Security Testing Terminology and Concepts

Abstract

Describe Security Testing Terminology and Concepts/Abstract here.

Introduction

Security Testing

Security testing: Testing whether the system meets its specified security objectives.

Types of security testing:

- Risk assessment, Risk analysis (ISO/IEC, TVRA)
- Functional testing (CC, ETSI TVRA)
- Penetration testing (CC, TVRA)
- Vulnerability testing/auditing (CC, TVRA)
- Performance testing (recent perftest TR)
- Fuzzing (new content)

Why each part? How do they relate to security standardization? Where is it used? Map them to software/system lifecycle Who is using it? Builders? Buyers?

Basic terminology

Vulnerability:

- Weakness, Vulnerability, Bug
- Known vs. Unknown Vulnerability, Zero-day vulnerability
- Design, Implementation and Configuration vulnerabilities

Attack:

- Known vs. Unknown Attack, Zero-day attack
- Threat and Threat agent
- Attack tree
- Attack potential
- Malicious code, malicious logic, malware

Exploitability:

- Confidentiality, Integrity, Availability
- Denial of Service
- Distributed Denial of Service
- Buffer overflow and other memory handling bugs
- SQL injection and other execution bugs
- Directory traversal and other file handling bugs
- Other Availability issues: Busy loop, Memory leak

Testing tools:

- SAST, Static analysis
- DAST, Dynamic analysis
- Vulnerability Scanner
- Port scanner
- Fuzzing, Fuzz testing
- Monitoring

Attack Surface:

- Attack surface analysis
- Attack vector analysis

Test verdicts in Security Testing:

- Failure, Fault
- Failsafe - When software or device fails
- False positive
- False negative

Observability of Failures/Faults:

- Fault tolerance, Reliability engineering
- Instrument, Instrumentation
- Recovery
- Failure trace, Audit trace, Crash trace
- Logging, debug logs

Use Cases

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 lifecycle - as derived from ISO/IEC 15288 "Systems and software engineering -- System life cycle processes" is provided at Annex A.

Describe:

- Penetration Testing
- Vulnerability Testing

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

Describe Security Testing Terminology and Concepts/Performance Testing here.

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
- Negative testing
- Grammar testing
- Syntax testing
- Risk-based testing
- Black-box testing
- Input Fault Injection

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