Security Testing Terminology and Concepts

Abstract

Describe Security Testing Terminology and Concepts/Abstract here.

Introduction

Security Testing

The purpose of security testing is to find out whether the system meets its specified security objectives, or security requirements. Security testing is performed at various phases in the product lifecycle, starting from requirements definition and analysis, through design, implementation and verification, all the way to maintenance.

In this document we will look at how security testing maps into software development, from testers perspective. The purpose is to provide an introduction to security testing, a starting point for people who are not familiar with security testing methods and where they are used.

This first chapter will introduce the basic terminology and concepts regarding security testing. The second chapter will look at where security testing is performed in the product lifecycle. In the third chapter, we will examine where and how security test requirements are collected. Finally the last three chapters discuss three different types of security testing: functional, performance and robustness testing.

Types of security testing

The term *security testing* means different things to different people. Here are some related work from ETSI to different methodologies in security testing. Security engineering starts with **Risk and Threat Analysis**, which are covered by the ETSI TVRA. Risk and threat analysis are focused on identification of risks and threats to a system at early phase during requirements analysis, or late in the process, reactively during security assessments and acceptance analysis at the validation phase.

Tests during the implementation of the software are mostly based on static code analysis, which is out of scope for this technical report.

During verification and validation, security tests can be divided in three main domains (Figure 1).

Figure 1: The Security Testing Triangle.

Functional testing for security is explained in more detail in TVRA, and is focused on testing the security functionality in the system. In functional tests, "positive" software requirements result in use cases, and are implemented in functional test cases. **Performance testing** overview can be found from TR by MTS, which is extended here for security testing. In performance testing, any use case is executed sequentially and in parallel in order to find performance bottlenecks. **Robustness testing, or Fuzzing** is covered here for the first time at ETSI, providing an introduction and starting point for further work. In robustness testing, thousands of misuse cases are built for each use case, exploring the infinite input space, testing a wide range of unexpected inputs that can cause problems to the software.

Penetration testing and vulnerability testing (included in TVRA) are typically performed late in the product lifecycle, by security specialists, not by testers, and therefore are out of scope for this document. They aim at verifying that known vulnerabilities are not left into the code.

Basic security testing terminology

A **Vulnerability** is a Weakness, a Bug, in code that can be used by malicious people to cause **Failure** in the operation of the software. A **Known Vulnerability** is a known weakness in software that has been found in the past, and that can easily be exploited by **Attacks**. **Unknown Vulnerability**, or **Zero-day Vulnerability** is a weakness that is hiding in software waiting for later discovery. Vulnerabilities can be caused by Design, Implementation and Configuration mistakes.

An **Attack** is a process or script, malicious code or malware that can be launched to trigger a vulnerability. **Zero-day Attack** is a special form of attack that exploits an unknown vulnerability, and therefore cannot be protected against. A **'Threat** is the possibility of an attack, and **Risk** is the probability of an attack. **Threat Agent** is the person or automated software that will realize the threat. Risk is sometimes also called **Attack Potential**.

Exploitability is often divided by which security target the vulnerability threatens: Confidentiality, Integrity or Availability. 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 **Attack Surface** is analyzed by **Attack vector analysis** and **Attack surface analysis**, where the first one looks at the interfaces and the second one looks at the code that can be attacked.

Testing tools

Security tests using **Static Analysis**, also called Static Application Security Testing (SAST), analyze the source code or the binary without executing it. Security tests using **Dynamic Analysis**, or Dynamic Application Security Testing (DAST), execute the code and analyze the behavior. A **Vulnerability Scanner** is a library or 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 vulnerabilities. Monitoring tools and Instrumentation, or Instruments, analyze the network traffic or the binary, or the operating platform, in order to detect failures and abnormal behavior that could indicate existence of a vulnerability.

Test verdicts in Security Testing

A Failure, or Fault, in software is the indication of a vulnerability. **Fail Safe** means the software can control the failure and restrict the exploitability of the vulnerability. **Fail Open** means the software will attempt to recover from the failure, and **Fail Closed** means the software will attempt to shut itself down in case of a vulnerability to prevent further attack attempts. **False Positive** means that a vulnerability was detected, but it is not a real vulnerability. **False Negative** means a vulnerability was not detected even if there was one.

Observability of Failures/Faults is critical in security testing. A **Fault Tolerant System** 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. Failure traces, audit traces, and crash traces are critical for analyzing the exploitability of failures. Log files and debug logs are required for fault identification and repair.

Use Cases for Security Testing

Security Testing is not a monolithic, stand alone activity, but rather can take place at a number of differing stages of the System Development Lifecycle (SDLC).

The various clusters of testing activity are:

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

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

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

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

A model to map these against a generic system lifecycle - as derived from ISO/IEC 15288 "Systems and software engineering -- System life cycle processes" is provided at Annex A.

Security Test Requirements

Requirements are drawn from:

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

Risk Assessment / Analysis

According to [C.Eckert 2004 Oldenburg-Verlag: IT-Sicherheit, Chapter 4 Security Engineering] Risk Assessment means the risk analysis of threats by calulating their probabilites of occurance. The probability of occurance of a threat specifies the product of the effort an attacker must take with the gain an attacker expects from executing the threat successfully.

The "method and proforma for threat, risk, vulnerability analysis" (TVRA) as presented in [ETSI TS 102 165-1 TISPAN methods and protocols part 1: TVRA] risk assessment is to be achieved by steps 6 and 7 (out of the 10 steps comprising TVRA method). Step 6 is the "calculation of the likelihood of the attack and its impact" whereas step 7 comprises the "establishment of the risks" by assessing the values of the asset impact plus the attack intensity plus the resulting impact (TVRA provides with value assessment tables accordingly).

Functional Testing

Functional Testing considers the system from the customers perspective, i.e. addressing the functionality from the user's viewpoint. This could include different testing types like interoperability and conformance testing on different levels. Functional security testing adopts this approach and also includes "users" not being intended, who wish to apply behaviour not intended, like consuming benefits from the system without registering etc.

In the following we provide a list of terms and concepts esthablished for traditional functional testing that appears also suitable for functinal security testing.

According to ISTQB functional testing is based on an analysis of the specification of the functionality on a target level (i.e. of a component or system) without knowledge of the internal structure (black-box testing), depending on e.g.

- scope: testing of *components* or the full *system*
- context: *Integration* or *interoperability* (IOP) testing or testing during the System Evaluation (common criteria)

The tests need to include a set of elements that froms the so-called test specification. The exact terms may differ in the different test notations:

- test scenarios including behaviour with data defining a (conditional) sequence of statements or actions
- expections: outcome or results in combination with test verdicts
- a configuration or architecture 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)

Independent from the selected notation the tests are presented in different styles: The ISO conformance test methodology (CTMF) had defined a clear definition of multiple abstraction levels and the (de)composition of single test goals. Following the understanding given in CTMF a distinction between abstract (specification) and executable (program/script) tests is recommended. Furthermore CTMF follows the understanding that a single test objective is implemented in 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 which may combine multiple test objectives in a single test case.

Traditional test developoment starts from the understanding of the choosen test method including test architecture etc. Taking into account a test base with all the requirements of the system/service under test next step is followed by the description of test purposes including test objectives that must not be provided in a formal way.The following step to find test cases with some concrete test oracle, the conditions and test procedure (i.e. the sequence of test steps) belongs to the test design and results in the test model. Final development step adds a test selection criterion towards a conditional execution considering special prerequsites only. In most cases the test generation happens offline, i.e. before any test execution. We speak about online test generation if test generation considers also observations from any test execution (see below).

From the methodological viewpoint of the test process the test development is followed by the realisation and test execution, i.e. the interaction of the implementation under test (IUT) and the test system. This step may require a test bed or test tool/harness. Any parameterization of the test need to have concrete settings to select and/or instantiate the test suite. The values are provided in the Implementation Conformance Statement (ICS) and Implementation eXtra Information for Testing (IXIT). The final steps in the realm of functional testing addresses the test analysis. A test Comparator (tool) may be used for an automated comparison of observation and expectation.

Functional testing from the Common Criteria viewpoint focus on the Target of Evaluation (TOE) security functional interfaces that have been identified as enforcing or supporting Security Functional Requirements (SFRs) identified and stated for the TOE. The test documentation shall consist of test plans, expected test results and actual 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 dependencies on the results of other tests. Following the BSI application notes the Test plan (procedure) is an informal description of the tests. According to the related test, the description uses pseudo code, flow diagram etc.; related test vectors, test programmes are referenced.

References

- The Common Criteria for Information Technology Security Evaluation (CC)
- ISO 27000 series of standards have been specifically reserved by ISO for information security matters.
- rfc2828 (191 pages of definitions and 13 pages of references) provides abbreviations, explanations, and recommendations for use of information system security terminology.
- OUSPG's Glossary https://www.ee.oulu.fi/research/ouspg/Glossary
- ISTQB Glossray of Testing Terms
- ISO 9646-x CTMF multipart standard
- ETSI ES 202 951 V1.1.1 (2011-07) - Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations
- ETSI TR 102 840 V1.2.1 (2011-02) - Methods for Testing and Specifications (MTS); Model-based testing in standardisation
- BSI Guidelines for Evaluation Reports according to Common Criteria Version 3.1, Bonn, July 2010.

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. In worst case, the entire system can crash under overwhelming load.

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 1). 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.

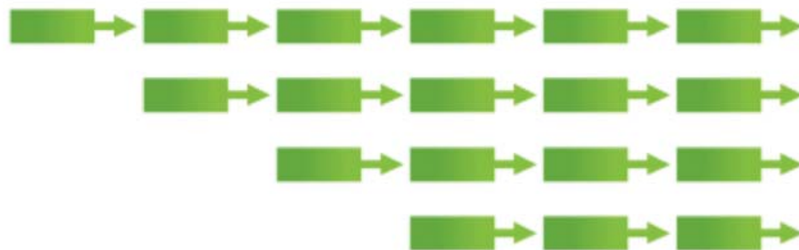

Figure 1: Parallel and Sequential Load.

A special case of attack is to send only the initial packets of a complex session, and never close the session. This could mean for example sending SIP INVITE messages without ACK reply, or opening a TCP session and never closing them. Timeouts for open sessions can affect the result of attacks.

A simple metric for load in security tests is the number of tests per second, or the number of sessions per second. For more detailed metric, the average amount of parallelism should also be measured.

Instrumentation required for performance includes monitoring data rate, CPU usage and disk usage. The purpose of instrumentation is to find out which resources are the bottleneck for security, and to identify the failure modes for each test scenario.

Solutions for load related attacks include load balancers, and early rejection of clearly repetitive messages from single source. Distributed attacks are harder to defend against as each attack comes from a different source address.

Fuzz Testing

Introduction

Fuzz testing, or Fuzzing, is a form of negative testing, where software inputs are randomly mutated or systematically modified in order to find security-related failures such as crashes, busy-loops or memory leaks. In some areas, fuzzing is also used to find any types of 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 whitenoise anomalies. Smart form of fuzzing is sometimes also called Grammar Testing, or Syntax Testing.

Fuzzing is a form of Risk-Based Testing, and is closely related to activities such as Attack Surface Analysis or Attack Vector Analysis in Risk Assessments. Fuzzing is typically performed in black-box testing manner, through the interfaces such as communiation protocols, command-line parameters or windows events. This interface testing can be either local or remote. As these inputs are modified to include anomalies or faults, fuzz testing was also sometimes called Input Fault Injection, although this name is very rarely used.

Terminology:

- **Fuzzing, Fuzz testing**: is a 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.
- **Robustness testing**: testing for robustness of the software system. Robustness is: "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. See also: error tolerance; fault tolerance." [IEEE Standard Glossary of Software Engineering Terminology, IEEE St. 610.121990]
- **Negative testing**: Testing for the absence of (undesired) functionality.
- **Grammar testing**: An abstract grammer, eg. an ABNF, serves as the basis for test case generation.
- **Syntax testing**: A grammer serves as the basis for testing the syntax of an ex- or implicit language.
- **Risk-based testing**: Testing is prioritized on the likelihood of detecting significant failures.
- **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 Standard Glossary of Software Engineering Terminology, IEEE St. 610.121990]
- **Input Fault Injection**: mutates the software or data at interfaces [Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security. 2001.]

Types of Fuzzers

"Smart Fuzzing" is typically based on behavioral model of the interface being tested. Fuzzing is smart testing when it is both protocol aware and has optimized anomaly generation. When fuzz tests are generated from a model built from the specifications, the tests and expected test results can also be documented automatically. Protocol awareness increases test efficiency and coverage, going deep in the behavior to test areas of the interfaces that rarely appear on typical use cases. Smart fuzzing is dynamic in behavior, with the model implementing the required functionality for exploring deeper in the message sequence. Dynamic functionality is programmed using keyword-based action code embedded into the executable model, but can also be implemented as precondition code or test script after which fuzzer steps in. The anomaly creating can also be optimized, and can go beyond simple boundary value analysis Smart model-based fuzzers explore a much wider range of attacks including testing with data, structure and sequence anomalies. The libraries of anomalies are typically built by error guessing, selecting known hostile data and 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 captures or files. In simplest form, template-based fuzzers will use the template sample as a binary block, modifying it quite blindly. Depending on the

fuzzing algorithm used, template-based fuzzing can appear similar to random whitenoise ad-hoc testing. Random test generators include everything from simple bit-flipping routines to more complex "move, replace and delete" algorithms.

Test generation in fuzz testing can be either on-line or off-line. Online test generation has the benefit of adapting to the behavior 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:

- Specification-based fuzzer
- Model-based fuzzer
- Block-based fuzzer
- Random fuzzer
- Mutation fuzzer
- Evolutionary/Learning fuzzer
- File fuzzer
- Protocol fuzzer
- Client fuzzing
- Server fuzzing

Fuzzing test setup and test proces

Fuzz testing phases:

1. Identification of Interfaces
2. Verifying Interoperability
3. Setting up Instrumentation
4. Test generation and execution
5. Reporting and reprodution

First step in starting fuzz testing is analyzing all interfaces in the software, and prioritizing those based on how likely they are to be attacked. Selection of fuzzing tools based on the initial risk assessment can take into account e.g. how likely each fuzzing strategy is in finding vulnerabilities, and how much time there is for test execution.

Second important phase is verifying that the test generator interacts correctly with the test target, and that the tests are processed correctly. The number of use scenarios that need to be fuzz tested can be narrowed down by e.g. using code coverage to see that adequate attack surface in the code is covered by the valid use scenarios. In template based fuzzing, this is also called "corpus distillation", selection of the optimal seed for fuzz tests.

Setting up instrumentation can consist of debuggers and other active monitors in the test device, but also passive monitoring such as network analysis and checks in the behavioral model itself. Changes in the executing process can be detected using operating system monitors, some of which can also be remotely monitored using SNMP instrumentation. Virtual cloud setups allow one additional monitoring interface, being able to monitor the operating system from outside e.g. in case of kernel level failures.

Finally test generation and execution should be as automated as possible. Fuzz testing is often used in build tests and regression tests, requiring full automation of the test setup independent from changes in the test target. During test execution various logging levels can allow to save significant amount of storage space when test case volume is in tens of millions of test cases.

Last and most important step for fuzzing is reporting and reproduction. Adequate data collection about the test case should be stored for all failed tests for test reporting and automated test case reproduction.

Critical area for fuzz testing is understanding different types of failures and categorizing and prioritizing different test verdicts. Each test case can have three different inline test results. The anomalous message can result in:

1. expected response
2. error response
3. no response

The test can also generate other external results such as error events to logs or anomalous requests to backend systems. A simple comparison of valid responses and normal software behavior to the behavior under fuzz testing can reveal majority of the failures. Use of debugging frameworks and virtualization platforms will help in catching low level exceptions that would otherwise go undetected if the software tries to hide such failures.

Fuzzing Requirements and Metrics

Simplest fuzzer metric is looking at the number of test cases, and number of found failures. This failure rate is a similar rating as MTBF (Mean Time Between Failures), basically an estimate how much fuzzing the system can survive. The simplest metric for fuzzer survivability is defining a number of tests that a system must survive without failures. Unfortunately this metric promoted "dumb" fuzzing, as it is less likely to find failures in the test target. Still, with right choice of tools, this metric is closest to the typical risk assessment estimation: resources needed for breaking a system can be calculated from the time needed to find a flaw using a special maturity model fuzzer.

Test coverage is second step in measuring fuzzing efficiency. The most objective metric for fuzzing is specification coverage, looking at what areas of the interface are included in the behavioral model of the fuzzer, or covered by the template use cases in case of template-based fuzzing. Anomaly coverage, or input coverage, looks at what types of anomalies are covered by the test generator. Finally, an implementation specific metric is looking at code or branch coverage of the target of testing itself. Multi-field anomalies will

grow the number of test cases exponentially, and will make coverage measurement difficult.

Fuzzer maturity model is the second greatest challenge. A simple 5 step maturity model consists of analyzing the various metrics related to fuzzing. Note that these steps are not inclusive, but a fuzzer can implement one or several of the different maturity levels:

1. random whitenoise fuzzing with scenarios based on templates
2. random fuzzing using educated algorithms with scenarios based on templates
3. model-inference based fuzzing from templates
4. evolutionary fuzzing
5. model-based fuzzing based on templates
6. model-based fuzzing created from input specification
7. adaptive model-based fuzzing created from specification

The most neutral means for fuzzer comparison has been done using error seeding. In error seeding, a range of fuzzers are executed against same implementation, in which wide range of different types flaws have intentionally been implemented. Fuzzers are compared based on which flaws they are able to find.

Fuzzing performance is about how fast tests are generated and executed. Depending on the interface, there can be several different metrics, with simplest one being test cases per second. The complexity of a test case will have significant impact on this metric. For test execution, also the test generation speed can sometimes have significant impact, especially if tests need to be regenerated several times.

Annex A: Mapping Tests To Lifecycle

Activity	ISO/IEC 15288										
	Stakeholder Requirements Definition (SRD)	Requirements Analysis (REQ)	Architectural Design (DES)	Implementation (IMP)	Integration (INT)	Verification (TST)	Transition (TRA)	Validation (VAL)	Operation (OPE)	Maintenance (MAI)	Disposal (DIS)
Architecture Framework Level 1 (Conceptual)	Architectural Reference Model										
Architecture Framework Level 2 (Contextual)		Architectural Reference Case									
Architecture Framework Level 3 (Logical)			Architectural Specification Case								
Architecture Framework Level 4 (Physical)		Design / Effect Class Selection	Design / Effect Pattern / Package Selection	Design / Effect Implementation	Design / Effect Integration						
Architecture Framework Level 5 (Detailed)			**Product:** Component Design **System:** Overall Design	**Product:** Component Implementation **System:** Component Selection	**Product:** Component Integration / Configuration **System:** Product Integration / Configuration	**Product/System:** Acceptance	**Product/System:** Delivery			**Product/System:** Upkeep (Configuration and Patching)	**Product/System:** Decommissioning Process Definition
Assurance Framework Level 1 (Conceptual)	Producer Organisation (PRD) Specification Competence			Producer Organisation (PRD) Realisation Competence					In Service Management Organisation (MGT) Competence		
Assurance Framework Level 2 (Contextual)	**Product:** Adversity (Hazard + Threat) Analysis **System:** Asset & Adversity (Hazard + Threat) Analysis	**Product/System:** Vulnerability Analysis	**Product/System:** Risk Analysis	**Product/System:** Control Selection					**Product/System:** Risk Monitoring	**Product/System:** Revised Risk Analysis	
Assurance Framework Level 3 (Logical)			**Product/System:** Initial Assurance Case				**Product/System:** Final Assurance Case	**Product:** Assurance Review **System:** Assurance and Acceptance Review		**Product/System:** Assurance Case Update and Compliance Review	**System:** Disposal Review
Assurance Framework Level 4 (Physical)	Practictioner (PRA) Specification Competence			Practictioner (PRA) Realisation Competence					Practictioner (PRA) In Service Management Competence		
Assurance Framework Level 5 (Detailed)				**Product:** Component Test **System:** Component Test Review	**Product:** Integration Test **System:** Integration Test and Product Test Review	**Product/System:** Acceptance Test	**Product:** Weakness Test **System:** Penetration Test	**Product:** Release Review **System:** Commissioning Review	**System:** Compliance Testing	**Product:** Weakness Test **System:** Penetration Test	**System:** Decomissioning Review

Key

Common application for Systems and Products

Differing application for Products and Systems

Applies only to one of Product or Systems