ETSI TR 101 583 V1.1.1 (2015-03)

Methods for Testing and Specification (MTS);
Security Testing;
Basic Terminology

Reference

DTR/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

Contents

Intellectual Property Rights

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

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 latest research, but also current standards and best practices specified by a broad range of standards organizations and industry bodies. The present document aims to provide information to practitioners on techniques used in testing, and assessment of security, robustness and resilience throughout the product and systems development lifecycle. The present document lists terms and methods for the following security testing approaches:

- Verification of security functions and risk-based testing.
- Load, stress and performance testing.
- Resilience and robustness testing (fuzzing).
- Penetration testing.

Static Application Security Testing (SAST) tools and techniques are out of scope for the present document.

2 References

2.1 Normative 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 reference 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.

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

Not applicable.

2.2 Informative 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 reference document (including any amendments) applies.

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

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.1] ETSI TS 102 165-1: "Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); Methods and protocols; Part 1: Method and proforma for Threat, Risk, Vulnerability Analysis".

[i.2] IEEE St. 610.12-1990: "IEEE Standard Glossary of Software Engineering Terminology".

[i.3] ISO/IEC 9646-1:1994: "Information technology -- Open Systems Interconnection -- Conformance testing methodology and framework -- Part 1: General concepts".

[i.4] ISO/IEC 15408:2009: "Information technology -- Security techniques -- Evaluation criteria for IT security -- Part 1: Introduction and general model".

[i.5] VTT Publications 447: "A Functional Method for Assessing Protocol Implementation Security", 2001, Espoo. Technical Research Centre of Finland,. Kaksonen, Rauli.128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.).

[i.6] "Fuzzing for Software Security Testing and Quality Assurance", 2008. Takanen, Ari, Artech House. 287 p. ISBN-13: 978-1596932142.

[i.7] ETSI TR 101 590 (V1.1.1): "IMS Network Testing (INT); IMS/NGN Security Testing and Robustness Benchmark".

[i.8] ETSI TR 101 577 (V1.1.1): "Methods for Testing and Specifications (MTS); Performance Testing of Distributed Systems; Concepts and Terminology".

[i.9] Recommendation ITU-T X.1524: "Common weakness enumeration".

3 Definitions and abbreviations

3.1 Definitions

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

asset: anything that has value to stakeholders, its business operation and its continuity (ETSI TS 102 165-1 [i.1])

attack: technique, process or script, malicious code or malware that can be launched to exploit a vulnerability or to bypass security controls on the system

attack surface: consists of user interfaces, target protocol interfaces and reachable data paths that can be attacked within the system

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 (IEEE St. 610.12-1990 [i.2])

bottleneck: severe limitation of the throughput capacity of a system service due to a single cause (ETSI TR 101 577 [i.8])

consequence: outcome of an event affecting objectives, in security testing, the impact from the resulting failure to the protected assets after a successful attack or security test case

constant load: load pattern where the SUT is exposed to a fixed rate of service requests per time unit. Constant load is commonly used in performance tests of stability and availability characteristics (ETSI TR 101 577 [i.8])

exploit: security jargon for automated attacks

fail closed: software will attempt to shut itself down in case of an undesired failure to prevent further corruption by attacks

fail open: software will attempt to recover from the failure and maintain service

fail safe: software will attempt to control the failure and restrict the exploitability of the vulnerability

failure: result of a fault, in security testing, an indication of a vulnerability

false negative: in security testing, a vulnerability was not detected even if one existed

false positive: in security testing, a vulnerability was indicated, even if it did not exist or was not possible to exploit

fuzzing, Fuzz testing: negative testing technique for automatically generating and injecting into a target system anomalous invalid message sequences, broken data structures or invalid data, in order to find the inputs that result in failures or degradation of service

known vulnerability: vulnerability in a specific version of software that has been found in the past

likelihood: chance of something happening

load testing: load testing uses large volumes of valid protocol traffic to ensure that a system is able to handle a predefined amount of traffic (ETSI TR 101 590 [i.7])

model-based security testing: approach of automatically generating security tests from behavioural models

negative testing: testing for the absence of undesired functionality

off-line testing: automated test generation technique where series of tests are generated and stored for later execution, typically as a test script or set of individual tests

on-line testing: automated test generation and execution technique where test is generated and executed at the same time, possibly with capability to adjust the functionality of the subsequent test based on earlier test or the current test sequence

performance testing: uses large volumes of valid traffic to find the limits of how much traffic a system is able to handle

penetration testing: practical, proactive and authorized use of social attacks, environment attacks, load attacks, automated input validation attacks, data attacks, logic attacks and other relevant attacks to test for vulnerabilities in a system, and to verify the consequence of successful attacks to the assets protected by the system

NOTE: In formal audits, penetration testing should be performed by professionals and experienced penetration testers.

response time: elapsed time from receiving a service request to the beginning of sending the response to the request (ETSI TR 101 577 [i.8])

risk: combination of the consequences of an event with respect to an objective and the associated likelihood of occurrence

risk-based security testing: integration of security risk assessment results in the testing process aiming for systematic guidance, prioritization and optimization of security testing activities

robustness: degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions (IEEE St. 610.12-1990 [i.2])

robustness testing: sends large volumes of invalid, malformed or otherwise unexpected traffic to the SUT in order to make it fail (ETSI TR 101 590 [i.7])

security requirement: statements about security functions, performance limitations and software reliability for a piece of software, sub-component or system

security test case: set preconditions, inputs (including actions, where applicable), and expected results, developed to determine whether the security features of the SUT have been implemented correctly or to determine whether or not the covered part of the SUT has vulnerabilities that may harm the availability, confidentiality and integrity of the SUT

susceptibility testing: informal penetration test or other type of security review, not necessarily conducted by professionals or experienced penetration testers

system Under Test (SUT): set of hardware and software components constituting the tested object

test-based risk assessment: risk assessment approach that modifies the analysis based on results of security tests

threat: potential for violation of security, which exists when there is a circumstance, capability, action, or event that could cause harm

unknown vulnerability, Zero-day vulnerability: vulnerability that is hidden in software waiting for later discovery and potential exploitation, and which are unknown to the software developer and/or the public

vulnerability: any weakness in software that can be used to cause a failure in the operation of the software

weakness: shortcoming or imperfection in the software code, design, architecture, or deployment that, could, at some point become a vulnerability, or contribute to the introduction of other vulnerabilities (Recommendation ITU-T X.1524 [i.9])

zero-day attack: special form of attack that exploits an unknown vulnerability, and therefore cannot be protected against

3.2 Abbreviations

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

CC	Common Criteria
CTMF	Conformance Test Methodology and Framework
DAST	Dynamic Application Security Testing
DDoS	Distributed Denial of Service
DoS	Denial of Service
MBST	Model-Based Security Testing
MBT	Model-Based Testing
SAST	Static Application Security Testing
SDLC	System/Software Development Lifecycle
SUT	System Under Test
TOE	Target Of Evaluation
TSFI	TOE Security Functional Interface
TVRA	Threat, Vulnerability and Risk Analysis

4 Introduction to security testing

In software engineering terms, security can be seen as an umbrella activity. The assessment of the security of a system is not a single, stand-alone activity but, rather, takes place at a number of different stages of the System or Software Development Lifecycle (SDLC).

The purpose of security testing is to find weaknesses in software implementation, configuration or deployment. These weaknesses can potentially create or become vulnerabilities in the system. Various security testing techniques are applied at various phases in the product/system lifecycle, starting from requirements definition and analysis and continuing through design, implementation, verification, operations and maintenance (figure 1).

Security tests can be performed in two complementary approaches. 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. Problem with SAST tools is the number of **false positives**, indications of security flaws that cannot be triggered to cause security **failures**. Security tests using Dynamic Analysis, or Dynamic Application Security Testing (DAST), execute the code and analyse the behaviour. DAST tools can have **false negatives** caused by bad test design, missing tests that do not trigger the security failures due to bad test coverage or bad choice of tools. In the present document, our focus is in dynamic tests and our use of the term security testing refers to dynamic security testing only.

The actors in the security testing activities include developers, internal testers and external security evaluators. Our focus in this article is in the activities related to internal testing typically performed by internal testers during Verification and Validation (V&V). These activities include:

- Risk Assessment and Risk-based Security Testing (clause 5).
- Functional Testing of Security Features (clause 6).
- Performance Testing (clause 7).
- Robustness Testing (clause 8).
- Penetration Testing (clause 9).

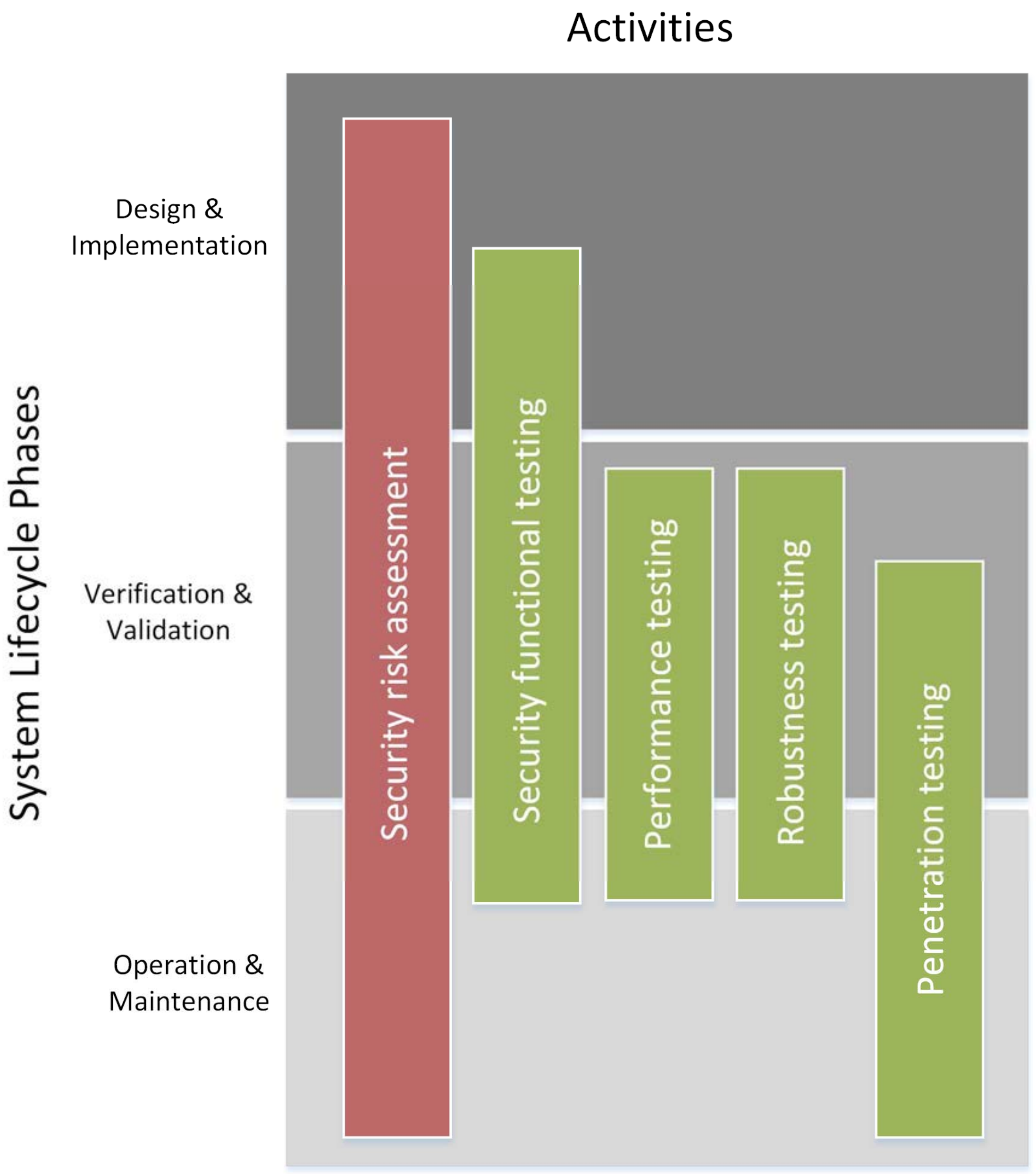

Figure 1: Mapping the security assessment and testing techniques to different phases of the system lifecycle phases

The purpose of security testing is to determine whether a system meets its specified security requirements. The requirements should include statements about security functions, performance limitations and software reliability.

Risk assessment can provide information on potential threats, potential vulnerabilities and the identification of the most critical areas of the system. Moreover the execution of tests can be prioritized based on the risk analysis so that the most relevant test cases are executed with high priority (clause 5).

In penetration testing and third party security audits, additional tests for **attack surface** analysis and scanning for **known vulnerabilities** are used (clause 9).

Several of the testing categories discussed in the present document have other names in different standardization bodies and industry practises. The definition for security testing itself is often limited to only touch the tests focused on security functionality only, and excludes performance and robustness. **Performance testing** can be called stress test or **load test**. **Penetration tests** can be called vulnerability tests or susceptibility tests. **Robustness testing** is often called **fuzzing**.

The observability of failures/faults is of prime importance in security testing, and therefore all fault tolerance features should be disabled during security tests. Detecting and identifying different types of failures is essential in analysing the root causes in order to get the problems fixed. 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. When system is built for reliability, and will try to recover from failure situations (**fail open** or **fail safe**), a debug build of the system that will shut down in case of slightest errors (**fail open**) is usually required for security tests, although the same tests might still be good to run also in the release build mode.

As shown in figure 2, a test execution/run comprises the execution of individual test cases or test procedures that are executed once or repeated millions of times. Sometimes a group of test cases can cover a single security test purpose, or the individual test cases can be considered as part of a larger test procedures that covers the security requirements. Assigning test verdicts can apply to individual test cases, groups of test cases, or test procedures.

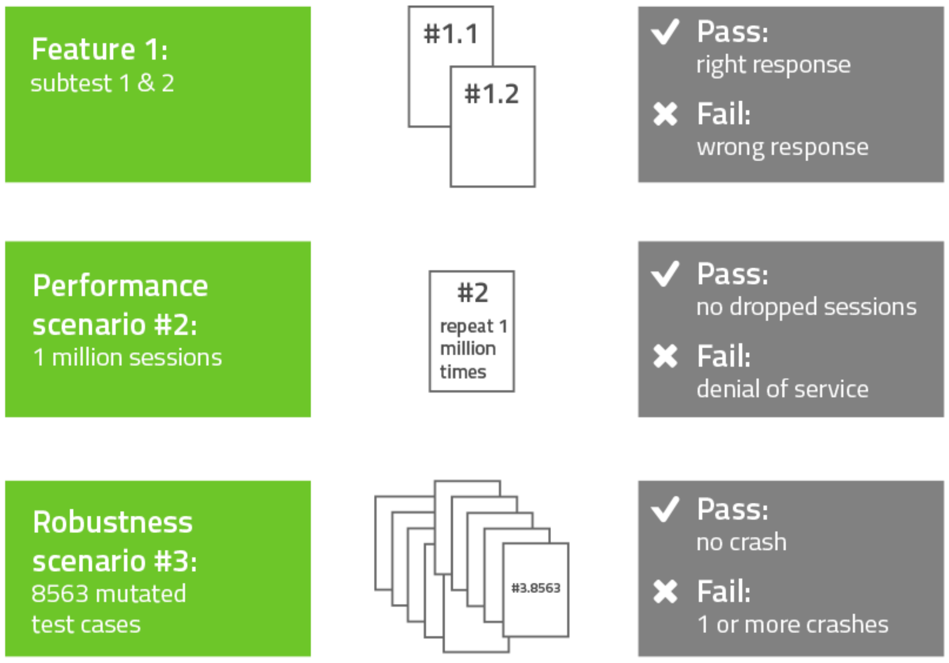

Figure 2: Various types of security tests and test verdicts

Test verdicts in security testing can be assigned to the following three categories:

- Pass.
- Fail.
- Inconclusive.

Depending on the testing approach, these verdicts have different meanings. A "fail" verdict is clear: the test case or test procedure exposes a potential compromise of one or several of the security requirements. This means either a crash (availability), data modification (integrity), or unauthorized data read (confidentiality). On the other hand, a "pass" verdict almost never means that a method of compromising the security does not exist, as the quality of the test cases and test procedures is based on the selected test coverage. In **negative testing**, there can always be a test case that would trigger the vulnerability, but that was out of the scope of the selected test coverage, resulting in a **false positive** test result. Inconclusive means that the analysis was incomplete, e.g. in DoS tests it usually means a crash was found but was impossible or difficult to reproduce.

After detection, a failed security test can be further analysed based on the exploitability of the flaw. Exploitability is often categorized by which security target or requirement the vulnerability threatens: Confidentiality, Integrity or Availability. Some examples of exploitable failure situations (**attacks**) include:

- Denial of Service attack will aim to halt or break a system, or make it unavailable for valid users. Other availability issues include Busy Loops, Memory Leaks and other resource limitations.
- 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 attack 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.
- Code Injection and other Execution attacks will inject parameters to executed commands such as database queries.
- Directory Traversal attacks 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.

5 Risk Assessment and Risk-based Security Testing

The security engineering process begins with the specification of security requirements. It typically involves an iterative security risk assessment that analyses the potential threats to a system in order to calculate the likelihood of their occurrence (IEEE St. 610.12-1990 [i.2]). The risk assessment comprises the identification of assets, threats and vulnerabilities as well as the identification and propagation of risk treatments (i.e. security controls and other counter measures). Risk itself is considered a metric that indicates the combination of the consequences of an unwanted incident with respect to an asset and the associated likelihood or estimated frequency of occurrence.

There are several risk assessment methodologies available that provide dedicated guidance on how to identify the sources of risks, their causes and their potential consequences within different contexts and with different strategies. ETSI has published the Threat, Vulnerability and Risk Analysis (TVRA) method (ETSI TS 102 165-1 [i.1]) to support and rationalize security standardization, and to support and rationalize system design decisions.

While functional testing, performance testing, robustness testing and penetration testing all address different test objectives, risk assessment lays the ground for the other three activities in such a way that it systematically helps to optimize the test design process and to focus the security testing activities to the most critical areas. Risk-based security testing is useful when a complex system requires numerous tests for adequate coverage in limited time. It addresses the problem that, although in theory it is indeed desirable that a system is tested as extensively as possible, in practice there are time and budget constraints that make a systematic selection of tests necessary. Risk-based security testing provides guidance for optimization on different level:

- Risk-based security test planning deals with the integration of security risk assessment in the test planning process. For that, security risk assessment is used to roughly identify high-risk areas or features of the **system under test** (SUT) and thus determine and optimize the respective test effort that is needed to verify the related security functionality or to address the related vulnerabilities.
- Risk-based security test design and implementation deals with the integration of security risk assessment in the test design and implementation process. Risk assessment is used to systematically determine and identify test conditions (testable aspects of a system), test purposes or high-level test scenarios that are dedicated to address the identified threats and vulnerabilities. Combined with attack surface analysis, risk analysis can focus on identifying the most likely functions and features that can be used to launch attacks against the system, and to identify areas of interest, up the actual lines of code, that need special attention and a more detailed assessment through testing.
- Risk-based test analysis and summary aims for improving the evaluation of the test progress by introducing the notion of risk coverage and remaining risks on basis of the intermediate test results as well as on basis of the errors, vulnerabilities or flaws that have been found so far. This process is meant to support the test management process with risk related information that can be used to depict the test results in terms of their relation to the overall security risks.

Furthermore, risk-based testing approaches can help to optimize the risk assessment itself. The feedback loop from test results to the risk analysis is called **test-based risk assessment** and helps in refining the risk models and related probabilities. Particularly relevant in this setting is security testing that uses automated testing tools such as vulnerability scanners, fuzzing tools or network discovery tools. Knowledge on vulnerabilities and their location in the code can additionally help in identifying new threats and their potential locations in the code.

At minimum, risk analysis should be used to prioritize various testing approaches, the required test coverage for each testing approach, and the time allocated for testing. For example, a system that is not performance critical might not require that much performance testing. On the other hand, a system that is directly exposed to Internet might require more thorough robustness testing. An especially slow system that performs client-side functionality such as a mobile application might require special attention to simulated test environments and parallelized testing.

6 Functional Testing of Security Features

Functional testing considers the system from the system functionality and functional requirements perspective. It comprises of unit, integration, product, system, interoperability and conformance testing. Functional security testing adopts the same approach but, in addition to benign, legitimate users, functional security testing also considers the possibility of intentional attacks attempting to use the resources from the system without legitimate right to use it. Functional security tests can address both positive and negative test requirements:

1) What software should do, security functionalities such as providing authentication

2) What software should not do, security requirements such as not storing confidential data in memory

Functional testing is based on analysing the specification of the functionality of a component or a system without knowledge of the internal structure (**black-box testing**). Although security tests can be integrated in all phases of testing, the focus is usually in unit and system tests as opposed to integration, conformance or interoperability. Security features are usually a critical focus area during third-party 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]. Most significant difference is that security requirements are often expressed as negative requirements such as "system should not accept wrong password", and therefore a certain test objective or requirement can require tens or sometimes millions of unique tests to validate the functionality. When test requirements are mostly negative requirements, the testing approach is called **negative testing**. Finally, the process of observations and evaluation regarding test outcome or expected results, can be very different from traditional functional testing as they might require extensive instrumentation of the target system or monitoring of the communications.

Functional security testing in the context of system evaluation and certification has been described in TVRA and Common Criteria (CC) in a similar way but using different terminology. It focuses 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, ETSI TS 102 165-1 [i.1] and ISO/IEC 15408 [i.4].

The TVRA and Common Criteria (CC) include fewer guidelines 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 the expected and observed test results. The purpose of such test plans is to identify the tests to be performed and to describe the scenarios for performing each of the tests, including ordering dependencies to 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 TVRA, ETSI TS 102 165-1 [i.1].

7 Performance Testing

Performance testing, also called stress testing or load testing, aims to verify that the SUT can tolerate required **constant load** of service requests, and that the SUT will perform according to the performance requirements, and will have adequate **response time** for valid requests even while under load-based attacks.

In traditional load or performance tests, the system is stressed just slightly above the load that is expected in real deployment. In security testing, performance testing goes a step further, and tries to find the load or stress level that will result in denial of service. Purpose is to identify the service **bottlenecks**, and to eliminate the reason for them, or to optimize the system so that it will be harder to trigger the cause. 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.

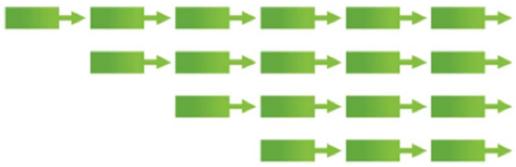

Figure 3: Simplified visualization of sequential and parallel sessions

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.

Distributed Denial of Service (DDoS) attack is an example of an easy to perform and thus common load-based attack. In DDoS attack, messages or message sequences are sent to the target system in order to restrict or limit valid access to the system. The attack exhausts resources either on the target or on the way to the target:

- the target system will not receive legitimate 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 the worst case scenario, the entire system can crash under overwhelming load, which is often the goal of the attacker.

If countermeasures for DDoS are applied, then the load and performance tests should be written also as functional tests against the relevant countermeasures. Attacks based on various fast-paced or parallel load scenarios can be very difficult to block if they originate from distributed bot networks with millions of participating machines and use valid requests or very complex quickly varying mutated requests.

8 Robustness Testing

Robustness testing aims to test that the system can tolerate certain level of attacks, and function correctly after the attack. Often referred to as "Fuzzing", this 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. Attackers use these flaws as stepping-stones in order to inject malicious code into the system, compromising the integrity of the system [i.5], [i.6] and [i.7].

Fuzzing tests a live executable system to uncover **unknown vulnerabilities**. 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 functional black-box testing through the external interfaces such as networked message sequences, file inputs or user inputs.

Fuzzing techniques are typically categorized based on three aspects:

1) how the behaviour of the system is modelled: Model-based fuzzing vs. static or template-based fuzzing;
2) how the tests are altered to test for unexpected messages or structures,
3) and what type of anomalies are used in data elements.

"Smart Fuzzing" or Generational Fuzzing is typically based on a behavioural model of a protocol. 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.

Many different names for smart fuzzers are used. Generation or generational fuzzing refers to the fact that tests are directly generated from a behavioural model. Behavioural fuzzing technique is a term that is sometimes are referred to when the behaviour is mutated. Grammar fuzzing or grammar testing can also sometimes be categorized under smart fuzzing.

"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. Other names for dumb fuzzers include mutation fuzzers and data fuzzers, although majority of smart fuzzers also do mutations and data fuzzing.

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

Fuzzing techniques (note that a fuzzing tool can feature several of these techniques):

- Specification-based fuzzing is always also model-based, and where the behavioural model is built from the interface/protocol specification.
- Model-based fuzzing is 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 fuzzing uses a simple model where the structure of a message is described as data blocks, with meta data to help test generation.

Fuzzers can also be categorized based on the anomalization technique:

- In random fuzzing the tests are generated by applying random mutations in random places in the data.
- Mutation fuzzing applies random or non-random mutations into the data. It can be either model-based or template based fuzzer.

A special form of fuzzing is based on evolutionary test generation where the model and applied mutations to the structures and data are based on replies from the target system, or based on information provided by other monitoring or instrumentation tools such as branch coverage information.

9 Penetration Testing

Penetration testing is very strictly reserved term used in formal security audits, reviews and conformance for assessments conducted by professional accredited security experts. Software testers should use caution to use the term, or use the term susceptibility testing for the security reviews, scans and/or tests. In the present document, the term penetration testing will be used whether or not the tests are conducted by professional experts or by software testers.

In penetration testing, the system, device or a software component is tested using various available hacking tools and methods, with the mentality of an attacker. Some of the available tools are collections of specific exploits or attack scripts (real-life attacks), whereas others are commonly used tools for mapping the attack surface or scanning for common weaknesses in software. Penetration tests will use all above testing practices: functional tests, performance and robustness.

Known vulnerabilities are scanned by trying to trigger vulnerabilities with known attacks, or by checking version information of software from the responses. A **vulnerability scanner** is a tool that contains a library of vulnerability fingerprints and friendly attacks in order to reveal known vulnerabilities in the system. Non-intrusive penetration tests will base its test results on non-hostile checks such as behavioural changes or version information, whereas hostile penetration test will actually trigger the flaw, often resulting in a crash or system compromise through use of harmless malware.

First part of a penetration test is to identify the attack surface. This can be done externally or internally. An internal study will look at which processes are listening to which network port. A **port scanner** is a piece of software that will send probes to all network ports in order to trigger responses, mapping the attack vectors by identifying open network services.

Identification of zero-day vulnerabilities is done with fuzzing or static analysis of the code. **Fuzzing tools**, f**uzzers**, or **robustness testing tools** send a multitude of generated unexpected and abnormal inputs to a service in order to reveal both known and unknown vulnerabilities. These are studied in more detail in clause 8.

Catching a weakness in software requires monitoring of network data, logs and events, or process status. **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.

Penetration test can also be based on trying out a wide range of hostile **attack patterns**. **Exploit frameworks**, or **exploitation frameworks** are collections of operational malware scripts and tools that will compromise the system under test.

10 Model-based Security Testing

Most security testing practises are manual activities or tool driven processes. Functional security tests e.g. those for TVRA are checklist type verifications against functional requirements, and typically require security experts or consultants to be performed. Load, stress and performance tests are at best automated to run typical good use-case scenarios based on real deployed system and data, but very rarely explore the threat surface any deeper into misbehaving scenarios or incorrect data. Robustness tests are often hand-written or random, and provide no confidence on the test coverage. The transition to better safety, security and reliability is driven by more intelligence in the test scenarios providing more test coverage.

One improvement for test metrics and improved test coverage is Model-based Testing (MBT), which constitutes a number of technologies, methods, and approaches with the aim, to improve the quality, the efficiency, and the effectiveness of test processes, tasks and artefacts. Model-based Security Testing (MBST) is a special form of MBT that is focused on security aspects, including security features, performance and robustness. The models that are currently used are usually abstract, partial representations of the system under test specifying the desired behaviour and the interaction with the environment.

Model-based security testing involves defining the system and security requirements, building the behavioural model, defining test selection or generation criteria and transforming them into operational test case specifications. When time is limited, the test selection criteria can be narrowed down by risk assessment. The overall MBST process is completed by generating tests, implementing and setting up adaptor components and the test environment and finally by executing the tests on the SUT. Especially in case of robustness testing, the model-based security test generation will have to consider on-the-fly observations from current or previous test execution to guide the test generation process. This type of dynamic test generation behaviour is called online testing, and is needed in security testing where the number of tests is easily in millions of test cases. Documenting millions of test cases can be also challenging without automated generation of test documentation.

Greatest benefit to model-based security testing is repeatability and maintenance of the security tests for regression purposes. A security test that finds a critical failure such as a crash during tests is typically integrated into regression testing for later verification that the bug does not re-emerge later. Security flaws are also often found in third party libraries and components, requiring other development groups to reproduce the same message sequences and data in other locations.

History

Document history		
V1.1.1	March 2015	Publication