TS xxx xxx V<0.0.3> (<2013-05>)

Methods for Testing and Specification (MTS);

Security Testing;

Security testing terminology and concepts



**TECHNICAL SPECIFICATION**

Reference

DTS/MTS-101583 SecTest\_Terms

Keywords

Analysis, SECURITY, TESTING

***ETSI***

650 Route des Lucioles

F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C

Association à but non lucratif enregistrée à la

Sous-Préfecture de Grasse (06) N° 7803/88

***Important notice***

Individual copies of the present document can be downloaded from:  
[http://www.etsi.org](http://www.etsi.org/)

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at <http://portal.etsi.org/tb/status/status.asp>

If you find errors in the present document, please send your comment to one of the following services:  
<http://portal.etsi.org/chaircor/ETSI_support.asp>

***Copyright Notification***

No part may be reproduced except as authorized by written permission.  
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute yyyy.

All rights reserved.

**DECT**TM, **PLUGTESTS**TM, **UMTS**TM and the ETSI logo are Trade Marks of ETSI registered for the benefit of its Members.  
**3GPP**TM and **LTE**™ are Trade Marks of ETSI registered for the benefit of its Members and  
of the 3GPP Organizational Partners.  
**GSM**® and the GSM logo are Trade Marks registered and owned by the GSM Association.

Reproduction is only permitted for the purpose of standardization work undertaken within ETSI.  
The copyright and the foregoing restriction extend to reproduction in all media.

Contents

Intellectual Property Rights 5

Foreword 5

1 Scope 5

2 References 5

2.1 Normative references 5

2.2 Informative references 5

3 Definitions, symbols and abbreviations 6

3.1 Definitions 6

3.3 Abbreviations 7

4 Introduction to security testing 7

4.1 Types of dynamic security testing 8

4.2 Penetration testing tools 8

4.3 Test verdicts in security testing 8

6 Security test requirements 9

7 Functional security testing 9

8 Performance testing for security 11

9 Robustness testing and fuzzing 11

9.1 Types of fuzzers 11

10 Security testing activities mapped to SDLC 12

History 15

# Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://ipr.etsi.org).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

# Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee {ETSI Technical Committee|ETSI Project|<other>} <long techbody> (<short techbody>).

# 1 Scope

The present document defines terminology and an ontology which, together, provide the basis for a common understanding of security testing techniques which can be used in testing communication products and systems. The terminology and ontology have been derived from current standards and best practices specified by a broad range of standards organizations and industry bodies and offers guidance to practitioners on testing and assessment of security, robustness and resilience throughout the product and systems development lifecycle. This document specifies terms and methods for the following security testing approaches:

* Verification of security functions
* Load and performance testing
* Resilience and robustness testing (fuzzing)

Target audience: for ETSI MTS and related committees, and the generic testing community.

# 2 References

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at <http://docbox.etsi.org/Reference>.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

## 2.1 Normative references

The following referenced documents are necessary for the application of the present document.

N/A

## 2.2 Informative references

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.] ETSI TR 187 011 (2008): "Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); NGN Security; Application of ISO-15408-2 requirements to ETSI standards - guide, method and application with examples"

[i.] ETSI TS 102 165-1 TISPAN methods and protocols part 1: TVRA

[i.] ISO/IEC 9646-1: 1994: "Information technology -- Open Systems Interconnection -- Conformance testing methodology and framework -- Part 1: General concepts"

[i.] ISO/IEC 15288: 2008: "Systems and software engineering -- System life cycle processes"

[i.] ISO/IEC 15408: 2009: "Information technology -- Security techniques -- Evaluation criteria for IT security -- Part 1: Introduction and general model"

[i.] C.Eckert 2004 Oldenburg-Verlag: IT-Sicherheit, Chapter 4 Security Engineering

[i.7] IEEE Standard Glossary of Software Engineering Terminology, IEEE St. 610.121990

[i.8] Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security. 2001. Espoo. Technical Research Centre of Finland, VTT Publications 447. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.).

[i.9] ISTQB Standard glossary of terms used in Software Testing. Version 2.2 (dd. October 19th, 2012).

[i.10] Takanen, Ari. Fuzzing for Software Security Testing and Quality Assurance. 2008. Artech House. 287 p. ISBN-13: 978-1596932142

[i.11] IETF. Augmented BNF for Syntax Specifications: ABNF. RFC 2234. http://www.ietf.org/rfc/rfc2234.txt

# 3 Definitions, symbols and abbreviations

## 3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

**Attack: A process or script, malicious code or malware that can be launched to trigger a vulnerabilit****y.**

**Zero-day attack: A special form of attack that exploits an unknown vulnerability, and therefore cannot be protected against.**

**Black-box testing:** Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to the selected inputs and execution conditions. [i.7]

**Bit flipping:** A fuzzing technique where input data is mutated by systematically modifying all bits in the communication message or a file.

**Fail closed: T**he software will attempt to shut itself down in case of a vulnerability to prevent further attack attempts.

**Fail open: T**he software will attempt to recover from the failure.

**Fail safe: The software can control the failure and restrict the exploitability of the vulnerability.**

**Failure:** A fault, an indication of a vulnerability.

**False negative: A vulnerability was not detected even if there was one.**

**False positive: A vulnerability was detected, but it is not a real vulnerability.**

**Fuzzing, Fuzz testing: Technique for intelligently and automatically generating and passing into a target system valid and invalid message sequences to see if the system breaks, and if it does, what it is that makes it break.**

**Grammar testing:** An abstract grammar, e.g. ABNF [i.11], serves as the basis for test case generation.

**Input fault injection:** mutates the software or data at interfaces. [i.8]

**Negative testing:** Testing for the absence of (undesired) functionality.

**Risk-based testing:** Testing is prioritized on the likelihood of detecting significant failures.

**Robustness testing:** Testing for robustness of the software system.

**Robustness:** The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. [i.7]

**Syntax testing:** A grammar serves as the basis for testing the syntax of an explicit or implicit language.

**Threat: The possibility of a successful attack.**

**Threat agent: The person or automated software that will realize the threat.**

**Vulnerability: A weakness or a bug in code by design, implementation and configuration mistakes that can be used by malicious people to cause failure in the operation of the software.**

**Known vulnerability: A known weakness in software that has been found in the past.**

**Unknown vulnerability, or a Zero-day vulnerability: a weakness or a bug that is hiding in software waiting for later discovery and exploitation.**

## 3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ABNF Augmented Backus–Naur Form

DAST Dynamic Application Security Testing

DoS Denial of Service

DDoS Distributed Denial of Service

SAST Static Application Security Testing

SDLC System/Software Development Lifecycle

TVRA Threat, Vulnerability and Risk Analysis

# 4 Introduction to security testing

Various security testing techniques are performed at various phases in the product/system lifecycle, starting from requirements definition and analysis and continuing through design, implementation, verification, operations and maintenance. Actors in the security testing activities include developers, internal testers and external security evaluators.

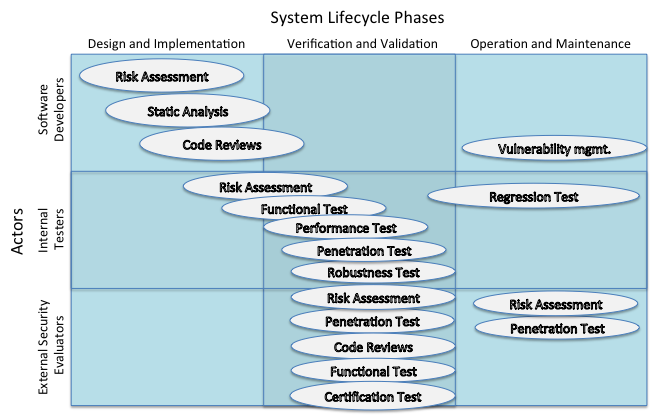


Figure 1: Mapping the security testing techniques to different actors in the SDLC

The assessment of the security of a system is not a single, stand-alone activity but, rather, takes place at a number of differing stages of the System or Software Development Lifecycle (SDLC). Examples of various testing activities are:

1. Internal Assurance (by the Customer and/or Producer):

* Specification Validation
* Unit Test
* Product Test
* System / Acceptance Test

1. External Assurance (review Independent 3rd party):

* Producer Organisation Verification
* Producer Practitioner Verification
* Operating Organisation Verification
* Product / Component Verification
* System Verification
* System Compliance

A model mapping security testing against a generic system lifecycle - as derived from ISO/IEC 15288 can be found in ETSI DEG 201 581.

Security tests using Static Analysis, also called Static Application Security Testing (SAST), analyse the source code or the binary for security weaknesses without executing it. Security tests using Dynamic Analysis, or Dynamic Application Security Testing (DAST), execute the code and analyse the behaviour. SAST tools and techniques are out of scope for this document.

## 4.1 Types of dynamic security testing

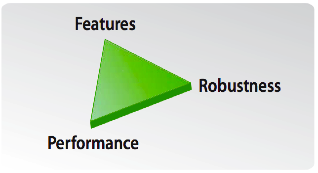


Figure 2: Categories of dynamic application security testing domains [i.10]

Dynamic security testing can be divided in three main domains [i.10], as shown in Figure 1:

* testing for security features and functionality (see clause 7);
* testing for performance, load and stress situations (see clause 8); and
* testing for robustness and reliability (see clause 9).

The purpose of security testing is to determine whether a system meets its specified security objectives and security requirements. The security objectives and requirements should include statements about security functions, performance limitations and software reliability. The security engineering process begins with the specification of security objectives and associated requirements [i.1] and involves iterative Threat, Vulnerability and Risk Analysis (TVRA) [i.2].

Especially in the fields of penetration testing and security auditing, additional tests for attack surface analysis and scanning for known vulnerabilities are used. Selected penetration testing tools are briefly discussed in Section 4.2, but otherwise penetration testing is out of scope for this document.

## 4.2 Penetration testing tools

In penetration testing, the system, device or a software component is analysed using various available hacking tools, with the mentality of a hacker. Some of the available tools are collections of specific exploits or hacker scripts, whereas others are commonly used tools for mapping the attack surface or scanning for common weaknesses in software.

A **vulnerability scanner** is a library of vulnerability fingerprints and friendly attacks in order to reveal known vulnerabilities in the system.

A **port scanner** is a piece of software that will send probes to all UDP and TCP ports in order to trigger responses, mapping the attack vectors by identifying open network services.

**Fuzzing tools**, or **Fuzzers**, send a multitude of generated unexpected and abnormal inputs to a service in order to reveal both known and unknown vulnerabilities.

**Monitoring tools** and **instrumentation tools**, or **instruments**, analyse the network traffic, the executable binary, operating environment or the operating platform, in order to detect failures and abnormal behaviour that could indicate existence of a vulnerability.

**Exploit frameworks**, or **exploitation frameworks** are collections of operational malware scripts and tools that will compromise the system under test.

## 4.3 Test verdicts in security testing

**Observability** of **Failures**/**Faults** is critical in security testing. Detection and identification of different types of failures is required to analyse the root cause. Failure traces, audit traces, and crash traces are critical for analysing the exploitability of failures. Informative log files and debug logs are required for fault identification and repair.

Test verdicts in security testing should be mapped to the following three categories:

* Pass
* Fail
* Inconclusive

**After detection, a failed security test can be further analysed based on the exploitability of the flaw. Exploitability** is often divided by which security target the vulnerability threatens: Confidentiality, Integrity or Availability. For example, a **Denial of Service** exploit will aim to crash a system, or make it unavailable for valid users. A **Distributed Denial of Service** attack will launch a range of requests to the system from a distributed source, making the system unavailable under heavy load. **Buffer Overflow Exploit** and other memory handling bugs alter the internal system behaviour by overwriting memory areas. In worst case, this will result in the target system executing the input data. **SQL Injection Exploit** and other **Execution Exploits** will inject parameters to executed commands. **Directory Traversal Exploit** and other file handling attacks will modify file names and directory names to access data that was not intended to be accessible to the attacker. Other availability issues include **Busy Loops**, **Memory Leaks** and other resource limitations.

The opposite of an observable system is a f**ault tolerant system, which** attempts to hide or survive failures, making detection of vulnerabilities extremely hard, but not impossible. Good instrumentation and exception monitoring is required to detect faults and failures that are handled by the fault tolerant code. For security testing, a debug build of the system might be required, although tests should still be also run in the release build mode.

# 5 Security test requirements

Security test requirements are drawn from:

* Hazard/Threat Analysis
* Vulnerability Analysis
* Risk Analysis
* Control Selection

Hazard analysis and threat analysis should be performed early in the SDLC, but should also be revised during the lifecycle. Vulnerability analysis on the other hand should be performed against an operational system. Control selection affects security test requirements, as it is the starting point for selecting security functionalities in the system. Risk Assessment is the process of analysing potential threats to a system in order to calculate the likelihood of their occurrence [i.4]. The analysis involves the evaluation of the effort required to mount an attack and the gain an attacker might expect from executing the threat successfully.

ETSI TS 102 165‑1 [i.2] describes a method for carrying out Threat, Risk and Vulnerability Analysis (TVRA) within a standardization environment. This is a 10-step process involving both subjective and numerical analysis to determine the risk factor associated with each identified threat.

# 6 Functional security testing

Generically, functional testing considers the system from the end user's perspective. It comprises both interoperability and conformance testing. Functional security testing adopts the same approach but, in addition to benign, legitimate users, also considers possible attackers such as those attempting to consume benefits from the system without registering.

*Functional security testing* bases on the functional requirements related to security. It addresses both positive and negative tests. The list of terms and concepts given in this section has been established for traditional functional testing but are also suitable for functional security testing:

Functional testing is based on an analysis of the specification of the functionality of a component or a system [i.9] without knowledge of the internal structure (black-box testing), depending, for example, on:

* scope of testing:
  + components; or
  + full system;
* context of testing:
  + integration testing;
  + conformance testing;
  + interoperability (IOP) testing; or
  + testing during the System Evaluation []

Many of the details for the functional security testing *process* can be derived and reused from their definitions given in the conformance test methodology and framework (CTMF) as specified in ISO/IEC 9646‑1 [i.3]. It clearly defines a distinction between *abstract* (specification) and *executable* (program/script) test suites. Furthermore, it specifies that a single test *objective* should be implemented by a single separated test *case* and the full list of test cases forms the test *suite*. These mappings may be different in other standards and practices that may combine multiple test objectives in a single test case. Following the CTMF framework a test specification basically comprises the following elements [i.3]:

* a test *architecture* or configuration that describes the setting of the target system under test (components) in contrast to the environment including the test system (components) and e.g. communication utilities (middleware or network);
* test scenarios including *behaviour* and *data* defining a (conditional) sequence of statements or actions; and
* expectations, regarding test outcome or results that have been marked by test *verdicts*.

The test *development* procedure bases on the set of all the requirements of the system/service under test that are to be used for the synthesis of the definition of test purposes. Test purposes include a description of test objectives in an informal or formal language. As a result of this test design procedure the so called test *model* is created. It may include test selection criteria considering special prerequisites/configurations and (conditional) ordering for test execution. Test development is followed by the *implementation* and *execution* of the test, i.e. the interaction of the target Implementation under Test (IUT) and the test system. This step may require dedicated test tools/harness and/or a specific test bed configuration. Any parameterization within the test model requires concrete value settings which should be provided using e.g. Implementation Conformance Statement (ICS) and/or Implementation eXtra Information for Testing (IXIT). Test generation may also consider on the fly observations from some previous test execution and are called online tests.

Functional security testing in the context of System evaluation and certification as defined by the Common Criteria (CC) [i.5] has been described in a similar way but often using different terminology. It focus on the *Target of Evaluation* (TOE) and its *Security Functional Interfaces* (TSFI) that have been identified as enforcing or supporting *Security Functional Requirements* (SFRs) identified and stated for the TOE (cp. Figure 2).



Figure 3: Functional Security Testing in the evaluation process

The CC includes fewer guides about the derivation of the test specification (model) and put emphasis of the test documentation that consist of a test *plan* (a detailed overview of the tests and its configuration) including expected and observed test results. The test plans shall identify the tests to be performed and describe the scenarios for performing each test. These scenarios shall include any ordering depending on the results of other tests. Further requirements for the test plan (or procedure) may be given in national application notes. It could be an informal description of the tests, but also a description that uses pseudo code, flow diagram, but also concrete reference to e.g. test programs/vectors. Further details in the context of network testing are provided in [i.1].

# 7 Performance testing for security

One of the most common and easiest ways to deploy attacks against systems is a Distributed Denial of Service (DDoS) attack. In this attack, messages or message sequences are sent to the target system in order to restrict or limit valid access to the system. The target system will not receive important messages or information due to performance limitations in the application, limited network bandwidth, resource limitations of the platform operating system, or physical resource limits of the used hardware. In worst case, the entire system can crash under overwhelming load, which is often the goal of the hacker.

In traditional load or performance tests, the system is stressed just slightly above the load that is expected in real deployment. In security tests, however, the system is pushed to its limits by fast sequential or parallel load (Figure 4). Each parallel session can bind resources, and each sequential session can push the processing power to the limits. Both test scenarios are typically required to measure the performance limits, and to demonstrate what happens when those limits are reached.

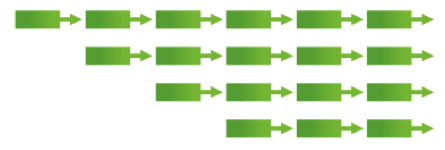


Figure 4: Simplified visualization of sequential and parallel sessions.

The actual threat scenarios related to load can be much more complex than simple repetition of valid sessions, such as half-open sessions where a session is opened but never closed resulting in consumption of target resources.

If countermeasures for DDoS are applied, then the load and performance tests should be written also as functional tests against the relevant countermeasures.

# 8 Robustness testing and fuzzing

Robustness testing, often referred to as "Fuzzing", is a form of testing where system inputs are randomly mutated or systematically modified in order to find security-related failures such as crashes, busy-loops or memory leaks. Hackers use these flaws in order to inject malicious code into the system, compromising the integrity of the system.

In some areas, fuzzing is also used to find reliability and robustness errors caused by corrupted packets or interoperability mistakes. Robustness testing is a more generic name for fuzzing, as the name "fuzz" typically refers to random white noise anomalies.

Fuzzing tests a live executable system for unknown vulnerabilities. Fuzzing is a form of dynamic risk-based system evaluation and should be used as part of the post-development TVRA activities. It is not a conformance activity although it can be used as part of testing the error handling conformity. There is no expected response to a test input, and therefore conformance oracles are very difficult to build for fuzz testing. Fuzzing is typically performed as black-box testing through the exposed interfaces but there are gray-box variants of fuzzing where the code execution is instrumented and fuzz test generation changes based on actions inside the tested binary.

"Smart Fuzzing" is typically based on a behavioural model of the interface being tested. Such testing needs to be protocol aware and have optimized anomaly generation. When fuzz tests are generated from a model built from the specifications, the tests and expected results can also be documented automatically. Protocol awareness increases test efficiency and coverage by going deep into the behaviour in order to test areas of the interfaces that rarely appear within typical use cases. Smart fuzzing is dynamic in behaviour with the model implementing the required functionality for exploring deeper in the message sequence. The creation of anomalies can be optimized and can go beyond simple boundary value analysis. Smart model-based fuzzers explore a much wider range of attacks by testing with data, structure and sequence anomalies. Libraries of anomalies are typically built by inspecting the system or design to determine what and where potential errors might occur, selecting known hostile data and then systematically trying it in all areas of the interface specification.

"Dumb Fuzzing" is typically template based, building a simple structural model of the communication from network activity capture or files. In its simplest form, a template-based fuzzer will use the template sample as a binary block of data, which it mutates. Depending on the algorithm used, template-based fuzzing can appear similar to random white noise ad-hoc testing. Random test generators include everything from simple bit-flipping routines to more complex mutation algorithms such as moving input data around, removing data, or replacing data with other unexpected data.

Test generation can be either on-line or off-line. Online test generation has the benefit of adapting to the behaviour and feature set of the test target. Offline tests can sometimes save time from the test execution, but can take significant amount of disk space. Offline tests will also require regeneration in case the interface changes, and therefore maintenance of the tests consumes a lot of time.

Fuzzer types (note that a fuzzing tool can feature several of these techniques or classifications):

* Specification-based fuzzer is a tool that is always also model-based, and where the behavioural model is build from the interface/protocol specification.
* Model-based fuzzer is a test generator that uses a behavioural model internally in order to generate and execute the fuzz tests. The model can be interpreted from an abstract test notation, formal specification, or from a template (traffic capture or a file).
* Block-based fuzzer is a simple model where the structure of a message is described as data blocks, with meta data to help test generation.
* Random fuzzer applies random mutations in random places in the data.
* Mutation fuzzer applies random or non-random mutations into the data. It can be either model-based or template based fuzzer.
* Evolutionary/Learning fuzzer applies changes to the data based on replies from the target system, or based on information provided by other monitoring or instrumentation tools such as branch coverage information.
* File fuzzer is a fuzzer that tests file formats such as videos, documents, pictures or audio.
* Protocol fuzzer is a fuzzer that tests communication protocols such as SIP, RTP, HTTP, or DHCP.
* Client-side fuzzer tests the client side implementation such as browser or VoIP terminal.
* Server-side fuzzer tests the server-side implementation such as a web server or VoIP proxy/registrar.

# History

|  |  |  |
| --- | --- | --- |
| **Document history** | | |
| <Version> | <Date> | <Milestone> |
| 0.0.1 | September 2012 | First draft |
| 0.0.2 | January 2013 | Stable draft |
| 0.0.3 | May 2013 | Revised stable draft (based on review comments) |
|  |  |  |