programs include the built-in knowledge otherwise contained in test procedures.

### 3.5.4 Test Programs

Another key step of any test system is the generation of the test programs. The test programs for each TCB primitive consist of the login sequence, password, and requested security level. The security profile of the test operator and of the possible workstation needs to be defined a priori by the system security administrators to allow logins and environment initialization at levels

required in the test plan. After login, a test program invokes several trusted processes (e.g., "mkdir," "rmdir," in some UnixYM systems) with predetermined parameters in the test plan and procedure to initialize the test environment. A nucleus of trusted processes, necessary for the environment set up, are tested independently of a TCB primitive under test whenever possible and are assumed to be correct.

After the test environment is initialized, the test program (which may require multiple logins at different levels) issues multiple invocations to the TCB primitive under test and to other TCB primitives needed for the current test. The output of each primitive issued by the test programs is collected in a result file associated with each separate test and analyzed. The analysis of the test results that are collected in the results file is performed by the test operator. This analysis is a comparison between the results file and the expected outcome file defined by the test plan prior to the test run. Whenever the test operator detects a discrepancy between the two files he records a test error.

### 3.5.5 Test Log

A test log should be maintained by each team member during security testing. It is to capture useful information to be included later in the test report. The test log should contain:

- Information on any noteworthy observations.
- Modifications to the test steps.
- Documentation errors.
- Other useful data recorded during the testing procedure test results.

### 3.5.6 Test Report

The test report is to present the results of the security testing in a manner that effectively supports the conclusions reached from the security testing process and provides a basis for NCSC test team security testing. The test report should contain:

- Information on the configuration of the tested system.
- A chronology of the security testing effort.
- The results of functional testing including a discussion of each flaw uncovered.
- The results of penetration testing covering the results of successful penetrations.
- Discussion of the corrections that were implemented and of any retesting that was performed.

A sample test report format is provided in Section 3.7.

## 3.6 SECURITY TESTING OF PROCESSORS' HARDWARE/FIRMWARE

## PROTECTION MECHANISMS

The processors of a computer system include the Central Processing Units (CPU), Input/Output (I/O) processors, and application-oriented co-processors such as numerical co-processors and signal-analysis co-processors. These processors may include mechanisms capabilities, access privileges, processor-status registers, and memory areas representing TCB internal objects such as process control blocks, descriptor, and page tables. The effects of the processor protection mechanisms become visible to the system users through the execution of processor instructions and I/O commands that produce transformations of processor and memory registers. Transformations produced by every instruction or I/O command are checked by the processors protection mechanisms and are allowed only if they conform with the specifications defined by the processor reference manuals for that instruction. For few processors these transformations are specified formally and for less processors a formal (or informal) model of the protection mechanisms is given [3 and 4].

## 3.6.1 The Need for Hardware/Firmware Security Testing

Protection mechanisms of systems processors provide the basic support for TCB isolation, noncircumventability, and process address space separation. In general, processor mechanisms for the isolation of the TCB include those that (1) help separate the TCB address space and privileges from those of the user, (2) help enforce the transfer of control from the user address space to the TCB address space at specific entry points, and (3) help verify the validity of the user-level parameters passed to the TCB during primitive invocation. Processor mechanisms that support TCB noncircumventability include those that (1) check each object reference against a specific set of privileges, and (2) ensure that privileged instructions which can circumvent some of the protection mechanisms are inaccessible to the user. Protection mechanisms that help separate process address spaces include those using base and relocation registers, paging, segmentation, and combinations thereof.

The primary reason for testing the security function of a system's processors is that flaws in the design and implementation of processor-supported protection mechanisms become visible at the user level through the instruction set. This makes the entire system vulnerable because users can issue carefully constructed sequences of instructions that would compromise TCB and user security.

(User visibility of protection flaws in processor designs is particularly difficult to deny. Attempts to force programmers to use only high-level languages, such as PL1, Pascal, Algol, etc., which would obscure the processor instruction set, are counterproductive because arbitrary addressing patterns and instruction sequences still can be constructed through seemingly valid programs (i. e., programs that compile correctly). In addition, exclusive reliance on language compilers and on other subsystems for the purpose of obscuring protection flaws and denying users the ability to produce arbitrary addressing patterns is unjustifiable. One reason is that compiler verification is a particularly difficult task; another is that reliance on compilers and on other subsystems implies reliance on the diverse skills and interests of system programmers. Alternatively, hardware-based

attempts to detect instruction sequence patterns that lead to protection violations would only result in severe performance degradation.)

The additional reason for testing the security function of a system's processor is that, in general, a system's TCB uses at least some of the processor's mechanisms to implement its security policy.

Flawed protection mechanisms may become unusable by the TCB and, in some cases, the TCB may not be able to neutralize those flaws (e.g., make them invisible to the user). It should be noted that the security testing of the processor protection mechanisms is the most basic life-cycle evidence available in the context of TCSEC evaluations to support the claim that a system's reference notion is verifiable.

### 3.6.2 Explicit TCSEC Requirements for Hardware Security Testing

The TCSEC imposes very few explicit requirements for the security testing of a system's hardware and firmware protection mechanisms. Few interpretations can be derived from these requirements as a consequence. Recommendations for processor test plan generation and documentation, however, will be made in this guideline in addition to explicit TCSEC requirements. These recommendations are based on analogous TCB testing recommendations made herein.

#### Specific Requirements for Classes C1 and C2

The following requirements are included for security classes C1 and C2:

"The security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation."

The security mechanisms of the ADP system clearly include the processor-supported protection mechanisms that are used by the TCB and those that are visible to the users through the processor's instruction set. In principle it could be argued that the TCB security testing implicitly tests at least some processor mechanisms used by the TCB; therefore, no additional hardware testing is required for these mechanisms. All processor protection mechanisms that are visible to the user through the instruction set shall be tested separately regardless of their use by a tested TCB. In practice, nearly all processor protection mechanisms are visible to users through the instruction set. An exception is provided by some of the I/O processor mechanisms in systems where users cannot execute I/O commands either directly or indirectly.

#### Specific Requirements for Classes B1 to B3

In addition to the above requirements of classes C1 and C2, the TCSEC includes the following specific hardware security testing guidelines in Section 10 "A Guideline on Security Testing":

"The [evaluation] team shall have 'hands-on' involvement in an independent run of the test package used by the system developer to test security-relevant hardware and software.

The explicit inclusion of this requirement in the division B (i.e., classes B1 to B3) of the TCSEC guideline on security testing implies that the scope and coverage of the security-relevant hardware

testing and test documentation should be consistent with those of the TCB security testing for this division. Thus, the security testing of the processor's protection mechanisms for division B systems should be more extensive than for division C (i.e., C1 and C2) systems.

#### Specific Requirement for Class A1

In addition to the requirements for divisions C and B, the TCSEC includes the following explicit requirements for hardware and/or firmware testing:

"Testing shall demonstrate that the TCB implementation is consistent with the formal top-level specifications." [Security Testing requirement] and

"The DTLs and FTLs shall include those components of the TCB that are implemented as hardware and/or firmware if their properties are visible at the TCB interface." [Design Specification and Verification requirement]

The above requirements suggest that all processor protection mechanisms that are visible at the TCB interface should be tested. The scope and coverage of the security-relevant testing and test documentation should also be consistent with those of TCB security-relevant testing and test documentation for this division.

#### 3.6.3 Hardware Security Testing vs. System Integrity Testing

Hardware security testing and system integrity testing differ in at least three fundamental ways.

First, the scope of system integrity testing and that of hardware security testing is different. System integrity testing refers to the functional testing of the hardware/firmware components of a system including components that do not necessarily have a specific security function (i.e., do not include any protection mechanisms). Such components include the memory boards, busses, displays, adaptors for special devices, etc. Hardware security testing, in contrast, refers to hardware and firmware components that include protection mechanisms (e.g., CPU's and I/O processors). Failures of system components that do not include protection mechanisms may also affect system security just as they would affect reliability and system performance. Failures of components that include protection mechanisms can affect system security adversely. A direct consequence of the distinction between the scope of system integrity and hardware security testing is that security testing requirements vary with the security class of a system, whereas system integrity testing requirements do not.

Second, the time and frequency of system integrity and security testing are different. System integrity testing is performed periodically at the installation site of the equipment. System security testing is performed in most cases at component design and integration time. Seldom are hardware security test suites performed at the installation site.

Third, the responsibility for system integrity testing and hardware security testing is different. System integrity testing is performed by site administrators and vendor customer or field engineers. Hardware security testing is performed almost exclusively by manufacturers, vendors, and system

evaluators.

#### 3.6.4 Goals, Philosophy, and Approaches to Hardware Security Testing

Hardware security testing has the same general goals and philosophy as those of general TCB security testing. Hardware security testing should be performed for processors that operate in normal mode (as opposed to maintenance or test mode). Special probes, instrumentation, and special reserved op-codes in the instruction set should be unnecessary. Coverage analysis for each tested instruction should be included in each test plan. Cyclic test dependencies should be minimized, and testing should be repeatable and automated whenever possible.

In principle, all the approaches to security testing presented in Section 3.3 are applicable to hardware security testing. In practice, however, all security testing approaches reported to date have relied on the monolithic testing approach. This is the case because hardware security testing is performed on an instruction basis (often only descriptions of the hardware/firmware-implemented, but no internal hardware/firmware design details, are available to the test designers). The generation of test conditions is, consequently, based on instruction and processor documentation (e.g., on reference manuals). Models of the processor protection mechanisms and top-level specifications of each processor instruction are seldom available despite their demonstrable usefulness [3 and 4] and mandatory use [13, class A1] in security testing. Coverage analysis is restricted in practice to boundary-value coverage for similar reasons.

#### 3.6.5 Test Conditions, Data, and Coverage Analysis for Hardware Security Testing

Lack of DTLS and protection-model requirements for processors' hardware/firmware in the TCSEC between classes C1 and B3 makes the generation of test conditions for processor security testing a challenging task (i.e., class A1 requires that FTLS be produced for the user-visible hardware functions and thus these FTLS represent a source of test conditions). The generation of test data is somewhat less challenging because this activity is related to a specific coverage analysis method, namely boundary-value coverage, which implies that the test designer should produce test data for both positive and negative outcomes of any condition.

Lack of DTLS and of protection-model requirements for processors' hardware and firmware makes it important to identify various classes of security test conditions for processors that illustrate potential sources of test conditions. We partition these classes of test conditions into the following categories: (1) processor tests that help detect violations of TCB isolation and noncircumventability, (2) processor tests that help detect violations of policy, and (3) processor tests that help detect other generic flaws (e.g., integrity and denial of service flaws).

##### 3.6.5.1 Test Conditions for Isolation and Noncircumventability Testing

(1) There are tests which detect flaws in instructions that violate the separation of user and TCB (privileged) domain:

Included in this class are tests that detect flaws in bounds checking CPU and I/O processors, top- and bottom-of-the-stack frame checking, dangling references, etc. [4]. Tests within this class should include the checking of all addressing modes of the hardware/firmware. This includes single and multiple-level indirect addressing [3 and 4], and direct addressing with no operands (i.e., stack addressing), with a single operand and with multiple operands. Tests which demonstrate that all the TCB processor, memory, and I/O registers are inaccessible to users who execute nonprivileged instructions should also be included here.

This class also includes tests that detect instructions that do not perform or perform improper access privilege checks. An example of this is the lack of improper access privilege checking during multilevel indirections through memory by a single instruction. Proper page-or segment-presence bit checks as well as the proper invalidation of descriptors within caches during process switches should also be tested. All tests should ensure that all privilege checking is performed in all addressing modes. Tests which check whether a user can execute privileged instructions are also included here. Examples of such tests (and lack thereof) can be found in [3, 4, 22, and 23].

(2) There are tests that detect flaws in instructions that violate the control of transfer between domains:

Included in this class are tests that detect flaws that allow anarchic entries to the TCB domain (i.e., transfers to TCB arbitrary entry points and at arbitrary times), modification and/or circumvention of entry points, and returns to the TCB which do not result from TCB calls. Tests show that the local address space of a domain or ring is switched properly upon domain entry or return (e.g., in a ring-based system, such as SCOMP, Intel 80286-80386, each ring stack segment is selected properly upon a ring crossing).

(3) There are tests that detect flaws in instructions that perform parameters validation checks:

Included in this class are tests that detect improper checks of descriptor privileges, descriptor length, or domain/ring of a descriptor (e.g., Verify Read (VERR), Verify Write (VERW), Adjust Requested Privilege Level (ARPL), Load Access Rights (LAR), Load Segment Length (LSL) in the Intel 80286-80386 architecture [24], Argument Addressing Mode (AAM) in Honeywell SCOMP, [22 and 23], etc.).

### 3.6.5.2 Text Conditions for Policy-Relevant Processor Instructions

Included in this class are tests that detect flaws that allow user-visible processor instructions to allocate/deallocate objects in memory containing residual parts of previous objects and tests that detect flaws that would allow user-visible instructions to transfer access privileges in a way that is inconsistent with the security policy (e.g., ca

## Share this article



## Receive all the latest articles by email!

Receive Real-Time & Monthly WindowSecurity.com article updates in your mailbox. Enter your email below!  
Click for [Real-Time sample](#) & [Monthly sample](#)

## Become a WindowSecurity.com member!

Discuss your security issues with thousands of other network security experts. [Click here](#) to join!

---

[About Us](#) : [Email us](#) : [Product Submission Form](#) : [Advertising Information](#)

WindowsSecurity.com is in no way affiliated with Microsoft Corp. \*Links are sponsored by advertisers.

Copyright © 2008 [TechGenix Ltd.](#) All rights reserved. Please read our [Privacy Policy](#) and [Terms & Conditions](#).