Chapter 3
Test based security certifications

Abstract Test-based certifications are a powerful tool for software users to assess the extent to which a software product satisfies their security requirements. To exploit the full potential of this idea, it is necessary to have a shared way to describe the security properties to be tested, as well as to define the tests to be carried out and the testing environment itself. Since the early days of the US Department of Defense's *Trusted Computer System Evaluation Criteria (TCSEC)* [17], also known as *Orange Book*, several attempts have been made to standardize such descriptions, especially for operating system platforms; the trend has been toward increasing expressive power and complexity. A major result of this standardization activity is the *Common Criteria* (CC) , an ISO standard that defines a general test-based framework to specify, design and evaluate the security properties of IT products [12]. The introduction of CC came after almost two decades of joint work of many international organizations aimed at creating a common security standard that can be recognized at an international level. The rationale behind CC is to formalize the language used by customers, developers and security evaluators to have the same understanding when security requirements are specified and evaluated. In this chapter, after a brief primer to software testing, we will focus on test-based security certification, looking at the milestones that have led to the introduction of the CC standard; then, the general conceptual model underlying the CC standard will be explained.

3.1 Basic Notions on Software Testing

Software certification (see Chapter 1) is aimed at generating certificates demonstrating non-functional properties of software systems, such as the ones linked to dependability, security and safety. While the notion of interoperable software certificates as metadata items is a relatively new one, certification techniques build on well-known software validation and verification techniques. Therefore, an in-depth discussion of test-based security certification requires some background knowledge on the software testing process. This section will provide such a background, focusing on the notion of *risk-oriented* testing. Experienced readers may safely skip it. Our main reference here is the excellent book by Paul Ammann and Jeff Offutt [1] and its rich bibliography, as well as the recent book by Pezzè and Young [19].[1]

Informally, software development can be defined as the process of designing (and implementing) a software product which meets some (functional and non-functional) user requirements. In turn, software testing can be defined as the process of validating a software product's functionality, and, even more importantly from our point of view, of verifying that the software has all the desired non-functional properties (performance, robustness, security and the like) its users expect. Hopefully, the testing process will also reveal the software product's defects.

In practice, software testing is performed as an iterative process: in each iteration, some tests are designed and executed to reveal software problems, and the detected problems are fixed. One should not expect miracles from this procedure: software testing can reveal the presence of failures, but it cannot provide proof of their absence.[2]

In this book, we distinguish between test-based certification as proceeding from testing, and model-based certification as proceeding from abstract models of the software such as automata or graphs. Some researchers, including Ammann and Offutt [1] justifiably argue that testing is also driven by abstract models, which can be developed to provide a *black-box* or a *white-box* view. Occasionally, *black-box* testing is carried out by professional testers who do not need to be acquainted with the code. The main purpose of black-box testing is to assess the extent to which functional and non-functional user requirements are satisfied;[3] each requirement is checked by a set of *test cases*.[4] The most important component of a test case is the *test case value*, that is, the input values fed into the software under test during a test execution of it. To evaluate the results of a black-box test case, we must know in advance its *expected results* or, in other words, the result that will be produced executing the test if (and only if) the program satisfies its requirements. Strictly speaking, a test case is the combination of test case values and expected results, plus two additional components: *prefix* and *postfix* values. Prefix values are inputs that, while not technically test values, are necessary to put the software into the appropriate state to feed it with the actual test case values. Postfix values are inputs that need to be sent to the software after the test

[1] We will follow references [1, 19] in the remaining of this chapter.

[2] The original statement is "But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." E. W. Dijkstra, "The Humble Programmer", Turing Award Lecture, 1972, Communications of the ACM, 15(10).

[3] Black-box tests can also be used to discover failures, such as crashes.

[4] Test-related literature often uses the term test set to indicate a set of test cases.

case values have been fed into it (e.g., for displaying results or terminating the test execution). For the sake of conciseness, throughout this book we will loosely use the term *test case* as a synonym of "test case value". Determining a software product's expected behavior may well be easy for some toy function (what should a `float squareroot(int )` function output when fed with a test case value of *4*?), but it is a much harder nut to crack for more complex programs (what should an image-enhancement software output when fed with a test-case value like `sampleimage.gif`?)

White-box tests have a very different aim: to ensure that some aspects of some code property are exercised in the hope that this will reveal any errors not otherwise detected. Also, white-box test can be used to verify coding standards; for instance, those barring the interchangeable use of pointer arithmetic and array reference in C. Therefore, they may consist of code walkthroughs rather than of sample executions.[5] In *white-box* testing , tests are designed and validated by developers, or at least by people who know well the code of the program under test.

Both white- and black-box testing can be *risk-oriented.* Risk orientation simply means that when some pre-defined constraints to testing are specified, (e.g., ones related to maximum test cost and available time for carrying out the tests) both types of tests can be prioritized so that risks for the software user are minimized as far as possible [10]. From the perspective of a software user, we may define the risk of running a software product in a very simple way, as follows:

$$R = \sum_{\{p\}} P_p I_p \tag{3.1}$$

where P_p is the probability of occurrence of a software failure p, and I is the resulting impact of that problem, expressed in dollars of damage or profit loss. Note that implicitly we are also defining the risk connected to each specific problem p, that is, $P_p I_p$. The summation of Equation (3.1) ranges over the entire set of problems the software may have. In general, not all of these problems are known or can be forecasted at development time.

Of course, by using Equation (3.1) we are making the additional assumption that software failures are independent from each other. Although this is in general not the case, and often conditional probabilities must be used for correctly quantifying total risk, Equation (3.1) is good enough for our present purposes, that is, for illustrating the risks related to software delivery. Curtailing the test on a problem or forgoing them altogether means that the risk associated to that problem, not covered by the testing, will remain when the software is used. Risk analysis consists in the identification of software problems. It can be used for:

[5] Of course, in many practical situations walkthroughs would not be acceptable as a replacement for white-box execution tests.

- *goal oriented testing*: software failures having different risks will be covered by test procedures of different depths (and cost);
- *prioritized (depth-first) testing*: software areas with a higher risk receive a higher priority in the testing process.
- *comprehensive (width-first) testing*: software testing will be carried out to cover all rest categories at a pre-set minimum depth.

Risk-based testing consists of analyzing the software code and deriving a test plan focusing on software modules most likely to experience a failure that would have the highest impact. Both the impact and probability of each failure must be assessed before risk-based test planning can proceed. The impact I_p of the risk equation depends on the specific nature of the application and can be determined by domain analysis. For instance, failures in mission critical and safety critical modules may have assigned an higher impact than failures in the rest of the code[6]. Impact assessment requires a thorough understanding of failure costs, which is again highly domain-dependent. Estimating the likelihood of failures P_p means determining how likely it is that each component of the overall software product will fail. It has been proven that code that is more complex has a higher incidence of errors or problems [20]. For example, *cyclomatic complexity* is a well-known criterion for ranking the complexity of source code [16]. Therefore, in procedural languages like C the prioritization of modules according to failures probability can be done simply by sorting them by their cyclomatic complexity. Using cyclomatic complexity rankings for estimating failure probabilities, together with impact estimates from domain knowledge, analysts can pinpoint which software modules should get the highest testing effort [7].

A fundamental way towards the reduction of product risks is the finding and removal of errors in a software product. In the remainder of the chapter we will connect the idea of removing errors with the one of certifying software properties.

3.1.1 Types of Software Testing

Modern software products are often developed following processes composed of pre-defined activities.[8] The test process is also composed of several activities: during *test planning*, the test objects (e.g., the software functions to be tested) are identified. In *test case investigation*, the test cases for these test

[6] From a different point of view, software project management assigns highest impact to failures in modules on the project's *critical path*

[7] Of course, different programming languages and paradigms may use different complexity metrics than cyclomatic complexity for estimating failure probabilities.

[8] Such processes do include agile ones. We shall discuss how testing is carried out within community-based development process typical of open source software in Chapter 6.

objects are created and described in detail. Finally, during *test execution*, test scripts are executed to feed the test cases to the software under test.

Most software products have modular architectures, and testing can be done at different levels of granularity [19]:

- *Unit Testing*: tests the implementation of individual coding units.
- *Module Testing*: tests the detailed design of software modules
- *Integration Testing* tests each subsystem's design.
- *System Testing*: tests the entire system's architecture.
- *Acceptance Testing*: tests the extent to which the entire system meets requirements.

We are particularly interested in acceptance tests, because they are often related to non-functional requirements, that is, to properties the software product should possess in order to be acceptable for the customer. In principle, acceptance tests could be written by the end user herself; in practice, they are often co-designed with the customer, so that passing them with flying colors is guaranteed to satisfy both the software supplier and the purchaser that the software has all the desired properties. In a sense, obtaining the customer sign-off of a set of non-functional properties can be seen as equivalent to providing the same customer with a certificate of the same properties, signed by an authority she trusts.

Each of the above types of software testing involves one or more testing techniques, each of which is applied for different purposes, and requires the design and execution of specific test cases. Software testing techniques include [1]:

- *Functionality testing,* to verify the proper functionality of the software under test, including its meeting business requirements, correctly performing algorithms and providing the expected user experience.
- *Forced error testing,* to try extreme test cases (oversized and malformed inputs, erratic user behavior, etc.) in order to break (and fix) the software during testing, preventing customers from doing so in production.[9]
- *Compatibility testing,* to ensure that software is compatible with various operating systems platforms and other software packages.
- *Performance testing,* to see how the software product under test behaves under specific levels of load. Performance testing includes *stress testing* to see how the system performs under extreme conditions, such as a very large number of simultaneous users.
- *Regression testing,* to ensure that code added to address a specific problem did not introduce new problems.
- *Scalability testing,* a form of performance testing which aims to ensure that the software will function well as the number of users and size of databases increase.

[9] This can also be seen as a form of stress testing. See below.

- *Usability and accessibility testing,* to ensure that the software is accessible to all categories of users (including for example visually impaired ones,) and is intuitive to use. This kind of testing is traditionally the bailiwick of human computer interaction [5].
- *Security testing,* to make sure that valuable and sensitive data cannot be accessed inappropriately or compromised under concerted attack.

Of course the above list is not exhaustive: there are several other types of testing, including one very relevant to our purposes: *regulatory-compliance testing*, which verifies whether a software product complies with the regulations of the industry where it will be used (e.g., for the software controlling a manufacturing tool, one may test whether the user interface can be easily operated without taking off protection gloves). These regulations, and the corresponding testing procedure may vary depending on the type of software and application domain.

Since this book focuses on security certification, it is important to highlight the strict relationship we would like to establish between testing and certification. A key concept goes under the acronym *IV&V*, which stands for *Independent Verification and Validation.* Here, the term *independent* means that the tests on a software product are performed by non-developers; indeed, sometimes the IV&V team is recruited within the same project, other times within the same company developing the software. Conceptually, however, the step between this and outsourcing IV&V to an independently accredited lab is a small one.

It is also important to clarify the relation between security certification and security testing, which is less strict than one might expect. Security testing verifies all software features, from the password checking to channel encryption to making sure that information is not disclosed to unauthorized users. It also includes testing for known vulnerabilities, e.g., assessing how a software product responds to hostile attacks like feeding a software with so much input data that an *input buffer overflow* failure is produced, and the machine on which the software is run can be taken over by the attacker. Buffer overflows are by far the most commonly exploited bug on Linux.

Buffer overflows are the result of putting more data into a programs buffer or input device than is defined/allowed for in the program. A simple example of a C program susceptible to buffer overflow is given below:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
char buf[20];
if(argc < 2)
{
```

```
printf("Usage: \%s <echo>\n", argv[0]);
exit(0);
}
strcpy(buf, argv[1]);
printf("You typed: \%s\n", buf);
return 0;
}
```

The program (we will call `vultest`) copies the string `argv`[1] passed to it on the command line into an internal buffer whose address is `buf`. The internal buffer holds at most 20 chars, so if we pass a longer string on the command line we expect something bad to happen. What we get is the following behavior:

```
M-10:~ edamiani\$  ./vultest aaaaaaaaaaaaaaaaaaaaaaa
<echo> aaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

The segmentation fault error message says that the buffer was filled with more data than it can hold. More specifically, `vultest` wrote too much data on the program stack, which was supposed to hold the arguments passed to `strcpy` (i.e., the `buf` array). These data overwrote the return address to the caller code, which is stored in the stack below the local variables; when `strcpy` tried to jump back to the caller using this return address, it found an illegal address in its stead (most likely the ASCII code for `aa`, `0x6565`) and tried to use it as the target for a jump, causing a violation. Suppose now the input string, rather than a sequence of `a` chars, contains the address of some code supplied by the attacker e.g., the one of a `sh` shell. This would give to the attacker the opportunity to seize `vultest`'s execution environment. There is an additional difficulty here: the attacker's code (let's call it `attack`) must be stored at a valid memory address on the system. In other words, the attacker has to know how to store her code in a valid address, obtain the address, and pass it (instead of `aa`) to `vultest` to overflow the buffer. For instance, the attacker can write `attack`, translate it into hexadecimal figures and load these figures into an array; then, he can use a memory copy system call like `memcpy` to copy the content of this array to some valid location in memory[10] keeping the address to pass it to `vultest`.[11]

[10] We will discuss `memcpy` in some detail in the next Chapter

[11] Some pre-processing (e.g., format conversion) is required before the address can be passed to `vultest`.

3.1.2 Automation of Test Activities

All types of software testing described above can be performed manually or automated.

Manual software testing usually consists of having a human tester feed test cases to the software interface, and performing other types of manual interaction, including trying to hack the software. There are some areas for which manual software testing is most appropriate (or, rather, is the only possibility), such as exploratory security testing where testers do not execute a pre-defined script, but rather explore the application and use their experience and intuition to identify vulnerabilities. Manual software testing has two major disadvantages: first of all, it requires a huge effort and remarkable skills on the part of the tester. Secondly, it is not fully repeatable: manual exploratory tests tend to show high variability of results depending on who is performing them.

Automated software testing can be seen as a way of accelerating the software testing process while minimizing the variability of results. Automation is based on test scripts that can be run automatically, repetitively, and through several iterations of the test process. Automated software testing can help to minimize the variability of results, speed up the testing process, increase test coverage, and ultimately provide greater confidence in the quality of the software being tested. Many testing tasks that defied automation in the past have now become candidates for such automation due to advances in technology. For example, generating test cases that satisfy a set of functional requirements was typically a hard problem that required intervention from an experienced test engineer. Today, however, many tools are available, both commercial and open source, that automate this task to varying degrees, e.g., generating Unix shell scripts to feed input files into the program under test and produce a report. We shall mention again these testing platforms in Chapter 6. The interested readers are however referred to [19].

3.1.3 Fault Terminology

The remainder of this section presents three terms that are important in software certification as well as in testing and will be used later in this chapter. Like terms about certification, terms about testing are defined by standard glossaries such as the IEEE Standard Glossary of Software Engineering Terminology, DOD-STD-2167A and MIL-STD-498 from the US Department of Defense. We focus on the types of problems that using a software product may involve. Throughout the book, following [1], we shall adopt the following definitions of software *fault*, *error*, and *failure*.

Definition 3.1 (Software Fault). A static defect in the software.

Definition 3.2 (Software Error). An incorrect internal state that is the manifestation of some fault.

Definition 3.3 (Software Failure). External, incorrect behavior with respect to the requirements or other description of the expected behavior.

In order to clarify the definitions of fault, error, and failure, we shall use a revised version of an example presented in [1]. Namely, let us consider the following C program:

```
int digitcount (char s[])
{
 int digit = 0;
 int i;
 for (i = 1; s[i] != '\0'; i++)
      if(s[i] >= '0' && s[i] <= '9')
          digit++;
 return digit;
}
```

The *software fault* in the above sample function is of course the instruction `for`, where the function starts counting digit characters at index 1 instead of 0, as would be correct for C character arrays.

For example, using the test case values [`a`, `b`, `0`] and [`0`, `7`, `c`] we notice that `digitcount`([`a`, `b`, `0`]) correctly evaluates to 1, while the test case `digitcount`([`0`, `7`, `c`]) incorrectly gives the same result. Note that only the latter test execution of `digitcount` results in a *software failure*, although the faulty instruction is executed the same number of times in both cases. Also, both the execution with failure and the one without it involve a more elusive concept, the one of *software error*. In order to fully understand it, we need to execute our faulty function stepwise, meaning that we consider its state (i.e., the content of the function's local variables).

The state of `digitcount` consists of three memory locations, containing values for the variables `s`, `digit`, `i`. For the first test case execution, the state at the first iteration of the loop is (`s` = [`a`, `b`, `0`], `digit` = `0`, `i` = `1`). This execution state is a software error, because the value of the variable `i` should be zero on the first iteration. However, since the value of variable `digit` is (purely by chance!) correct, this time the error does not cause a failure. In the second test case execution, the error state is (`s` = [`0`, `7`, `c`], `digit` = `0`, `i` = `1`). Here the error propagates to the variable `digit` and causes a failure. Now, it is clear that when a software product contains a fault, not all test cases will ensure that the corresponding error will cause a failure, how it would be desirable in order to reveal and fix the fault itself. In addition, even when a failure does occur, it is may be very difficult to trace it back to the fault which caused it.

3.1.4 Test Coverage

Since the term "certification" has the same root as the term "certainty", one might be tempted to think that black-box tests can provide conclusive evidence that a software product holds (or does not hold) a given property. For instance, can we use black-box testing to prove to a software purchaser's satisfaction that the execution of the software product she is buying will never require more than 1 MByte of user memory?

Unfortunately, this need for conclusive evidence clashes with a theoretical limitation of software testing. Even for our simple `digitcount` function, using 8-bit ASCII character coding and excluding arrays of characters longer than 256 characters, one would require 2^{64} executions to run all possible test cases.

We may understand better how the identification of the "right" test cases is carried out via the notion of *coverage criteria*. In practice, coverage criteria correspond to properties of test cases. The tester tries to select test cases showing the whole range of their properties values, that is, providing the maximum coverage.

Let us briefly examine some types of coverage that are relevant to our purposes. A classic coverage criterion is to execute all `if` alternatives (i.e., cover all decisions) in the program. This criterion is called *branch coverage*. The corresponding property of test cases we are interested in is which `if` selectors (if any) they trigger. To satisfy this criterion, the tester will chose test cases so that each of them causes the execution of one or more (non-overlapping) branches controlled by `if` selectors in the program. Ideally, the tester will obtain a test set which will cause the execution of all the program's branches, achieving *full branch coverage*. An analogous line of reasoning leads us to the notion of *full statement coverage* criterion, that is, a set of test cases which causes the execution of all the program's statements.

Coverage criteria can be related to one another, in terms of a relation called *subsumption* [11]. A coverage criterion C_1 subsumes C_2 if (and only if) every test set that satisfies criterion C_1 also satisfies C_2. In the case of branch and statement coverage, it is easy to see that if a test set covers every branch in a program, then the same test set is guaranteed to have covered all statements as well. In other words, the branch coverage criterion subsumes the statement coverage criterion.[12]

It is important to realize that some coverage criteria cannot be satisfied at all. For instance in the case of the following C function:

```
int digitcount (char s[])
{
 int i;
 int digit = 0;
```

[12] Our intuition may tell us that if one coverage criterion subsumes another, it should reveal more faults. However, this intuition is not supported by any theoretical result [23].

```
  if (digit) i=0;
  for (i = 1; s[i] != '\0'; i++)
       if (s[i] >= '0' && s[i] <= '9')
           digit++;
  return digit;
 }
```

there is no set of test cases ensuring full statement coverage, due to the presence of *dead code*: the statement `i = 0;` can never be reached, regardless of the input.

One may think to find an algorithm to decide whether such a test set exists or not; unfortunately, there can be no algorithm for deciding whether an arbitrary program can get full coverage with respect to an arbitrarily chosen set of coverage criteria, even though some partial solutions (i.e., for special classes of programs and/or criteria) have been proposed (see Chapter 4). In other words, achieving 100% coverage for any set of coverage criteria is impossible in practice, and there is no way to design a test set that will detect all faults.[13].

Something can be done, anyway: coverage criteria can be used either to generate test case values or to validate randomly generated or manually picked ones. Both problems are in general (i.e., when the criteria are arbitrary) undecidable; the latter technique, however, is the one adopted in practice, because the validation problem turns out to be tractable much more often than the generation one.[14] There is however a drawback: validating randomly chosen test cases will allow us to assess the extent to which a given test set provides coverage, but leave us clueless on how to increase it. In terms of commercial automated test tools, a *test case generator* is a tool that automatically creates test case values. A validator tool that takes a test set and performs its coverage analysis with respect to some criterion. Both types of tools are available as commercial and open source products. Some well-known tools include *xUnit* (JUnit, CPPUnit, NUnit and so on), *IBM Rational Functional Tester*, *WinRunner*, *DejaGnu*, *SMARTS*, *QADirector*, *Test Manager*, *T-Plan Professional*, and *Automated Test Designer* (ATD).

3.2 Test-based Security Certification

For a long time, testing in general and security testing in particular have been internal processes of software suppliers. Software purchasers had prac-

[13] The result that finding all failures in a program is undecidable is due to Howden [13]

[14] More precisely, given a criterion checking whether some existing test cases satisfy, it is feasible far more often than it is possible to generate tests for that criterion starting from scratch.

tically no way to obtain an independent appraisal of a software product's security prior to buying it. Often, disclaimers coming with software products would exclude any guarantee, expressed or implied, of any security or dependability property. This situation is now simply unacceptable for organizations purchasing safety and mission-critical systems. Generally speaking, security certification standards have been devised to provide purchasers with some guarantee of the security properties of their software.

Intuitively, the security certification process of a software product should reveal all the problems and faults of the product's security features, which could lead to vulnerability to attacks. In the early days of security certifications software vendors would limit themselves to asserting (and testing) the presence and functionality of security features, often as a part of non-functional requirements elicitation, and left it to the user to make the link between the support of a given security feature and the corresponding security property. For instance, early certificates would state that a given software system supported Access Control Lists (ACL) on data resources, leaving it to the user to make the connection between the ACL mechanism and the specific category of discretionary access control policies she was interested in.

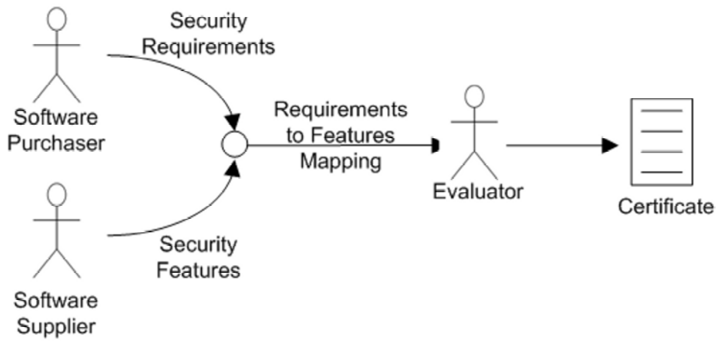

Fig. 3.1: The conceptual model for test-based security certification

The software certification process has greatly evolved along the years. Figure 3.1 shows an abstract conceptual model for today's test-based security certification. The first phase of the security certification process is providing a mapping between the security properties (in terms of security requirements) the software purchaser is interested in and the security features the software vendor has included in the product. Once a mapping has been specified between a set of security properties and the corresponding security features, test-based security certifications provide test-backed proof that: *(i)* the software product under certification actually possesses the required features and *(ii)* such features perform exactly their intended functionalities and nothing else. If the test process is carried out in a controlled environment and by a trusted evaluation body, e.g., as mandated by an internationally accepted *assurance* standard, this proof is usually acceptable enough to the software purchaser.

In the next chapters, we shall use the term “assurance” to refer to all activities necessary to provide enough confidence that a software product will satisfy its users’ security requirements. In other words, security standards specify which security requirements a product should satisfy, while assurance standards specify how to collect and provide the evidence that it does. We shall elaborate further on this issue in Section 3.3. Also, in this chapter we shall use the term “proof” quite loosely: security feature testing (like any other software test) can never be exhaustive, and the mapping between features and properties may or may not have been formalized and proved. We shall come back to these problems when dealing with model-based security certification in Chapter 4.

In the remainder of this chapter, we shall briefly review some early security certification standards adopted in the US, in Canada and in Europe. These early mechanisms are still with us, providing much of the vocabulary shared by software purchasers, vendors and security auditors.

3.2.1 The Trusted Computer System Evaluation Criteria (TCSEC) standard

As one of the biggest software buyers worldwide, the U.S Department of Defense (DoD) has been understandably keen on addressing the issue of standardizing security certifications . The U.S. Department of Defense’s *National Computer Security Center* (NCSC) has sponsored the introduction of what is known as the *Trusted Computer System Evaluation Criteria* (TCSEC) or *Orange Book*. Originally, the Orange Book was devised as a way of standardizing security requirements coming from both the government and the industry [21]; although it was originally written with military systems and applications in mind, its security classifications have been broadly used within the entire computer industry. According to its proposers, the Orange Book was created with the following basic objectives in mind [17].

1. Provide sound security guidelines for manufacturers to be able to build products that satisfy the security requirements of applications.
2. Define a systematic way to (qualitatively) measure the level of trust provided by computer systems for secure, classified and other sensitive information.
3. Allow software purchasers to specify their own security requirements, rather than take or leave fixed sets of security features defined by suppliers.

The classifications in the Orange Book provide a useful shorthand for the basic security features of operating systems.[15]

In the course of time, the NCSC has published different "interpretations" of the Orange Book . These interpretations have clarified the Orange Book requirements with respect to specific families of operating system components. For example, the NCSC's Trusted Network Interpretation of the Orange Book, also known as *Red Book*, is an interpretation of Orange Book security requirements as they apply to the networking components of a secure operating system. The Red Book does not change the Orange book original requirements; it simply indicates how a networking system should operate to meet them. Interpretations come in several colors: in the same way as the Red Book is an interpretation of the Orange Book for network systems, there is a Blue Book interpreting the Orange Book for subsystem components, and other books for other component families. The NCSC has made available a complete set of Orange Book interpretations (the so-called *Rainbow Series*), to assist software vendors in ensuring that their systems comply with Orange Book requirements.

Orange Book security certification goes one step further with respect to the situation where suppliers agreed on non-functional requirements about the presence of some security features. Provisions that need to be present for considering a system to be "secure" are mapped to specific *security requirements*, which must be provided by purchasers. To help software users in defining their own security requirements, the Orange Book introduced four fundamental *requirement types*, which are derived from the objectives stated above [17]: *Policy, Accountability, Assurance* and *Documentation.*

Despite promoting a novel view in supporting software purchasers' own security requirements, the Orange Book ended up with severe limitations in many aspects, which prevented its generalized adoption. These limitations are mainly due to the Book's lack of flexibility in expressing security requirements and in mapping them to security features. The original users of the Orange Book were military and governmental organizations having very specific security requirements, mostly related to preventing disclosure of classified data. This brought families of security requirements which are of paramount importance for business applications (e.g., requirements related to availability and integrity) to be insufficiently considered, and the Orange Book failed to adapt to the security requirements of a wider general market [2].

Today, the Orange Book's main legacy consists in its classification of software products (mainly operating systems) into pre-set categories based on the security requirements they satisfy (and, correspondingly, on the features they possess). TCSEC categories are identified by labels forming an ordinal

[15] While the Orange Book security categories have played an important role in filling the communication gap between vendors, evaluators, and customers, the Orange Book itself [17] is notoriously difficult to read. Also, the Orange Book is not readily available to non-US parties, which has made a full understanding of TCSEC security ratings rather difficult to achieve for security experts outside the US.

scale, allowing for qualitative assessment of security and system comparison. Verifying the actual functionality of security features via suitable tests is left to an external evaulation body, who should be neither the software supplier nor the purchaser. The Orange Book provides some high-level guidelines for testing security requirements, but does not mandate a specific test process or laboratory setting. Also, software products are only listed on the NCSC's Evaluated Products List after a long evaluation process culminating in a Final Evaluation Report from NCSC. It is important to remark that the system submitted to NCSC for certification can include additional modules (e.g., hardware ones) specifically added to comply with TCSEC requirements. This means that users will have to install this additional hardware or software in order to meet the security requirements.

3.2.1.1 TCSEC categories and requirements

The Orange Book security categories range from *D* (Minimal Protection) to *A* (Verified Protection). To be classified in a given category, a software system must provide all the security features corresponding to that category. Namely, categories are defined as follows (see also Table 3.1).

Category	**Class**	**Comment**
D - Minimal Protection	-	Category D includes any system that does not comply with any other category, or has failed to receive a higher classification.
C - Discretionary Protection	C1 - Discretionary Security Protection	Category C applies to *Trusted Computer Bases* (TCBs) with optional object (i.e., file, directory, devices etc.) protection.
	C2 - Controlled Access Protection	
B - Mandatory Protection	B1 - Labelled Security Protection	Category B specifies that the TCB protection systems should be mandatory, not discretionary.
	B2 - Structured Protection	
	B3 - Security Domains	
A - Verified Protection	A1 - Verified Protection	Category A is characterized by the use of formal security verification methods to assure that the mandatory and discretionary security controls are correctly employed.
	A1 and above	

Table 3.1: Orange book categories and classes

- *D - Minimal Protection.* This category includes any system that does not comply with any other category, or has failed to receive a higher classification. D-level certification is very rare.
- *C - Discretionary Protection.* Discretionary protection applies to *Trusted Computer Bases* (TCBs) with optional object (i.e., file, directory, devices and the like) protection.
  - *C1 - Discretionary Security Protection.* This category includes systems whose users are all on the same security level; however, the systems have provisions for Discretionary Access Control by providing separation of users and data, as for example Access Control Lists (ACLs) or User/Group/World protection. An example of C1 requirements is shown in Table 3.2. C1 certification is quite rare and has been used for earlier versions of Unix.
  - *C2 - Controlled Access Protection.* This category has the same features as C1, except for the addition of object protection on a single-user basis, e.g., through an ACL or a Trustee database. C2 is more fine-grained of C1 and makes users individually accountable for their actions. An example of C2 requirements is shown in Table 3.2. C2 is one of the most common certifications. Some of the Operating Systems using C2 certification are: VMS, IBM OS/400, Windows NT, Novell NetWare 4.11, Oracle 7, DG AOS/VS II.

Class	**Requirements**
C1	Username and Password protection and secure authorisations database (ADB)
	Protected operating system and system operations mode
	Periodic integrity checking of TCB
	Tested security features with no obvious bypasses
	Documentation for User Security
	Documentation for Systems Administration Security
	Documentation for Security Testing
	TCB design documentation
C2	Authorisation for access may only be assigned by authorised users
	Object reuse protection (i.e., to avoid reallocation of secure deleted objects)
	Mandatory identification and authorisation procedures for users, e.g., Username/Password
	Full auditing of security events (i.e., date/time, event, user, success/failure, terminal ID)
	Protected system mode of operation
	Added protection for authorisation and audit data
	Documentation as C1 plus information on examining audit information

Table 3.2: C classes' requirements

- *B - Mandatory Protection.* It specifies that the TCB protection systems should be mandatory, not discretionary. A major requirement here is the protection of the integrity of sensitivity labels and their adoption to enforce a set of mandatory access control rules.
  - *B1 - Labelled Security Protection.* B1 systems require all the features required for class C2 and the requirements provided in Table 3.3. Some of the operating systems and environments using B1 certification include: HP-UX BLS, Cray Research Trusted Unicos 8.0, Digital SEVMS, Harris CS/SX, SGI Trusted IRIX, DEC ULTRIX, Trusted Oracle 7.
  - *B2 - Structured Protection.* B2 systems require all the features required for class B1 and the requirements provided in Table 3.3. Some of the systems using B2 certification are: Honeywell Multics, Cryptek VSLAN, Trusted XENIX 4.0.
  - *B3 - Security Domains.* B3 systems require all the features required for class B2 and the requirements provided in Table 3.3. The only B3-certified OS is Getronics/Wang Federal XTS-300.
- *A - Verified Protection.* It is the highest security category. Category A is characterized by the use of formal security verification methods to assure that the mandatory and discretionary access control models correctly protect classified or other sensitive information stored or processed by the system [17]. TCB must meet the security requirements in all aspects of design, development and implementation.
  - *A1 - Verified Protection.* A1 systems require all the features required for class B3 and formal methods and proof of integrity of TCB. The following are the only A1-certified systems: Boeing MLS LAN, Gemini Trusted Network Processor, Honeywell SCOMP. All of them are network components rather than operating systems.
  - *A1 and above.* The Orange Book mentions future provisions for security levels higher than A2, although these have never been formally defined.

3.2.1.2 A closer look

We now take a closer look to the security features corresponding to the Orange Book categories, in order to spell out the main innovation of the TCSEC approach, that is, the mapping between security requirements and security features. We shall focus on C2-level security, which is a requirement of many U.S. government installations.[16] Here, we shall consider four of the most important requirements of TCSEC C2-level security:

[16] In the European Union, government agencies usually refer to ITSEC categories, introduced in the next Section. The corresponding ITSEC rating is E3.

Class	Requirements
B1	Mandatory security and access labeling of all objects, e.g., files, processes, devices and so on
	Label integrity checking (e.g., maintenance of sensitivity labels when data is exported)
	Auditing of labelled objects
	Mandatory access control for all operations
	Ability to specify security level printed on human-readable output (e.g., printers)
	Ability to specify security level on any machine-readable output
	Enhanced auditing
	Enhanced protection of Operating System
	Improved documentation
B2	Notification of security level changes affecting interactive users
	Hierarchical device labels
	Mandatory access over all objects and devices
	Trusted path communications between user and system
	Tracking down of covert storage channels
	Tighter system operations mode into multilevel independent units
	Covert channel analysis
	Improved security testing
	Formal models of TCB
	Version, update and patch analysis and auditing
B3	ACLs additionally based on groups and identifiers
	Trusted path access and authentication
	Automatic security analysis
	TCB models more formal
	Auditing of security auditing events
	Trusted recovery after system down and relevant documentation
	Zero design flaws in TCB, and minimum implementation flaws

Table 3.3: B classes' requirements

- *Discretionary Access Control* (also in C1). The owner of a resource, such as a file, must be able to control access to it.
- *Secure Object Reuse* (C2 specific). The operating system must protect data stored in memory for one process so that it is not randomly reused by other processes. For example, operating systems must swipe per-process memory after use, including kernel-level data structures, so that (user and kernel) per-process data cannot be peeked after the memory has been freed.[17]
- *Identification and Authentication* (C2 specific). Each user must uniquely identify himself or herself when logging onto the system. After login, the system must be able to use this unique identification to keep track the user's activities. In many operating systems, this is achieved by typing a

[17] This requirement applies to the entire memory hierarchy, including disk storage: after a file has been deleted, users must no longer be able to access the file's content. This requires some protection to be applied when the disk space formerly used by a file is re-allocated, e.g., for use by another file.

username and password pair before being allowed access. We shall further explore this requirement in Chapter 5.
- *Auditing* (C2 specific). System administrators must be able to audit the actions of individual users, as well as all security-related events. Access to this audit data must be limited to authorized administrators.

Let us now describe the mapping between these requirements and the security features.

- *Discretionary Access Control.* From a system management perspective, the discretionary access control requirement involves the presence and functionality of a number of security features. For example, a mechanism such as Access Control Lists must be in place for the system administrator to control which users have access rights to which system resources, including files, directories, servers, printers, and applications. Rights must be definable on each resource basis and managed centrally from any single location. A user-group management tool must also be present, through which the system administrator can specify group memberships and other user account parameters.
- *Secure Object Reuse.* This requirement involves the presence and functionality of a number of security features corresponding to the different levels of the memory hierarchy. When a program accesses data, those data are placed in main memory, from where they can be swapped to disk by the virtual memory mechanism. This means that at the level of main memory, two security features are required to satisfy this requirement: *(i)* a *Memory Management Unit (MMU)* level mechanism ensuring the protection of data in the machine's physical memory, so that only authorized programs can access them, and *(ii)* a swap partition protection memory to ensure that no process can access the disk portion hosting the virtual memory used by another process. When these two mechanisms are in place and work correctly, it is impossible for a rogue application to take advantage of another application's data.
- *Identification and Authentication.* A simple password-based log-on procedure may suffice, provided it uses a system-level encryption of passwords so that they are never passed over the wire. This encryption prevents unauthorized discovery of a user's password through eavesdropping.[18]
- *Auditing.* An encrypted log feature must exist supporting logging of all security related events such as user access to files, directories, printers and other resources and log-on attempts. A simple symmetric key encryption mechanism is sufficient to guarantee the secure access to logs mentioned by the requirement.

The mapping described above underlies the entire certification process . When checking a security requirement, the evaluator checks that all the cor-

[18] However, more complex system can be used to satisfy other requirements as well. We shall further explore these mechanisms in Chapter 5.

responding security features are in place and test them (according to some pre-set test guidelines) to verify they are working correctly. If all tests succeed, the product certifiably satisfies the requirements and gets the corresponding certificate. It is important to understand that there are a number of other requirements, such as the usability of the security features, that the TCSEC guidelines do not directly address. For example, the fact that a software product has achieved the C2 certification guarantees that it includes a security feature (e.g., an ACL-based one) capable of controlling which users have access rights to which resources; but such a feature can be extremely awkward to use without a GUI. Again, as far as user accounts and group memberships are concerned, having checked the presence and functionality of a barebones security feature for managing users and groups will not guarantee the possibility of displaying log-in times, account expirations, and other related parameters which will substantially increase the feature usability. These additional requirements are not covered by certifications and therefore remain part of the negotiation between software system vendors and purchasers.

3.2.2 CTCPEC

The Canadian Trusted Computer Product Evaluation Criteria or the CTCPEC was proposed as a revised, "demilitarized" version of the US Orange Book. The CTCPEC goal was to define a wider set of types to accommodate diverse security requirements. The original TCSEC security requirement types were extended to deal with software *integrity, assurance, accountability, confidentiality* and *availability* as well as the original types defined by the Orange Book [2]. In other words, the CTCPEC addressed the commercial market demands by supporting a richer classification of security requirements and more expressive mapping of these requirements to security features. Understandably, and regardless of the efforts to ensure backward compatibility, the wider scope of the CTCPEC caused a growing incompatibility with the Orange Book [2]. This incompatibility, in turn, became a major driver toward a unified, international security certification standard.

3.2.3 ITSEC

At the same time when Canada was trying to define its new security certification standard, on the other side of the Atlantic several countries like France, Germany, UK and the Netherlands started working together to develop a certification designed to satisfy the security needs of the European industry. Learning from the US experience, the Information Technology Security Evaluation Criteria (ITSEC) authors tried also to define a set of goals to overcome the limitations of the Orange Book as well as defining new goals that fit in the European context. Version 1.2 of the ITSEC standard was released in 1991 and is still used today. The major goals of the ITSEC as described in [15] are:

- *Generality.* ITSEC certification criteria are not limited to any category of software products.
- *Interoperability.* ITSEC ensures compatibility with the national catalogs of security evaluation criteria used by each European country, and definition of mappings from these national catalogs to ITSEC.
- *Neutrality.* ITSEC is supported by third parties, taking a neutral role between software.
- *Scalability.* ITSEC contains guidelines for testing security features, aimed at achieving full automatization of the certification process.

3.3 The Common Criteria : A General Model for Test-based Certification

The work done within ITSEC identified two major extension areas to test-based security certification techniques. The first area regarded increasing the expressive power of the security requirements, extending and formalizing their type systems and specifying matching from the requirement to the feature space. The second extension area dealt with automation of feature testing. Both issues were addressed by a standardization group including representatives of the US, Canada and European Union, the latter with the exception of Italy (the group included neither Japan nor Australia). This joint effort, started in 1993 under the label *Common Criteria for IT Security* (CC) became an ISO standard (ISO/IEC 15408) in 1998. The final version of ISO/IEC 15408 was released in 1999. The *Common Criteria* (CC) certification standard has been defined to fulfill the needs of an international standard for affordable software security certification . Common Criteria provides an unified process and a flexible framework to specify, design, and evaluate the security properties of IT products [12].

A major goal of CC evaluation is to certify that the security policies claimed by the developers are correctly enforced by the security functions of the product under evaluation. The Common Criteria model tries to capture all the security aspects of the product and uses the term *Target Of Evaluation* (TOE) for the technological product under evaluation. A TOE can be software, firmware, hardware or a combination of the three [6]; also, it can be a subsystem rather than an entire software system. In this case only the sub-system defined as the TOE will be certified and not the entire product. Figure 3.2 depicts the general model of the CC evaluation.

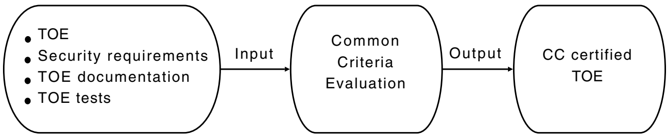

Fig. 3.2: The General Model of the CC evaluation process

In the following of this section, we start by surveying the CC terminology and rationale; in Chapter 6, we will present a detailed case study.

3.3.1 CC components

The CC defines a set of components and specifies the way they interact with each other. CC components can be divided into three categories.

- Catalogs of reusable security functional and assurance requirements.
- Evaluation Assurance Levels (EALs) , specifying the assurance level used in the certification process (from 1 to 7).
- The Protection Profile (PP) and specification of Security Target (ST) , describing respectively the security requirements and the security features of the product to be certified.

It is important to remark that CC Protection Profiles and specifications of Security Targets are themselves *software artifacts*, potentially to be published and exchanged among suppliers, purchasers and independent third parties such as evaluators.

3.3.1.1 The Protection Profile

The introduction of the *Protection Profile* (PP) is an important innovation of the CC, inasmuch it allows groups or consortia of software purchasers to define and share their own sets of security requirements. Of course, PPs do not mandate how (i.e., via which features) these requirements must be implemented; rather, they contain high-level descriptions of users' needs. Also, PPs are not written in a formal or controlled language; indeed, when comparing the PP structure defined in CC part 1 [6] to the publicly available instances of PPs, one may notice that even their structure changes slightly. To get a better idea of what a real PP looks like, we shall use examples taken from real-world PPs, considering their structure rather than the standard one defined by [6]. The general structure of a PP contains the sections showed in Figure 3.3 and discussed below.

1. PP introduction
    - The PP identification
    - PP overview
    - Conventions and document organization
    - Terms and keywords
    - EAL
2. TOE description
3. Security problem definition
    - Threats
    - Organizational security policies
    - Assumptions
4. Security objectives
    - Security objectives of the TOE
    - Security objectives of the TOE environment
    - Security objectives rationale
5. Security requirements
    - Security functional requirements
    - Security Assurance requirements
    - Security requirements rationale
6. PP rationale

Fig. 3.3: PP general structure

The *PP introduction* section provides general information about the PP, allowing it to be registered through the Protection Profile registry, searched and shared. This section includes the *PP identification*, that is, a descriptive information to identify, catalogue, register, and cross-reference a PP. The *PP overview* describes the scope of the PP and provides the necessary information for customers to decide if a particular PP is appropriate for their needs. The two other subsections of the PP introduction section (conventions and document organization, and terms and keywords) help the reader to understand how the PP document is organized and provide basic definitions of any domain specific-term used in the document. Furthermore, the *EAL* for which the PP claims conformance is also mentioned [18]. Figure 3.4 shows the introduction of a well-known PP , the Controlled Access Protection Profile (CAPP) specifying desirable high-level security requirements related to discretionary access control.

The *TOE description* describes the software product from the customer point of view, which includes the purpose of the TOE, the security functionalities needed and the intended operational environment. Other information concerning some technical details could be added such as cryptographic requirements, remote access requirements, and so on. Figure 3.5 shows an ex-

Example of PP Introduction section.

- *PP identification*:
  - *Title*: Controlled Access Protection Profile (CAPP)
  - *Registration*: Information Systems Security Organization (ISSO)
  - *Keywords*: access control, discretionary access control, general-purpose operating system, information protection
- *PP overview*: The Common Criteria (CC) Controlled Access Protection Profile , hereafter called CAPP , specifies a set of security functional and assurance requirements for Information Technology (IT) products. CAPP-conformant products support access controls that are capable of enforcing access limitations on individual users and data objects. CAPP-conformant products also provide an audit capability which records the security-relevant events which occur within the system.
- *Conventions*: This document is organized based on Annex B of Part 1 of the Common Criteria .There are several deviations in the organization of this profile. First, rather than being a separate section, the application notes have been integrated with requirements and indicated as notes. Likewise, the rationale has been
- *Terms*:
  - A user is an individual who attempts to invoke a service offered by the TOE.
  - An authorized user is a user who has been properly identified and authenticated. These users are considered to be legitimate users of the TOE.
  - An authorized administrator is an authorized user who has been granted the authority to manage the TOE. These users are expected to use this authority only in the manner prescribed by the guidance given them.
  - The Discretionary Access Control policy, also referred to as DAC, is the basic policy that a CAPP conformant TOE enforces over users and resources.
- *EAL*

Fig. 3.4: Example of the PP introduction section [4]

ample of a TOE description regarding a general purpose operating system including some *Components-Off-the-Shelf* (COTS) provided by third parties.

The *security problem definition* section describes the expected operational environment of the TOE. More specifically, it defines the known security threats, the security assumptions and the organizational security policies. It is important to notice that it is not mandatory to have statements for all the three subsections. In other words, there may well be cases in which there are no assumptions, or no organizational policies to speak of. However, defining the security threats in a clear and unambiguous way is important, because it makes the construction of the PP easier.

The threats subsection describes the potential threats that may put at risk the TOE assets. In other words this subsection states what we want to protect the TOE from, including violations by system administrators, hackers,

TOE Description. This protection profile specifies requirements for multilevel general-purpose, multi-user, COTS operating systems together with the underlying hardware for use in National Security Systems. Such operating systems are typically employed in a networked office automation environment containing file systems, printing services, network services and data archival services and can host other applications (e.g., mail, databases). This profile does not specify any security characteristics of security-hardened devices (e.g., guards, firewalls) that provide environment protection at network boundaries. When this TOE is used in composition with other products to make up a larger national security system, the boundary protection must provide the appropriate security mechanisms, cryptographic strengths and assurances as approved by NSA to ensure adequate protection for the security and integrity of this TOE and the information it protects.

Fig. 3.5: Example of PP TOE description [4]

unauthorized users, and so forth. Table 3.4 shows some examples of threats definitions suitable for an operating system.

Threat	**Description**
T.ADMIN_ERROR	An administrator may incorrectly install or configure the TOE resulting in ineffective security mechanisms.
T.ADMIN_ROGUE	An authorized administrator's intentions may become malicious resulting in user or TSF data being compromised.
T.SPOOFING	A malicious user, process, or external IT entity may misrepresent itself as the TOE to obtain authentication data.

Table 3.4: Threats definition [4]

Organizational Security Policies (OSPs) are the set of roles, rules and procedures adopted by the organizations using the TOE to protect its assets. OSPs can be defined either by the organization that controls the operational environment of the TOE or by external regulatory bodies [6]. Table 3.5 shows a fragment of an OSP definition specifying the administration roles which will be involved in setting the TOE access control policies, and some separation constraints on them.

During the certification process, some *assumptions* will inevitably have to be made, purely and simply because it is almost impossible to adopt the same set of requirements for all customers. When software purchasers write a PP, they need to take in consideration their specific needs which may change among different groups of customers, even regarding the same product. Focusing on the operating system example, we might well have two groups of customers who define two different PPs for the same operating system. One group may assume that the operating system will include features capable of enforcing access control to classified information, while the other group

OSP	**description**
P.ACCOUNTABILITY	The users of the TOE shall be held accountable for their actions within the TOE.
P.AUTHORIZED_USERS	Only those users who have been authorized to access the information within the TOE may access the TOE.
P.ROLES	The TOE shall provide multiple administrative roles for secure administration of the TOE. These roles shall be separate and distinct from each other.

Table 3.5: PP Organizational security policies from [4]

may assume that access will be regulated by suitable organizational security policies. Table 3.6 illustrates some assumptions [18].

Assumptions	**Description**
A.LOCATE	The processing resources of the TOE will be located within controlled access facilities which will prevent unauthorized physical access.
A.PROTECT	The TOE hardware and software critical to security policy enforcement will be protected from unauthorized physical modification.
A.MANAGE	There will be one or more competent individuals assigned to manage the TOE and the security of the information it contains.

Table 3.6: PP assumptions from [18]

Based on the security problems defined in the previous sections of the PP, the *security objectives* section provides a set of concise statements as responses to those issues. Every problem definition must be adequately addressed by one or more security objectives. Determining the security objectives is a crucial step in PP construction, since it consists the base for defining the testing activities to satisfy those objectives, and because testing without clear objectives may lead to waste of time and effort. The PP includes also a mapping between the security objectives and security problems to help the evaluator to recognize the relations between the different security objectives and their corresponding security problems. Some security objectives which address some of the security problems mentioned above are shown in Table 3.7. Table 3.8, instead, shows the mapping between the security objectives and the corresponding security problems.

The *security objectives rationale* is usually a short description of how the security objectives will address security problems. It can be either written as a separate subsection or embedded in the mapping table. Having it embedded in the table showing the mapping (security problems $\rightarrow$ security objectives) makes it much easier to understand. For instance if we take the mapping

Security objective	**Description**
Objectives to counter Threats	
O.ADMIN_GUIDANCE	The TOE will provide administrators with the necessary information for secure management of the TOE.
O.ADMIN_ROLE	The TOE will provide administrator role to isolate administrative actions
Objectives to enforce OSP	
O.AUDIT_GENERATION	The TOE will provide administrators with the necessary information for secure management of the TOE.
O.ACCESS	The TOE will ensure that users gain only authorized access to it and to resources that it controls.
Objectives to uphold assumptions	
O.PHYSICAL	Those responsible for the TOE must ensure that those parts of the TOE critical to security policy are protected from physical attack which might compromise IT security objectives.
O.INSTALL	Those responsible for the TOE must ensure that the TOE is delivered, installed, managed, and operated in a manner which maintains IT security objectives.

Table 3.7: Examples of Security objectives [18]

Security objective	**Security problem**
Threats	
O.ADMIN_GUIDANCE	T.ADMIN_ERROR
O.ADMIN_ROLE	T.ADMIN_ROGUE
OSP	
O.AUDIT_GENERATION	P.ACCOUNTABILITY
O.ACCESS	P.AUTHORIZED_USERS
Assumptions	
O.PHYSICAL	A.LOCATE
O.INSTALL	A.MANAGE

Table 3.8: Mapping security objectives to security problem definitions [18]

between the threat O.ADMIN_ROLE and the objective T.ADMIN_ROGUE, the standard document [4] defines the rationale as in Figure 3.6.

The *Security Requirements* section represents the core part of the PP document, that is, the one dealing with the desired security properties. To be able to assess or evaluate the security level of a TOE, a PP needs to define a set of requirements that would allow the evaluator to know which software features should be tested. Compared to the other PP sections, the security requirements section is usually much larger. The reason behind that security requirements need to be described clearly, including all the needed details to avoid any ambiguous interpretation. The CC certification defines two types of

TOE Description. It is important to limit the functionality of administrative roles. If the intentions of an individual in an administrative role become malicious, O.ADMIN_ROLE mitigates this threat by isolating the administrative actions within that role and limiting the functions available to that individual. This objective presumes that separate individuals will be assigned separate distinct roles with no overlap of allowed operations among the roles. Separate roles include an authorized administrator and a cryptographic administrator.

Fig. 3.6: Rationale of the mapping between O.ADMIN_ROLE and T.ADMIN_ROGUE [4]

security requirements: *Security Functional Requirements* (SFRs) and *Security Assurance Requirements* (SARs) .

SFRs define the requirements that the security features of the product under evaluation should satisfy. In other words SFRs specify how the TOE should work to preserve its expected behavior. The CC standard includes a predefined extendable catalogue of security functional requirements "that are known and agreed to be of value by the CC part 2 authors" [8]. However, the SFRs are flexible and can be extended for particular scenarios. The CC authors have divided the set of the SFRs into four hierarchies depicted in Figure 3.7 (i.e., *Classes, Families, Components and Elements*), each one providing more fine-grained security requirements [12]. For instance, in our

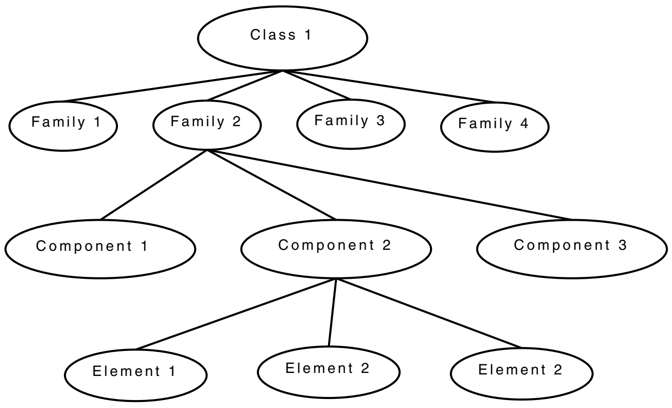

Fig. 3.7: The hierarchical structure of the SFRs

operating system example it is important to limit the functionality of administrative roles. If the intentions of an individual in an administrative role become malicious, O.ADMIN_ROLE mitigates this threat by isolating the administrative actions within that role and limiting the functions available to that individual. This objective presumes that separate individuals will be assigned separate distinct roles with no overlap of allowed operations between the roles. Separate roles may include an authorized administrator and an encryption administrator, as well as provisions for enforcing *division of labor* between the two. Figure 3.8 shows an example of SFRs.

FAU_GEN.2.1 The TSF shall be able to associate each auditable event with the identity of the user that caused the event.

- Application Note: There are some auditable events which may not be associated with a user, such as failed login attempts. It is acceptable that such events do not include a user identity. In the case of failed login attempts it is also acceptable not to record the attempted identity in cases where that attempted identity could be misdirected authentication data; for example when the user may have been out of sync and typed a password in place of a user identifier.
- Rationale: O.AUDITING calls for individual accountability (i.e., "TOE users") whenever security-relevant actions occur. This component requires every auditable event to be associated with an individual user.

Fig. 3.8: An example of SFRs [4]

In the example in Figure 3.8, the name of the SFRs is represented by the standard notation used by CC, namely (`classes`, `families`, `components` and `elements`). The first letter `F` indicates that this is a `Functional requirement`. The two following letters indicate the requirement class (`AU` = `Security audit`); the next three letters represent the family name (`GEN` = `Security audit data generation`); the first digit represents the component number and the second digit indicates the element number. The application note and the rationale provide some details which are specific to this particular PP, to help in interpreting the requirements correctly.

The SARs describe some practical ways to check the effectiveness of the security features of the product under evaluation [22]. The SARs catalogue includes predefined requirements focusing on different phases of the product life cycle such as development, configuration management, testing and so forth. SARs specify the actions deemed necessary to provide enough confidence that the software product will satisfy the security requirements, that is, how to investigate the efficiency of the security functions for the required level of security [8]. Figure 3.9 shows an example of SARs.

Syntactically, SARs follow the same notation standard introduced for SFRs. However, they have an additional letter at the end called the `action element type`. Since the assurance elements are the most fine-grained entities used by the CC , dividing them to even smaller entities may not lead to significant results. For this reason, the standard defines three different action types that identify each of the assurance elements [9].

- *Developer action elements* (letter D): identify the tasks that shall be performed by the developer.

- **ADO_DEL.1.1D** The developer shall document procedures for delivery of the TOE or parts of it to the user.
- **ADO_DEL.1.2D** The developer shall use the delivery procedures.
- **ADO_DEL.1.1C** The delivery documentation shall describe all procedures that are necessary to maintain security when distributing versions of the TOE to a user's site.

Fig. 3.9: An example of SARs [4]

- *Content and presentation of evidence elements* (letter C): describe the required evidence and what the evidence shall show.
- *Evaluator action elements* (letter E): identify the tasks that shall be performed by the evaluator.

Finally the *PP Rationale* is the section where the mapping between security problems and security objectives, and the mapping between SFRs and security objectives are formalized. Also, the PP rationale discusses how threats will be addressed. This is a section where more details could be added to help understand how the TOE shall meet the stated security objectives. Further details can be added concerning SFRs and SARs classes, families, components and elements to help the evaluator to fully understand how the CC components should be applied [9].

3.3.1.2 Security Target

The *Security Target* (ST) is a security specification of a software product. ST contains the security requirements of a given software product, to be achieved by a set of specific security functions. Unlike PPs, STs are implementation dependent, which means that it specifies the implementation details about each SFR. The content of the ST is depicted in Figure 3.10.

The ST is a basis for agreement between the developers, evaluators and, where appropriate, users on the TOE security properties and on the scope of the evaluation. A ST can be derived from a given PP by instantiation; in general, each ST corresponds to a particular PP definition. A ST may then claim conformance to a PP by providing the implementation details concerning the security requirements defined by that PP [14]. Also, ST may augment the requirements derived from the PP. Indeed, there might be cases where there is no PP that matches the security properties of a specific product. In this case, the product developer can still create its own ST without claiming conformance to any PP [14].

An important aspect of ST requirements specification is the definition of the *threats* and *security objectives* of the TOE and its environment. Threats identify situations that could compromise the system assets, while *security*

1. ST introduction
    - ST reference
    - TOE reference
    - TOE overview
    - TOE description
2. Conformance claims
    - CC conformance claim
    - PP claim, Package claim
    - Conformance rationale
3. Security problem definition
    - Threats
    - Organizational security policies
    - Assumptions
4. Security objectives
    - Security objectives for TOE
    - Security objectives for the operational environment
    - Security objectives rationale
5. Extended components definition
    - Extended components definition
6. Security requirements
    - Security functional requirements
    - Security Assurance requirements
    - Security requirements rationale
7. TOE summary specification
    - TOE summary specification

Fig. 3.10: Content of a Security Target

objectives contain all the statements about the intents to counter identified threats and/or satisfy identified organization security policies and assumptions. Based on threats and security objectives, the ST defines the *security requirements* that the TOE security features need to satisfy to achieve the security objectives.

To help establishing an association between the components of the CC and the CC certification process, Figure 3.11 shows where the CC components are used during the CC process. First, software developers need to decide whether their ST will claim conformance to any PP, if yes the specified PP will be included in the certification documents and the ST will be validated against it. Additionally, a set of other documents including design documentation, customer guidance, configuration management and testing must be made

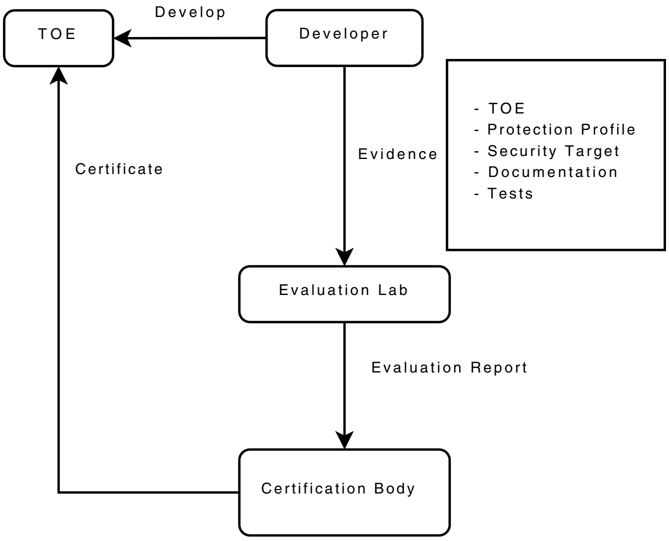

Fig. 3.11: CC process scheme

available to the evaluation body. The role of CC evaluation body is to inspect and analyze all evidence provided by developers.

A fundamental aspect that was taken into consideration when defining the CC is the ability to repeat and reproduce the evaluation results. For this reason, the CC authors have defined an additional document (Common Certification Methodology) that specifies the minimum set of actions to be performed by an evaluator during a CC evaluation [7].

3.3.1.3 Evaluation Assurance Levels

The Common Criteria standard defines seven hierarchical Evaluation Assurance Levels (EAL), which balance the desired level of security and the cost of deploying the corresponding degree of assurance [3]. EAL identifies different sets of security assurance requirements, which are shown in Table 3.9. In case the predefined requirements do not match the level of assurance required, the EAL might be augmented by adding additional assurance requirements.

An example of EALs achieved by operating systems and other software products are given in Figure 3.10.

EAL	Description
EAL1	Functionally tested (black-box testing)
EAL2	Structurally tested
EAL3	Methodologically tested and checked
EAL4	Methodologically designed, tested and reviewed
EAL5	Semiformally designed and tested
EAL6	Semiformally verified design and tested
EAL7	Formally verified design and tested

Table 3.9: Evaluation Assurance Levels

Product	Description
Apple	No evaluations
Linux	EAL 2, for Red Hat Enterprise Linux 3, February 2004
Linux	EAL 3+, for SuSE Linux Enterprise Server V8, Service Pack 3, RC4, January 2004
Solaris	EAL 4, for Solaris 8, April 2003
Solaris	EAL 4, for Trusted Solaris 8, March 2004
Windows	EAL 4+, for Windows 2000 Professional, Server, and Advanced Server with SP3 and Q326886, October 2002

Table 3.10: An example of Evaluation Assurance Levels achieved by software products

3.4 Conclusions

Test-based security certification approaches can provide some confidence (although no certainty) that a software product will preserve its properties of data confidentiality, integrity and availability even under hostile conditions. Many high technology, safety-critical products like aircrafts, include large distributed software systems. In such safety-critical systems, each computational node must be certified to perform its functions safely. As more and more safety-critical and mission-critical software systems communicate with other systems, malicious attempts to subvert those communications multiply, and security concerns become increasingly important. The Common Criteria defines seven Evaluation Assurance Levels (EALs) 1 (low) through 7 (high). The threat level and the value of the information jointly determine the appropriate level of confidence in both the correctness of the security functionality (EAL level) and the extent of the security functionality, specified in a Protection Profile. The consequences of some information being compromised may range from negligible effects to severe damage.

It is important to remark that security certification standards are strictly related to safety ones. The DO-178B standard for safety, like the Common Criteria standard for security, is mostly concerned with program correctness. The difference lies in the fact that DO-178B addresses post-certification quality assurance, while the Common Criteria covers topics such as vulnerability, user documentation and software delivery. DO-178B defines Level A through Level E.; there is no safety impact if Level E software fails, while if Level A software fails, the safety impact is catastrophic. Another characteristic common to both safety and security is that earning certification is much more difficult, risky, and therefore expensive if the certification was not an original design goal. This occurs when certification requirements are extended as the result of revised policies or regulations. Certifications are expensive, although open source tools that analyze source code for faults are becoming available.

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 2008.
2. E. Mate Bacic. The canadian trusted computer product evaluation criteria. In *Proc. of the Sixth Annual Computer Security Applications Conference*, Tucson, AZ, USA, December 1990.
3. K. Caplan and J.L. Sanders. Building an international security standard. *IEEE Educational Activities Department*, 22(3):29–34, March 1999.
4. Information Assurance Directorate. *US Government Protection Profile for Multilevel Operating Systems in Medium Robustness Environments*, 2007.
5. A.J. Dix, J.E. Finlay, G.D. Abowd, and R. Beale. *Human-Computer Interaction.* Prentice Hall, 2004.
6. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model*, 2006.
7. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Evaluation methodology*, 2007.
8. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components*, 2007.
9. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components*, 2007.
10. P.G. Frankl, R.G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transaction on Software Engineering*, 24(8):586–601, August 1998.
11. P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
12. D.S. Herrmann. *Using the Common Criteria for IT security evaluation.* Auerbach Publications, 2002.
13. W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.
14. ISO/IEC. *Guide for the production of Protection Profiles and Security Targets*, 2004.
15. C. Jahl. The information technology security evaluation criteria. In *Proc. of the 13th International Conference on Software Engineering*, Austin, TX, USA, May 1991.
16. Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1(2):308–320, 1976.
17. USA Department of Defense. *DEPARTMENT OF DEFENSE TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA.* USA Department of Defence, 1985.

18. Information Systems Security Organization. *Controlled Access Protection Profile version 1.d*, 1999.
19. M. Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley.
20. S.L. Pfleeger and J.D. Palmer. Software estimation for object oriented systems. In *Fall International Function Point Users Group Conference*, San Antonio, Texas, October 1990.
21. D. Russell and G.T. Gangemi. *Computer Security Basics*. O'REILLY, 1991.
22. K.S. Shankar, O. Kirch, and E. Ratliff. Achieving capp/eal3+ security certification for linux. In *Proc. of the Linux Symposium*, volume 2, page 18, 2004.
23. H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, April 1996.