Security Testing: A Survey

Michael Felderer, Matthias Büchler, Martin Johns,
Achim D. Brucker, Ruth Breu, Alexander Pretschner

Abstract

Identifying vulnerabilities and ensuring security functionality by security testing is a widely applied measure to evaluate and improve the security of software. Due to the openness of modern software-based systems, applying appropriate security testing techniques is of growing importance and essential to perform effective and efficient security testing. Therefore, an overview of actual security testing techniques is of high value both for researchers to evaluate and refine the techniques and for practitioners to apply and disseminate them. This chapter fulfills this need and provides an overview of recent security testing techniques. For this purpose, it first summarize the required background of testing and security engineering. Then, basics and recent developments of security testing techniques applied during the secure software development lifecycle, i.e., model-based security testing, code-based testing and static analysis, penetration testing and dynamic analysis, as well as security regression testing are discussed. Finally, the security testing techniques are illustrated by adopting them for an example three-tiered web-based business application.

1 Introduction

Modern IT systems based on concepts like cloud computing, location-based services or social networking are permanently connected to other systems and handle sensitive data. These interconnected systems are subject to security attacks that may result in security incidents with high severity affecting the technical infrastructure or its environment. Exploited security vulnerabilities can cause drastic costs, e.g., due to downtimes or the modification of data. A high proportion of all software security incidents is caused by attackers who exploit known vulnerabilities [115]. An important, effective and widely applied measure to improve the security of software are security testing techniques which identify vulnerabilities and ensure security functionality.

Software testing is concerned with evaluation of software products and related artifacts to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. Security testing verifies and validates software system requirements related to security properties like confidentiality, integrity, availability, authentication, authorization and non-repudiation. Sometimes security properties come as classical functional requirements, e.g., "user accounts are disabled after three unsuccessful login attempts" which approximates one part of an authorization property and is aligned with the software quality standard ISO/IEC 9126 [66] defining security as functional quality characteristic. However, it seems desirable that

security testing directly targets the above security properties, as opposed to taking the detour of functional tests of security mechanisms. This view is supported by the ISO/IEC 25010 [68] standard that revises ISO/IEC 9126 and introduces Security as a new quality characteristic which is not included in the characteristic functionality any more.

Web application security vulnerabilities such as Cross-Site Scripting or SQL Injection, which can adequately be addressed by security testing techniques, are acknowledged problems [11] with thousands of vulnerabilities reported each year [91]. Furthermore, surveys as published by the National Institute of Standards and Technology [96] show high cost of insecure software due to inadequate testing even on an economic level. Therefore, support for security testing, which is still often considered as a "black art", is essential to increase its effectiveness and efficiency in practice. This chapter intends to contribute to the growing need for information on security testing techniques by providing an overview of actual security testing techniques. This is of high value both for researchers to evaluate and refine existing techniques and practitioners to apply and disseminate them. In this chapter, security testing techniques are classified (and also the discussion thereof) according to their test basis within the secure software development lifecycle into four different types: (1) *model-based security testing* is grounded on requirements and design models created during the analysis and design phase, (2) *code-based testing and static analysis* on source and byte code created during development, (3) *penetration testing and dynamic analysis* on running systems, either in a test or production environment, as well as (4) *security regression testing* performed during maintenance.

This chapter provides provides a comprehensive survey on security testing and is structured as follows. Section 2 provides an overview of the underlying concepts on software testing. Section 3 discusses the basic concepts of security engineering and the secure software development lifecycle. Section 4 provides an overview of security testing and its integration in the secure software development lifecycle. Section 5 discusses the security testing techniques model-based security testing, code-based testing and static analysis, penetration testing and dynamic analysis as well as security regression testing in detail. Section 6 discusses the application of security testing techniques to three tiered business applications. Finally, Section 7 summarizes this chapter.

2 Software Testing

According to the classic definition in software engineering [17], *software testing* consists of the *dynamic* verification that a program provides expected behaviors on a *finite* set of test cases, a so called *test suite*, suitably *selected* from the usually infinite execution domain. This dynamic notion of testing, so called *dynamic testing*, evaluates software by observing its execution [4]. The executed system is called *system under test* (SUT). More general notions of testing [69] consist of all lifecycle activities, both static and dynamic, concerned with evaluation of software products and related artifacts to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. This definition also takes *static testing* into account, which checks software development artifact (e.g., requirements, design or code) without execution of these artifacts. The most prominent static testing approaches are (manual) reviews and (automated) static analysis, which are often combined with dynamic

testing, especially in the context of security. For security testing, the general notion of testing comprising static and dynamic testing is therefore frequently applied [99, 121, 9], and thus also in this chapter testing comprises static and dynamic testing.

After running a test case, the observed and intended behaviors of a SUT are compared with each other, which then results in a *verdict*. Verdicts can be either of *pass* (behaviors conform), *fail* (behaviors don't conform), and *inconclusive* (not known whether behaviors conform) [65]. A *test oracle* is a mechanism for determining the verdict. The observed behavior may be checked against user or customer needs (commonly referred to as testing for *validation*), against a specification (testing for *verification*), A *failure* is an undesired behavior. Failures are typically observed (by resulting in verdict fail) during the execution of the system being tested. A *fault* is the cause of the failure. It is a static defect in the software, usually caused by human error in the specification, design, or coding process. During testing, it is the execution of faults in the software that causes failures. Differing from active execution of test cases, passive testing only monitors running systems without interaction.

Testing can be classified utilizing the three dimensions objective, scope, and accessibility [125, 141] shown in Figure 1.

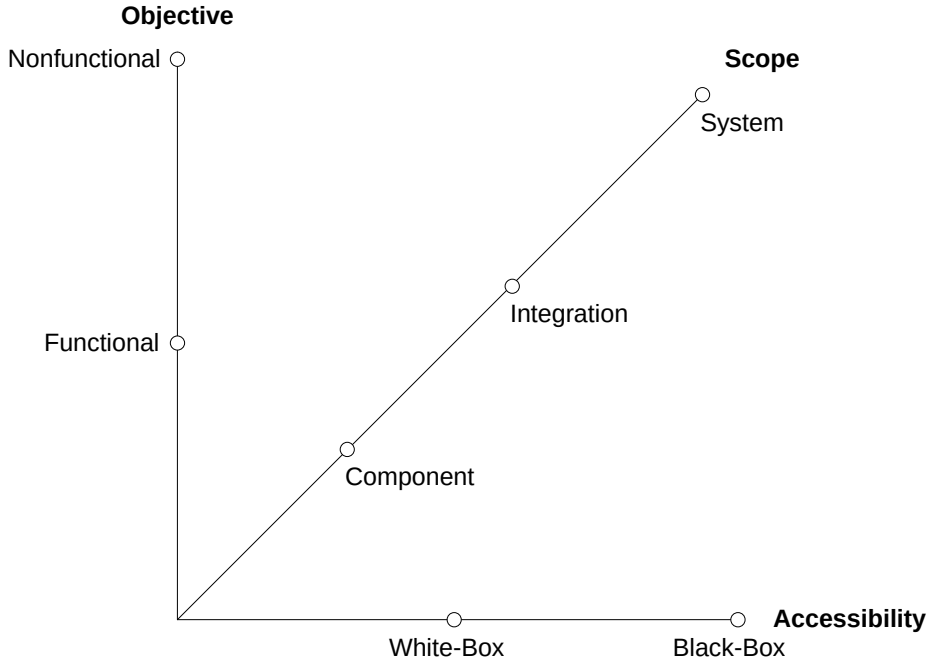

Figure 1: Testing Dimensions Objective, Scope, and Accessibility

Test objectives are reason or purpose for designing and executing a test. The reason is either to check the functional behavior of the system or its nonfunctional properties. *Functional testing* is concerned with assessing the functional behavior of an SUT, whereas *nonfunctional testing* aims at assessing nonfunctional requirements with regard to quality characteristics like security, safety, reliability or performance.

The *test scope* describes the granularity of the SUT and can be classified into component, integration and system testing. It also determines the *test basis*, i.e., the artifacts to derive test cases. *Component testing* (also referred to as *unit testing*) checks

the smallest testable component (e.g., a class in an object-oriented implementation or a single electronic control unit) in isolation. *Integration testing* combines components with each other and tests those as a subsystem, that is, not yet a complete system. *System testing* checks the complete system, including all subsystems. A specific type of system testing is *acceptance testing* where it is checked whether a solution works for the user of a system. *Regression testing* is a selective retesting to verify that modifications have not caused side effects and that the SUT still complies with the specified requirements [107].

In terms of *accessibility* of test design artifacts we can classifying testing methods into white-box and black-box testing. In *white-box testing*, test cases are derived based on information about how the software has been designed or coded [17]. In *black-box testing*, test cases rely only on the input/output behavior of the software. This classification is especially relevant for security testing, as black-box testing, where no or only basic information about the system under test is provided, enables to mimic external attacks from hackers. In classical software testing, a related classification of test design techniques [64] distinguishes between *structure-based testing techniques* (i.e., deriving test cases from internal descriptions like implementation code), *specification-based testing techniques* (i.e., deriving test cases from external descriptions of software like specifications), and *experience-based testing techniques* (i.e., deriving test cases based on knowledge, skills, and background of testers).

The process of testing comprises the core activities test planning, design, implementation, execution, and evaluation [69]. According to [63] and [69], *test planning* is the activity of establishing or updating a test plan. A test plan includes the test objectives, test scope, and test methods as well as the resources, and schedule of intended test activities. It identifies, amongst others, features to be tested and exit criteria defining conditions for when to stop testing. Coverage criteria aligned with the tested feature types and the applied test design techniques are typical exit criteria. Once the test plan has been established, test control begins. It is an ongoing activity in which the actual progress is compared against the plan which often results in concrete measures. During the *test design* phase the general testing objectives defined in the test plan are transformed into tangible test conditions and abstract test cases. For test derivation, specific test design techniques can be applied, which can according to ISO/IEC/IEEE 29119 [64] be classified into specification-based, structure-based and experience-based techniques. *Test implementation* comprises tasks to make the abstract test cases executable. This includes tasks like preparing test harnesses and test data, providing logging support or writing test scripts which are necessary to enable the automated execution of test cases. In the *test execution* phase, the test cases are then executed and all relevant details of the execution are logged and monitored. In manual test execution, testing is guided by a human, and in automated testing by a specialized application. Finally, in the *test evaluation* phase the exit criteria are evaluated and the logged test results are summarized in a test report.

In *model-based testing* (MBT), manually selected algorithms automatically and systematically generate test cases from a set of models of the system under test or its environment [114]. Whereas test automation replaces manual test execution with automated test scripts, MBT replaces manual test designs with automated test designs and test generation.

3 Security Engineering

In this section, we cover basic concepts of security engineering as well as an overview of the secure software development lifecycle.

3.1 Basic Concepts

Security testing validates software system requirements related to security properties of assets that include confidentiality, integrity, availability, authentication, authorization and non-repudiation. These security properties can be defined as follows [27]:

- *Confidentiality* is the assurance that information is not disclosed to unauthorized individuals, processes, or devices.
- *Integrity* is provided when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.
- *Availability* guarantees timely, reliable access to data and information services for authorized users.
- *Authentication* is a security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information.
- *Authorization* provides access privileges granted to a user, program, or process.
- *Non-repudiation* is the assurance that none of the partners taking part in a transaction can later deny of having participated.

Security requirements can be formulated as *positive requirements*, explicitly defining the expected security functionality of a security mechanism, or as *negative requirements*, specifying what the application should not do [99]. For instance, for the security property authorization as positive requirements could be "User accounts are disabled after three unsuccessful login attempts.", whereas a negative requirement could be formulated as "The application should not be compromised or misused for unauthorized financial transactions by a malicious user." The positive, functional view on security requirements is aligned with the software quality standard ISO/IEC 9126 [66] defining security as functional quality characteristic. The negative, non-functional view is supported by the ISO/IEC 25010 [68] standard that revises ISO/IEC 9126 and introduces Security as a new quality characteristic which is not included in the characteristic Functionality any more.

An *asset* is a data item, or a system component that has to be protected. In the security context, such an asset has one or multiple security properties assigned that have to hold for that asset.

A *fault* is a textual representation of what goes wrong in a behavioral description. It is the incorrect part of a behavioral description that needs to be replaced to get a correct description. Since faults can occur in dead code — code that is never executed —, and because faults can be masked by further faults, a fault does not necessarily lead to an error. At the other side, an error is always produced by a fault. A fault is not necessarily related to security properties but is the cause of errors and failures in general.

A *vulnerability* is a special type of fault. If the fault is related to security properties, it is called a vulnerability. A vulnerability is always related to one or more assets and

their corresponding security properties. An exploitation of a vulnerability attacks an asset by violating the associated security property. Since vulnerabilities are always associated with the protection of an asset, the security relevant fault is usually correlated with a mechanism that protects the asset. A vulnerability either means that (1) the responsible security mechanism is completely missing, or (2) the security mechanism is in place but is implemented in a faulty way.

An *exploit* is a concrete malicious input that makes use of the vulnerability in the system under test (SUT) and violates the property of an asset. Vulnerabilities can often be exploited in different ways. One concrete exploit selects a specific asset and a specific property, and makes use of the vulnerability to violate the property for the selected asset.

A *threat* is the potential cause of an unwanted incident that harms or reduces the value of an asset. For instance, a threat may be a hacker, power outages, or malicious insiders. An *attack* is defined by the steps a malicious or inadvertently incorrectly behaving entity performs to the end of turning a threat into an actual corruption of an asset's properties. This is usually done by exploiting a vulnerability.

Security aspects can be considered on the network, operating system and application level. Each level has its own security threats and corresponding security requirements to deal with them. Typical threats on the network level are distributed denial-of-service or network intrusion. On the operating system level, all types of malware cause threats. Finally, on the application level threats typical threats are related to access control or are application type specific like Cross-Site Scripting in case of web applications. All levels of security can be subject to tests.

Security testing simulates attacks and employs other kinds of *penetration testing* attempting to compromise the security of a system by playing the role of a hacker trying to attack the system and exploit its vulnerabilities [7]. Security testing requires specific expertise which makes it difficult and hard to automate [102]. By identifying risks in the system and creating tests driven by those risks, security vulnerability testing can focus on parts of a system implementation in which an attack is likely to succeed.

Risks are often used as a guiding factor to define security test processes. For instance, Potter and McGraw [102] consider the process steps creating security misuse cases, listing normative security requirements, performing architectural risk analysis, building risk-based security test plans, wielding static analysis tools, performing security tests, performing penetration testing in the final environment, and cleaning up after security breaches. Also the Open Source Security Testing Methodology Manual (OSSTMM) [56] and the OWASP Testing Guide [99] take risks into account for their proposed security testing activities.

3.2 Secure Software Development Lifecycle

Testing is often started very late in the software development lifecycle shortly before it is deployed. It has turned out that this is a very ineffective and inefficient practice. One of the best methods to prevent security bugs from appearing in production applications is to improve the software development lifecycle by including security in each of its phases, thereby extending it to a secure software development lifecycle. A *software development lifecycle* is a series of steps, or phases, that provide a model for the development and lifecycle management of an application or piece of software. It is a structure imposed on the development of software artifacts. A generic software development lifecycle

model considering testing as an orthogonal dimension comprises the phases analysis, design, development, deployment, and maintenance [99]. Each phase delivers specific artifacts, i.e., the analysis phase results in requirements, design provides design models, development delivers code, deployment results in a running system, and finally all artifacts are maintained.

A secure software development lifecycle takes security aspects into account in each phase of software development. A crucial concept within the secure software development lifecycle is risk. A *risk* is the likelihood of an unwanted incident and its consequence for a specific asset [84]. Taking into account the negative nature of many security requirements, the concept of risk can be employed to direct the selection or application of security counter-measures like testing [84, 102]. In all phases of the secure software development process, but especially at the design level [127], risk analyses provide effective means to guide security testing and thus detect faults and vulnerabilities.

Major security development processes are the Security Development Lifecycle (SDL) [57] from Microsoft and the Open Software Assurance Maturity Model (OpenSAMM) [97] from OWASP.

Microsofts SDL is an established security lifecycle for software development projects pursuing the following major principles [57]:

- Secure by Design: Security is a built-in quality attribute affecting the whole software lifecycle.
- Security by Default: Software systems are constructed in a way that potential harm caused by attackers is minimized, e.g. software is deployed with least necessary privilege.
- Secure in Deployment: software deployment is accompanied by tools and guidance supporting users and/or administrators.
- Communications: software developers are prepared for occurring threats communicating openly and timely with users and/or administrators.

The SDL is composed of security practices attached with the major activities of a software lifecycle, i.e., requirements, design, implementation, verification and deployment in case of SDL, which are extended by the two activities training and response. For instance, the security practice "establish security requirements" is attached to requirements analysis, "use threat modeling" to design, "perform static analysis" to implementation, "perform fuzz testing" to verification, and "certify release and archive" to release.

Similar to the SDL, OpenSAMM attaches security practices to core activities, i.e., governance, construction, verification, and deployment in case of OpenSAMM, within the software development lifecycle. For instance, verification includes the security practices design review, code review, as well as (dynamic) security testing.

In particular, OpenSAMM attaches each security practice with three maturity levels and a starting point of zero:

- Level 0: Implicit starting point representing the activities in the practice being unfulfilled
- Level 1: Initial understanding and ad-hoc provision of security practice

- Level 2: Increase efficiency and/or effectiveness of the security practice
- Level 3: Comprehensive mastery of the security practice at scale

For each security practice and maturity level, OpenSAMM does not only define the objectives and activities, but also gives support to achieve this particular level. This comprises assessment questions, success metrics, costs and personnel needed to achieve the targeted maturity level.

4 Security Testing

In this section, we cover basic concepts of security testing and the integration of security testing in the secure software development lifecycle.

4.1 Basic Concepts

Security testing is testing of security requirements related to security properties like confidentiality, integrity, availability, authentication, authorization, and non-repudiation.

Security testing identifies whether the specified or intended security properties are, for a given set of assets of interests, correctly implemented. This can be done by trying to show conformance with the security properties, similar to requirements-based testing; or by trying to address known vulnerabilities, which is similar to traditional fault-based, or destructive, testing. Intuitively, conformance testing considers well-defined, expected inputs. It tests if the system satisfies the security properties with respect to these well-defined expected inputs. In contrast, addressing known vulnerabilities means using malicious, non-expected input data that is likely to exploit the considered vulnerabilities.

As mentioned in the previous section, security requirements can be positive and functional, explicitly defining the expected security functionality of a security mechanism, or negative and non-functional, specifying what the application should not do. This classification of security requirements also impacts security testing. For *positive security requirements classical testing techniques* can be applied applied, whereas for *negative security requirements* (a combination of) *additional measures* like risk analyses, penetration testing, or vulnerability knowledgebases are essential. This classification is also reflected in by classification in the literature as provided by Tian-yang et al. [121] as well as by Potter and McGraw [103].

According to Tian-yang et al. [121] two principal approaches can be distinguished, i.e., *security functional testing* and *security vulnerability testing*. Security functional testing validates whether the specified security requirements are implemented correctly, both in terms of security properties and security mechanisms. Security vulnerability testing addresses the identification of unintended system vulnerabilities. Security vulnerability testing uses the simulation of attacks and other kinds of *penetration testing* attempting to compromise the security of a system by playing the role of a hacker trying to attack the system and exploit its vulnerabilities [7]. Security vulnerability testing requires specific expertise which makes it difficult and hard to automate [103]. By identifying risks in the system and creating tests driven by those risks, security

vulnerability testing can focus on parts of a system implementation in which an attack is likely to succeed.

Potter and McGraw [103] distinguish between *testing security mechanisms* to ensure that their functionality is properly implemented, and *performing risk-based security testing* motivated by understanding and simulating the attacker's approach. Testing security mechanisms can be performed by standard test organizations with classical functional test techniques, whereas risk-based security testing requires specific expertise and sophisticated analysis [103].

For security vulnerability testing approaches, Shahriar and Zulkernine [118], i.e., *vulnerability coverage*, *source of test cases*, *test case generation method*, *testing level*, *test case granularity*, *tool automation* as well as *target applications*. Tool automation is further refined into the criteria *test case generation*, *oracle generation*, and *test case execution*. The authors classify 20 informally collected approaches according to these criteria. The main aim of the criteria is support for security practitioners to select an appropriate approach for their needs. Therefore Shahriar and Zulkernine blend abstract criteria like source of test cases or test case generation method with technological criteria like tool automation or target applications.

Differing from classical software testing, where black-box and white-box test design are nowadays considered very similar, i.e., in both cases testing proceeds from abstract models [4], the distinction is essential for security testing. White-box testing performs testing based on information about how the software has been designed or coded, and thus enables testing from an internal software producer point of view. [17]. Black-box testing relies only on the input/output behavior of the software, and thus enables to mimic external attacks from hackers. The classification into white- and black-box testing is also pointed out by Bachmann and Brucker [9], who additionally classify security testing techniques due to execution (dynamic vs. static testing) and automation (manual vs. automated testing).

In addition, due to the critical role of negative security requirements, classical testing which focuses on testing functional requirements and security testing differ. It seems desirable that security testing directly targets security properties, as opposed to taking the detour of functional tests of security mechanisms. As the former kind of (non-functional) security properties describe all executions of a system, testing them is intrinsically hard. Because testing cannot show the absence of faults, an immediately useful perspective directly considers the *violation* of these properties. This has resulted in the development of specific testing techniques like penetration testing that simulates attacks to exploit vulnerabilities. Penetration tests are difficult to craft because tests often do not directly cause observable security exploits, and because the testers must think like an attacker [103], which requires specific expertise. During penetration testing, testers build a mental model of security properties, security mechanisms, and possible attacks against the system and its environment. Specifying security test models in an explicit and processable way, results in a *model-based security testing approach*. In such an approach, security test models provide guidance for the systematic specification and documentation of security test objectives and security test cases, as well as for their automated generation and evaluation.

(Functional) testing normally focuses on the *presence of some correct behavior but not the absence of additional behavior*, which is implicitly specified by negative requirements. Testing routinely misses hidden action and the result is dangerous side effect

behaviors that ship with a software. Figure 2 illustrates this side effect nature of most software vulnerabilities that security testing has to cope with [120].

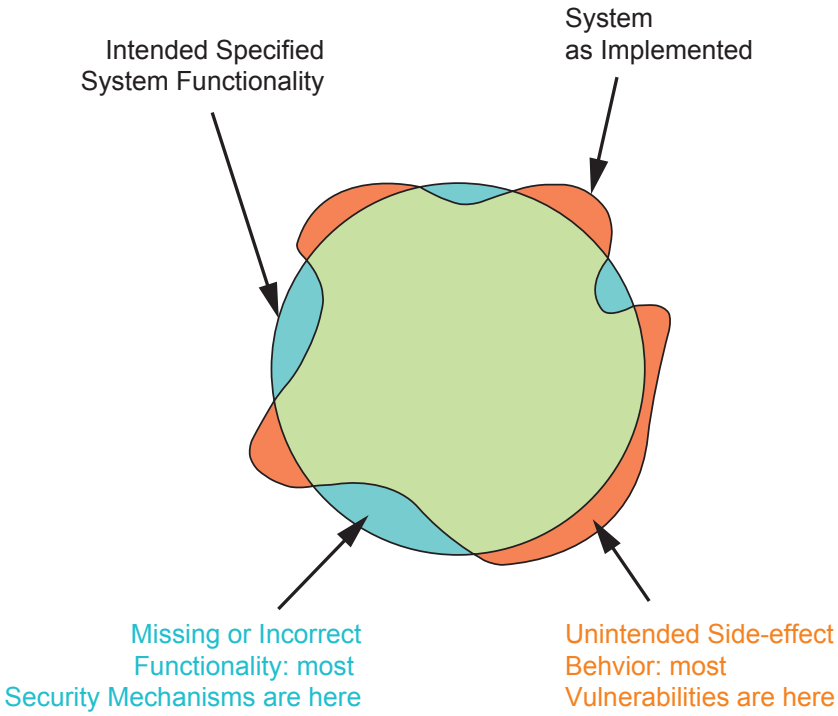

Figure 2: Most faults in security mechanisms are related to missing or incorrect functionality, most vulnerabilities are related to unintended side-effect behavior (adapted from Thompson [120])

The circle represents an application's intended functionality including security mechanisms, which is usually defined by the requirements specification. The amorphous shape superimposed on the circle represents the application's actual, implemented functionality. In an ideal system, the coded application would completely overlap with its specification, but in practice, this is hardly ever the case. The areas of the circle that the coded application does not cover represents typical functional faults (i.e., behavior that was implemented incorrectly and does not conform to the specification), especially also in security mechanisms. Areas that fall outside of the circular region represent unindented and potentially dangerous functionality, where most security vulnerabilities lay. The mismatch between specification and implementation shown in Figure 2 leading to faults in security mechanisms and vulnerabilities can be reduced by taking security and especially security testing aspects into account early and in all phases of the software development lifecycle as discussed in Section 4.2.

4.2 Security Testing in the Secure Software Development Lifecycle

As mentioned before, testing within the security lifecycle plays the role to validate and verify security requirements. Due to the negative nature of many security requirements

and the resulting broad range of subordinate requirements, also testing activities cover a broad range of scopes and employed methods. In keeping with research and experience, it is essential to take testing into account in all phases of the secure software development lifecycle, i.e., analysis, design, development, deployment, as well as maintenance. Thus, security testing must be holistic covering the whole secure software development lifecycle [9]. In concrete terms, Figure 3 shows a recommended distribution of static and dynamic testing efforts among the phases of the secure software development lifecycle according to [99]. It shows that security testing should be balanced over all phases, with a focus on the early phases, i.e., analysis, design, and implementation.

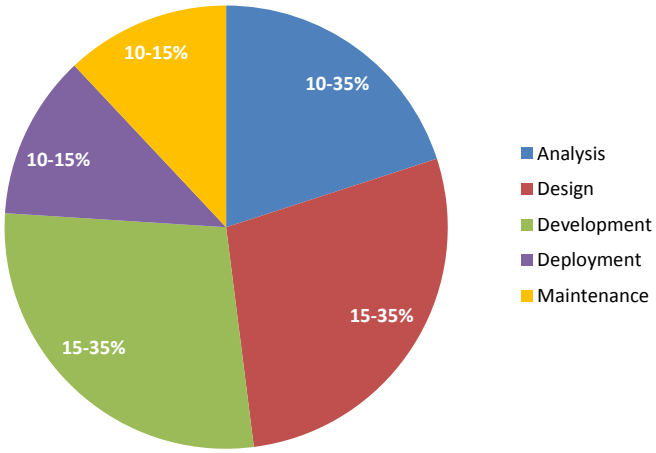

Figure 3: Proportion of Test Effort in Secure Software Development Lifecycle according to [99]

To provide support for the integration of security testing into all phases of the secure software development process, major security development processes (see Section 3.2), consider the integration of testing. In the Security Development Lifecycle (SDL) [57] from Microsoft practices with strong interference with testing efforts are the following:

- SDL Practice #2 (Requirements): Establish Security and Privacy Requirements
- SDL Practice #4 (Requirements): Perform Security and Privacy Risk Assessments
- SDL Practice #5 (Design): Establish Design Requirements
- SDL Practice #7 (Design): Use Threat Modeling
- SDL Practice #10 (Implementation): Perform Static Analysis
- SDL Practice #11 (Verification): Perform Dynamic Analysis
- SDL Practice #12 (Verification): Perform Fuzz Testing
- SDL Practice #13: Conduct Attack Surface Review
- SDL Practice #15: Conduct Final Security Review

In OpenSAMM [97] from OWASP, the verification activity includes the security practices design review, code review, as well as (dynamic) security testing.

The OWASP Testing Guide [99] and the OWASP Code Review Guide [98] provide a detailed overview of the variety of testing activities of web application security. While the Testing Guide has a focus on black-box testing, the Code Review Guide is a white-box approach focusing on manual code review. Overall, the Testing Guide distinguishes 91 different testing activities split into 11 sub-categories (i.e., information gathering, configuration and deployment management testing, identity management testing, authentication testing, authorization testing, session management testing, data validation testing, error handling, cryptography, business logic testing, as well as client side testing). Applying security testing techniques to web applications is covered in Section 6.

The OWASP testing framework workflow, which is also contained in the OWASP Testing Guide, contains checks and reviews of respective artifacts in all secure software development phases, creation of UML and threat models in the analysis and design phases, unit and system testing during development and deployment, penetration testing during deployment, as well as regression testing during maintenance. Proper security testing requires a mix of techniques as there is no single testing technique that can be performed to effectively cover all security testing and their application within testing activities at unit, integration, and system level. Nevertheless, many companies adopt only one security testing approach, for instance penetration testing [99].

Figure 4 abstracts from concrete security testing techniques mentioned before, and classifies them according to their test basis within the secure software development lifecycle.

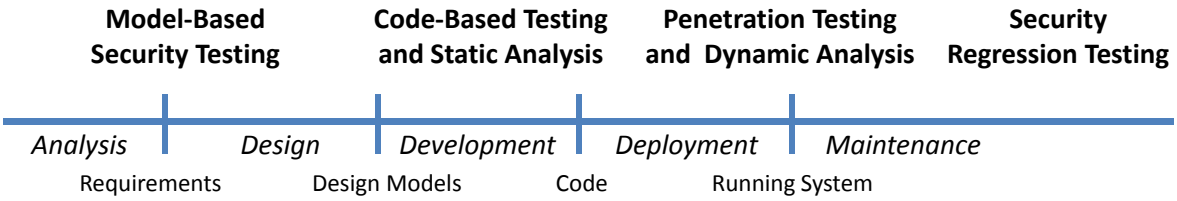

Figure 4: Security Testing Techniques in the Secure Software Development Lifecycle

Model-based security testing is grounded on requirements and design models created during the analysis and design phase. *Code-based testing and static analysis* is based on source and byte code created during development. *Penetration testing and dynamic analysis* is based on running systems, either in a test or production environment. Finally, *security regression testing* is performed during maintenance. We also apply this classification to structure the discussion of security testing techniques in the following section.

5 Security Testing Techniques

This section discusses the security testing techniques model-based testing, code-based testing and static analysis, penetration testing and dynamic analysis as well as regression testing in detail. For each testing technique, basic concepts as well as current approaches are covered.

5.1 Model-Based Security Testing

In model-based testing (MBT) manually selected algorithms automatically and systematically generate test cases from a set of models of the system under test or its environment [114]. MBT is an active area of research [31, 126] and offers big potential to improve test processes in industry [125, 114, 49]. Its prospective benefits include early and explicit specification and review of system behavior, better test case documentation, the ability to automatically generate useful (regression) tests and control test coverage, improved maintenance of test cases as well as shorter schedules and lower costs [114].

Process. The process of MBT consists of three main steps integrated into the overall process of test design, execution, and evaluation. (1) A model of the SUT and/or its environment is built from informal requirements, existing specification documents, or a SUT. The resulting model of the SUT dedicated to test generation is often called *test model*. (2) If they are executable, one execution trace of such a model acts as a test case: input and expected output for an SUT. Because there are usually infinitely many and infinitely long execution traces, models can therefore be used to generate an infinite number of tests. To cut down their number and length, *test selection criteria* are applied. These guide the generation of tests. (3) Once the test model and the test selection criteria are defined, a set of test cases is generated from the test model as determined by the chosen test selection criteria. Test generation is typically performed automatically. The generated test cases are traces of the model and thus in general at a higher level than the events or actions of an SUT. Therefore, the generated test cases are further refined to a more concrete level or adapted to the SUT to support their automated execution.

Model-based security testing. Model-based security testing (MBST) is an MBT approach that validates software system requirements related to security properties. It combines security properties like confidentiality, integrity, availability, authentication, authorization and non-repudiation with a model of the SUT and identifies whether the specified or intended security features hold in the model.

Both MBT and MBST have in common, that the input artifact is a model and not the SUT. Therefore the abstraction gap between the model and the SUT has to be addressed. In particular, an identified (security) issue at the model level does not automatically confirm an (security) issue at the SUT. Therefore an additional step is needed to map an abstract test case to an executable test case that can be executed on the SUT.

Selection criteria: "Good" test cases. Arguably, "good" test cases detect potential, rather than actual, defects with good cost effectiveness [104]. Potential defects need to be described by defect hypotheses. In order to turn these hypotheses into operational adequacy criteria [145], they need to be captured by some form of explicit defect model [93, 105, 104]. One form of defect is a fault, understood as the root cause of an incorrect system state (error) or incorrect system output (failure). As we show below, vulnerabilities can be understood as faults.

In addition to explicit models of (the functionality of) the system under test, model-based security testing usually makes use of one or more of the three following models

for test selection: properties, vulnerabilities, and attackers. *Models of an attacker* encode an attacker's behavior: the data they need, the different steps they take, the way they craft exploits. Attacker models can be seen as models of the environment of a system under test, and knowledge about a targeted vulnerability usually is left implicit. In contrast, *models of vulnerabilities* explicitly encode weaknesses in a system or a model of the system. In this sense, they can be seen as faults that are used for the generation of "good" test case generation (see above). Finally, *properties* describe desired characteristics of a model, or an implementation, and they include confidentiality, availability, integrity, and so on. Models of properties are needed to describe the properties of an asset that are supposed not to be violated: they describe what exactly the security tester targets, and what an exploit is supposed to exploit.

It is noteworthy that all forms of security testing, model-based or not, always work with an implicit or explicit hypothesis about a potential vulnerability.

Vulnerabilities as faults. Frequently, as a reaction to known relevant threats, assets are protected by explicit security mechanisms. Mechanisms include input sanitization, Address Space Layout Randomization (ASLR), encryption of password files, but also intrusion detection systems and access control components. Mechanisms are components of a system and can always be syntactically recognized: there is a piece of code (the mechanism) that is supposed to protect the asset; or there is no such piece of code. A vulnerability is a special kind of fault with security implications. It is defined as the absence of a correctly functioning mechanism. This can mean both (1) that there is no mechanism at all (e.g., no input sanitization takes place which can lead to buffer overflows or SQL Injections) and (2) that the mechanism does not work correctly, i.e., is partially or incorrectly implemented, for instance, if an access control policy is faulty.

Security testing can then be understood in three seemingly different ways: (1) to test if specific security properties of an *asset* can be violated (properties and property models); (2) to test the functionality of a *mechanism* (attacker models); and (3) to directly try to exploit a *vulnerability* (vulnerability models). The boundaries are blurred, however: With the above definition of vulnerabilities as the absence of effectively working defense mechanisms, and the observation that attacker models always involve implicit or explicit hypotheses on vulnerabilities, activities (2) and (3) are close to identical. In practice, they only differ in terms of the perspective that the tester takes: the mechanism or the vulnerability. Because the above definition also binds vulnerabilities to—possibly unspecified—assets, the goal of activities (2) and (3) always is activity (1). It hence seems hard to provide a crisp conceptual distinction between the three activities of (1) testing security properties, (2) testing security mechanisms, and (3) testing for vulnerabilities.

Classification of Model-Based (Security) Testing. Several publications have been published that propose taxonomies and classifications of existing MBT [126, 31] and MBST approaches [35, 116]. We will focus on the classification proposed by Schieferdecker et al. [116] considering different perspectives used in securing a system. The authors claim that MBST needs to be based on different types of models and distinguish three types of input models for security test generation, i.e., *architectural and functional models*, *threat, fault and risk models*, as well as *weakness and vulner-*

ability models. Architectural and functional models of the SUT are concerned with security requirements and their implementation. They focus on the expected system behavior. Threat, fault and risk models focus on what can go wrong, and concentrate on causes and consequences of system failures, weaknesses or vulnerabilities. Weakness and vulnerability models describe weaknesses or vulnerabilities by themselves.

In the following, we exemplary describe selected approaches, that make use of different models according to the classification of Schieferdecker et al.

5.1.1 A Model-based Security Testing Approach for Web Applications

An approach that makes use of functional, fault, and vulnerability models according to Schieferdecker et al. is presented by Büchler et al. [23]. They published a semi-automatic security testing approach for web applications from a secure model. The authors assume there is a formal model $\mathcal{M}$ for the specification of the System under Test (SUT). This model is secure as it does not violate any of the specified security goals (e.g., confidentiality, authenticity). Thus, a model-checker will report $\mathcal{M} \models \varphi$ for all security properties φ defining the security goals of the model. The model is built using abstract messages that are defined by the modeler. These messages represent common actions a user of the web application can perform. The idea is that these abstract messages are sent to the server to tell it which actions the client wants to perform, e.g., log in to the web application, view profiles of different users, delete profiles, update profiles, and so on. Thus, the modeler does not care about details at the browser/protocol level but only about abstract messages that represent web application actions.

To make use of such a secure model, Büchler et al. [23] define semantic mutation operators that represent common, well-known vulnerabilities at source code level. Semantic mutation operators are an abstraction that these vulnerabilities so that they can be injected into the model. After having applied a mutation operator to an original model, the model-checker may provide a trace from this mutated model that violates a security property. This trace is called an *attack trace* because it shows which sequence of abstract messages have to be exchanged in order to lead the system to a state where the security property is violated. Since abstract attack traces are at the same level of abstraction as the input model, they need to be instantiated to turn them operational. The approach proposes a multi-step instantiation since web applications are usually accessed via a browser. In the first step, abstract messages are translated into abstract browser actions. The second step is a mapping from these browser actions to executable API calls to make them operational in a browser. Finally, a test execution engine executes the operationalized test cases on the SUT to verify, if the implementation of the model suffers from the same vulnerability as reported by the model checker at the abstract level.

5.1.2 A Model-Based Framework for Security Policy Specification, Deployment and Testing

Mouelhi et al. [94] propose an approach based on architectural, functional, and fault models and focus on security policies. They propose a model-based approach for the specification, deployment and testing of security policies in Java applications. The

approach starts with a generic security meta-model of the application. It captures the high level access control policy implemented by the application and is expressed in a dedicated domain-specific language. Before such a model is further used, the model is verified to check the soundness and adequacy of the model with respect to the requirements. Afterwards the model is automatically transformed to policy decision points (PDP). Since such PDPs are usually not generated from scratch but are based on existing frameworks, the output of the transformation is, for instance, an XACML (Extended Access Control Markup Language) file that captures the security policy. This transformation step is essential in MBT since an identified security issue at model level does not automatically imply the same issue at implementation level, nor does a model without security issues automatically imply the same on the implementation. Mouelhi et al. make use of mutations at the model level to ensure that the implementation conforms to the initial security model. An existing test suite is executed on an implementation generated from a mutated security model. If such mutants are not detected by the existing test suite, it will be adapted to cover the mutated part of the security model as well. Finally the test objective is to check that the implementation (security policy) is synchronized with the security model.

5.1.3 Risk-Based Security Testing

In the following, we consider approaches that are based on risk models. Risk-based testing in general is a type of software testing that explicitly considers risks of the software system as the guiding factor to solve decision problems in all phases of the test process, i.e., test planning, design, implementation, execution and evaluation [46, 115, 38]. It is based on the intuitive idea to focus testing activities on those areas that trigger the most critical situations for a software system [135]. The precise understanding of risks as well as their focused treatment by risk-based testing has become one of the cornerstones for critical decisions within complex software development projects and recently gained much attention [38]. Lately, the international standard ISO/IEC/IEEE 29119 Software Testing [64] on testing techniques, processes and documentation even explicitly considers risks as an integral part of the test planning process. In the following, we describe three risk-based approaches to security testing in more detail.

Grossmann et al. [51] present an approach called Risk-Based Security Testing that combines risk analysis and risk-based test design activities based on formalized security test patterns. The involved security test patterns are formalized by using a minimal test design strategies language framework which is represented as a UML profile. Such a (semi-)formal security test pattern is then used as the input for a test generator accompanied by the test design model out of which the test cases are generated. The approach is based on the CORAS method [84] for risk analysis activities. Finally, a tool prototype is presented which shows how to combine the CORAS-based risk analysis with pattern-based test generation.

Botella et al. [16] describe an approach to security testing called Risk-Based Vulnerability Testing, which is guided by risk assessment and coverage to perform and automate vulnerability testing for web applications. Risk-Based Vulnerability testing adapts model-based testing techniques using a pattern-based approach for the generation of test cases according to previously identified risks. For risk identification and analysis, the CORAS method [84] is utilized. The integration of information from risk analysis activities with the model-based test generation approach is realized by a

test purpose language. It is used to formalize security test patterns in order to make them usable for test generators. Risk-Based Vulnerability Testing is applied to security testing of a web application.

Zech et al. [143, 142] propose a new method for generating negative security tests for non-functional security testing of web applications by logic programming and knowledge engineering. Based on a declarative model of the system under test, a risk analysis is performed and used for derivation of test cases.

5.2 Code-Based Testing and Static Analysis

Many vulnerabilities can only be detected by looking at the code. While traditionally not understood as a testing technique, *static analysis* of the program code is an important part of any security development process, as it allows to detect vulnerabilities at early stages of the development lifecycle where fixing of vulnerabilities is comparatively cheap [43]. In Microsoft's SDL [57], SAST is part of the *implementation* phase to highlight that this technique should be applied as soon as the first line of code is written. Note that in this section, we only discuss purely static approaches, i.e., approaches that do not require an executable test system. Thus, we discuss hybrid approaches, i.e., approaches that combine static analysis with dynamic testing (such as concolic testing) in Section 5.3.

Code reviews can either be done manually or automated. The latter is often called static code analysis (SCA) or Static Application Security Testing (SAST). Moreover, we can either analyze the source code (i.e., the code that was written by a developer) of the program or the compiled source code (i.e., binaries or byte-code). As they are closely related, we discuss them not separately. From a software vendor's perspective who is aiming at building secure software, the analysis on the source code is preferred over a binary analysis, as the source code analysis is more precise and can provide detailed recommendations to developers on how to fix a vulnerability on the source code level.

5.2.1 Manual Code Review

Manual code review is the process by which an expert is reading program code "line-by-line" to identify vulnerabilities. This requires expertise in three areas: the application architecture, the implementation techniques (programming languages, frameworks used to build the application), as well as security. Thus, a good manual code review should start with a threat model or at least an interview with the developers to get a good understanding of the application architecture, its attack surface, as well as the implementation techniques. After this, the actual code review can start in which code is, guided by the identified attack surface, manually analyzed for security vulnerabilities. Finally, the results of the analysis are reported back to development to fix the identified vulnerabilities as well as to educate architects and developers to prevent similar issues in the future. Overall, manual code reviews are a tedious process that requires skill, experience, persistence, and patience.

5.2.2 Static Application Security Testing

Automated *static program analysis* for finding security vulnerabilities, also called *Static Application Security Testing* (SAST) [26], is an attempt to automated code reviews: in principle, a SAST tool analyses the program code of a software component (e.g., an application or a library) automatically and reports potential security problems (potential vulnerabilities). This limits the manual effort to reviewing the reported problems and, thus, increases the scalability (i.e., the amount of program code that can be analyzed in a certain amount of time) significantly. Moreover, on the one hand, SAST tools "encapsulate" most of the required security expertise and, thus, they can (and should) be used by developers that are not necessarily security experts. On the other hand, SAST tools only report vulnerabilities they are looking for and, thus, there is still a need for a small team of experts that configures the SAST tools correctly [14, 21].

For computing the set of potential security problems in a program, a SAST tool mainly employs two different types of analysis:

1. *Syntactic checks* such as calling insecure API functions or using insecure configuration options. An example of this class would be an analysis of Java programs for calls to `java.util.random` (which does not provide a cryptographically secure random generator).
2. *Semantic checks* that require an understanding of the program semantics such as the data flow or control flow of a program. An example of this class would be an analysis checking for direct (not sanitized) data-flows from an program input to a SQL statement (indicating a potential SQL Injection vulnerability).

As SAST tools work on over-approximations of the actual program code as well as apply heuristics checks, the output of a SAST tool is a list of *potential* security vulnerabilities. Thus, or each finding, an human expert is necessary to decide:

- If the finding represents a vulnerability, i.e, a weakness that can be exploited by an attacker (*true positive*), and, thus, needs to be fixed.
- If the finding cannot be exploited by an attacker (*false positive*) and, thus, does not need to be fixed.

Similarly, if an SAST tool does not report security issues, this can have two reasons:

- The source code is secure (*true negative*)
- The source code has security vulnerability but due to limitations of the tool, the tool does not report a problem (*false negative*).

There are SAST tools available for most of the widely used programming language, e.g., FindBugs [8] that is able to analyzes Java byte code and, thus, can analyze various languages running on the Java Virtual Machine. There are also specializes techniques for Java programs (e.g., [83]), or C/C++ (e.g., [33]) as well as approaches that work on multiple languages (e.g., [25]). For a survey on static analysis methods, we refer the reader elsewhere [101, 26]. Moreover, we discuss further static analysis techniques in the context of a small case study in Section 6.

Besides the fact that SAST tools can be applied very early in the software development lifecycle as well as the fact that source code analysis can provide detailed fix recommendations, SAST has one additional advantages over most dynamic security testing techniques: SAST tools can analyze all control flows of a program. Therefore,

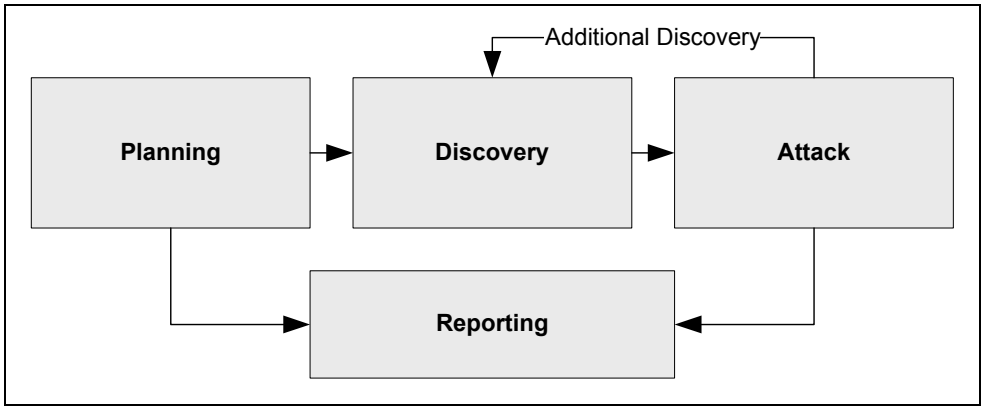

Figure 5: Phases of a penetration test [113]

SAST tools achieve, compared to dynamic test approaches, a significant higher coverage of the program under test and, thus, produce a significant lower false negative rate. Thus, SAST is a very effective method [112] for detecting programming related vulnerabilities early in the software development lifecycle.

5.3 Penetration Testing and Dynamic Analysis

In contrast to white-box security testing techniques (see Section 5.2), black-box security testing does not require access to the source code or other development artifacts of the application under test. Instead, the security test is conducted via interaction with the running software.

5.3.1 Penetration testing

A well known form of black-box security testing is *Penetration Testing*. In a penetration test, an application or system is tested from the outside in a setup that is comparable to an actual attack from a malicious third party. This means, in most settings the entity that is conducting the test has potentially only limited information about the system under test and is only able to interact with the system's public interfaces. Hence, a mandatory prerequisite for this approach is a (near) productive application, that is feature complete and sufficiently filled with data, so that all implemented workflows can be executed during the test. Penetration tests are commonly done for applications that are open for networked communication.

The NIST Technical Guide To Information Security Testing And Assessment [113] partitions the penetration testing process in four distinct phases (see Fig. 5):

1. *Planning:* No actual testing occurs in this phase. Instead, important side conditions and boundaries for the test are defined and documented. For instance, the relevant components of the applications that are subject of the test are determined and the nature/scope of the to be conducted tests and their level of invasiveness.
2. *Discovery:* This phase consists of a steps. First, all accessible external interfaces of the system under test are systematically discovered and enumerated. This

set of interfaces constitutes the system's initial *attack surface*. The second part of the discovery phase is vulnerability analysis, in which the applicable vulnerability classes that match the interface are identified (e.g., Cross-Site Scripting for HTTP services or SQL Injection for applications with database backend). In a commercial penetration test, this phase also includes the check if any of the found components is susceptible to publicly documented vulnerabilities which is contained in precompiled vulnerability databases.

3. *Attack:* Finally, the identified interfaces are tested through a series of attack attempts. In these attacks, the testers actively attempts to compromise the system via sending attack payloads. In case of success, the found security vulnerabilities are exploited in order to gain further information about the system, widen the access privileges of the tester and find further system components, which might expose additional interfaces. This expanded attack surface is fed back into the discovery phase, for further processing.
4. *Reporting:* The reporting phase occurs simultaneously with the other three phases of the penetration test and documents all findings along with their estimated severeness.

5.3.2 Vulnerability scanning

In general penetration tests are a combination of manual testing through security experts and the usage of *black-box vulnerability scanners*. Black-box web vulnerability scanners are a class of tools that can be used to identify security issues in applications through various techniques. The scanner queries the application's interfaces with a set of predefined attack payloads and analyses the application's responses for indicators if the attack was successful and, if this is not the case, hints how to alter the attack in the subsequent tries. Bau et al. [11] as well as Adam et al. [3] provide overviews of recent commercial and academic black-box vulnerability scanners.

5.3.3 Dynamic taint analysis

An important variant of black-box testing is an analysis technique called *taint analysis*. A significant portion of today's security vulnerabilities are *string-based code injection vulnerabilities* [71], which enable the attacker to inject syntactic content into dynamically executed programming statements, which — in the majority of all cases — leads to full compromise of the vulnerable execution context. Examples for such vulnerabilities include SQL Injection [52] and Cross-Site Scripting [50]. Such injection vulnerabilities can be regarded as information flow problems, in which unsanitized data paths from untrusted sources to security sensitive sinks have to be found. To achieve this, a well established approach is *(dynamic) data tainting*. Untrusted data is outfitted with *taint* information on runtime, which is only cleared, if the data passes a dedicated sanitization function. If data which still carries taint information reaches a security sensitive sink (e.g., an API that converts string data into executable code), the application can react appropriately, for instance through altering, auto-sanitization the data or completely stopping the corresponding process. If taint tracking is utilized in security testing, the main purpose is to notify the tester that insecure data flows, that likely lead to code injection, exist. Unlike static analysis, that also targets the identification of problematic data flows, dynamic taint analysis is conducted transparently while the

application under test is executed. For this, the execution environment, e.g., the language runtime, has to be made taint aware, so that the attached taint information of the untrusted data is maintained through the course of program execution, so that it can reliably be detected when tainted data ends up in security sensitive sinks.

5.3.4 Fuzzing

Fuzzing or *fuzz testing* is a dynamic testing technique that is based on the idea of feeding random data to a program "until it crashes." It was pioneered in the late 1980ies by Barton Miller at the University of Wisconsin [89]. Since then, fuzz testing has been proven to be an effective technique for finding vulnerabilities in software. While the first fuzz testing approaches where purely based on randomly generated test data (random fuzzing), advances in symbolic computation, model-based testing, as well as dynamic test case generation have lead to more advanced fuzzing techniques such as mutation-based fuzzing, generation-based fuzzing, or gray-box fuzzing.

Random fuzzing is the simplest and oldest fuzz testing technique: a stream of random input data is, in a black-box scenario, send to the program under test. The input data can, e.g., be send as command line options, events, or protocol packets. This type of fuzzing in, in particular, useful for test how a program reacts on large or invalid input data. While random fuzzing can find already severe vulnerabilities, modern fuzzers do have a detailed understanding of the input format that is expected by the program under test.

Mutation-based fuzzing is one type of fuzzing in which the fuzzer has some knowledge about the input format of the program under test: based on existing data samples, a mutation-based fuzzing tools generated new variants (mutants), based on a heuristics, that it uses for fuzzing. there are a wide range of mutation based fuzzing approaches available for different domains. We refer the interested reader elsewhere for details [108, 32].

Generation-based fuzzing used a model (of the input data or the vulnerabilities) for generating test data from this model or specification. Compared to pure random-based fuzzing, generation-based fuzzing achieves usually a higher coverage of the program under test, in particular if the expected input format is rather complex. Again, for details we refer the interested reader elsewhere [138, 144].

Advanced fuzzing techniques combine several of the previously mentioned approaches, e.g., use a combination of mutation-based and generation-based techniques as well as observe the program under test and use these observations for constructing new test data. This turns fuzzing into a gray-box testing technique that also utilizes symbolic computation that is usually understood as a technique used for static program analysis. Probably the first and also most successful application of gray-box fuzzing is SAGE from Microsoft [47, 48], which combines symbolic execution (a static source code analysis technique) and dynamic testing. This combination is today known as concolic testing and inspired several advanced security testing e.g., [12, 1], as well as functional test approaches.

5.4 Security Regression Testing

Due to ever changing surroundings, new business needs, new regulations, and new technologies, a software system must evolve and be maintained, or it becomes progressively

less satisfactory [80]. This makes it especially challenging to keep software systems permanently secure as changes either in the system itself or in its environment may cause new threats and vulnerabilities [37]. A combination of regression and security testing called security regression testing, which ensures that changes made to a system do not harm its security, are therefore of high significance and the interest in such approaches has steadily increased [36]. Regression testing techniques ensure that changes made to existing software do not cause unintended effects on unchanged parts and changed parts of the software behave as intended [82]. As requirements, design models, code or even the running system can be changed, regression testing techniques are orthogonal to the security testing techniques discussed in the previous sections.

Yoo and Harman [139] classify regression testing techniques into three categories: test suite minimization, test case prioritization and test case selection.

Test suite minimization seeks to reduce the size of a test suite by eliminating redundant test cases from the test suite.

Test case prioritization aims at ordering the execution of test cases in the regression test suite based on a criterion, for instance, on the basis of history, coverage, or requirements, which is expected to lead to the early detection of faults or the maximization of some other desirable properties.

Test case selection deals with the problem of selecting a subset of test cases that will be used to test changed parts of software. It requires to select a subset of the tests from the previous version, which are likely to detect faults, based on different strategies. Most reported regression testing techniques focus on this regression testing technique [139]. The usual strategy is to focus on the identification of modified parts of the SUT and to select test cases relevant to them. For instance, the *retest-all* technique is one naive type of regression test selection by re-executing all tests from the previous version on the new version of the system. It is often used in industry due to its simple and quick implementation. However, its capacity in terms of fault detection is limited [41]. Therefore, considerable amount of work is related to the development of effective and scalable selective techniques.

In the following, we discuss available security testing approaches according to the categories minimization, prioritization, and selection. The selected approaches are based on a systematic classification of security regression testing approaches by Felderer and Fourneret [36].

5.4.1 Test Suite Minimization

Test suite minimization seeks to reduce the size of a test suite by eliminating test cases from the test suite based on a given criterion. Current approaches on minimization [122, 55, 45] address vulnerabilities.

Toth et al. [122] propose an approach that applies automated security testing for detection of vulnerabilities by exploring application faults that may lead to known malware, such as viruses or worms. The approach considers only failed tests from the previous version revealing faults for re-run in a new system version after fault fixing.

He et al. [55] propose an approach for detecting and removing vulnerabilities for minor releases of web applications. In their approach, only strong-association links between pages from a previous version, optimized through iterations, are selected for exploration of web pages that contain vulnerabilities.

Finally, Garvin et al. [45] propose testing of self-adaptive system for already known failures. The authors avoid reproducing already known failures by considering only those tests for execution that exercise the known failures in previous versions.

5.4.2 Test Case Prioritization

Test case prioritization is concerned with right ordering of test cases that maximizes desirable properties, such as early fault detection. Also, current approaches to prioritization [59, 140, 129] address only vulnerabilities.

Huang et al. [59] propose a prioritization technique for security regression testing. Their technique gathers historical records, whose most effective execution order is determined by a genetic algorithm.

Yu et al. [140] propose fault-based prioritization of test cases, which directly utilizes the knowledge of their capability to detect faults.

Finally, Viennot et al. [129] propose a mutable record-replay system, which allows a recorded execution of an application to be replayed with a modified version of the application. Their execution is prioritized by defining a so called d-optimal mutable replay based on a cost function measuring the difference between the original execution and the mutable replay.

5.4.3 Test Case Selection

Test case selection approaches choose a subset or all test cases to test changed parts of software. As for classical regression testing [139], also for security regression testing most approaches fall into this category [36]. These approaches test both, security mechanisms and vulnerabilities. Several subset-based approaches [34, 75, 5, 58, 62] and retest-all approaches [128, 22, 78, 106] have been proposed.

Felderer et al. [34] provide a UML-based approach for regression testing of security requirements of type authentication, confidentiality, availability, authorization, and integrity. Tests, represented as sequence diagrams, are selected based on test requirements changes. Kassab et al. [75] propose an approach to improve regression testing based on non-functional requirements ontologies. Tests are selected based on change and impact analysis of non-functional requirements, such as security, safety, performance, or reliability. Each test linked to a changed or modified requirement is selected for regression testing. Anisetti et al. [5] propose an approach for providing test evidence for incremental certification of security requirements of services. This approach is based on change detection in the service test model, which will determine if new test cases need to be generated, or existing ones to be selected for re-execution on the evolved service. Huang et al. [58] address security regression testing of access control policies. This approach selects tests if they cover changed elements in the policy. Finally, Hwang et al. [62] propose three safe coverage-based selection techniques for testing evolution of security policies, each of which includes a sequence of rules to specify which subjects are permitted or denied to access which resources under which conditions. The three techniques are based on two coverage criteria, i.e., (1) coverage of changed rules in the policy, and (2) coverage of different program decisions for the evolved and the original policy.

Vetterling et al. [128] propose an approach for developing secure systems evaluated under the Common Criteria [67]. In their approach, tests covering security require-

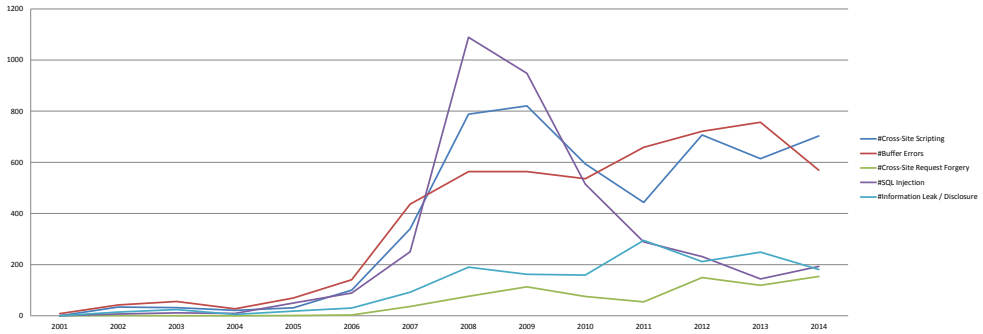

Figure 6: Number of entires in the Common Vulnerabilities and Exposures (CVE) index by category

ments are created manually and represented as sequence diagrams. In case of a change, new tests are written if necessary and then all tests are executed on the new version. Bruno et al. [22] propose an approach for testing security requirements of web service releases. The service user can periodically re-execute the test suite against the service in order to verify whether the service still exhibits the functional and non-functional requirements. Kongsli [78] proposes the use of so called misuse stories in agile methodologies (for instance, extreme programming) to express vulnerabilities related to features and/or functionality of the system in order to be able to perform security regression testing. The author suggests that test execution is ideal for agile methodologies and continuous integration. Finally, Qi et al. [106] propose an approach for regression fault-localization in web-applications. Based on two programs, a reference program and a modified program, as well as input failing on the modified program, their approach uses symbolic execution to automatically synthesize a new input that is very similar to the failing input and does not fail. On this basis, the potential cause(s) of failure are found by comparing control flow behavior of the passing and failing inputs and identifying code fragments where the control flows diverge.

6 Application of Security Testing Techniques

In this section, we make a concrete proposal on how to apply the security test techniques (and the tools implementing them) to a small case study: a business application using a three tiered architecture. We focus on security testing techniques that detect the most common vulnerability types that were disclosed in the Common Vulnerabilities and Exposures (CVE) index [91] over the period of the last 15 years (see Figure 6). This clearly shows that the vast majority of vulnerabilities, such as XSS, buffer overflows, are still caused by programming errors. Thus, we focus in this section on security testing techniques that allow to find these kind of vulnerabilities. For instance, we exclude techniques for ensuring the security of the underlying infrastructure such as the network configuration, as, e.g., discussed in [15, 20] as well as model-based testing techniques (as discussed in Section 5.1) that are in particular useful for finding logical security flaws. While a holistic security testing strategy makes use of all available security testing strategies, we recommend to concentrate efforts first on techniques for the most common vulnerabilities. Furthermore, we also do not explicitly discuss retesting after

changes of the system under test, which is addressed by suitable (security) regression testing approaches (as discussed in Section 5.4).

6.1 Selection Criteria for Security Testing Approaches

When selecting a specific security testing method and tool, many aspects need to be taken into account, e.g.:

- *Attack surface:* different security testing methods find different attack and vulnerability types. Many techniques are complementary so it is advisable to use multiple security testing methods together to efficiently detect a range of vulnerabilities as wide as possible.
- *Application type*: different security testing methods perform differently when applied to different application types. For example, a method that performs well against a mobile application may not be able to perform as well against three-tier client-server applications.
- *Performance and resource utilization:* different tools and methods require different computing power and different manual efforts.
- *Costs for licenses, maintenance and support:* to use security testing tools efficiently in a large enterprise, they need to be integrated into, e.g., bug-tracking or reporting solutions—often they provide their own server applications for this. Thus, buying a security testing tool is usually not a one-time effort—it requires regular maintenance and support.
- *Quality of results:* different tools that implement the same security testing technique provide a different quality level (e.g., in terms of fix recommendations or false positives rates).
- *Supported technologies:* security testing tools usually only support a limited number of technologies (e.g., programming languages, interfaces, or build systems). If these tools support multiple technologies, they do not necessary support all of them with the same quality. For example, a source analysis tool that supports Java and C might work well for Java but not as well for C.

In the following, we focus on the first two aspects: the attack surface and the application type. These two aspects are, from a security perspective the first ones to consider for selecting the best combinations of security testing approaches for a specific application type (product). In a subsequent step, the other factors need to be considered for selecting a specific tool that fits the needs of the actual development as well as the resource and time constraints.

6.2 A Three-tiered Business Application

In this chapter, we use a simple multi-tiered business application, e.g., for booking business travels, as a running example. This architecture is, on the first hand, very common for larger applications and, on the other hand, covers a wide variety of security testing challenges.

Let us assume that we want to plan the security testing activities for business application that is separated into three tiers:

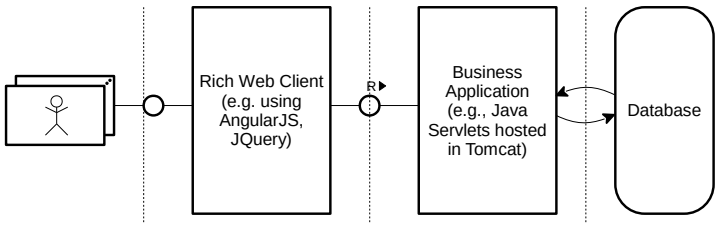

Figure 7: Architecture of Example Application

- *First tier:* A front-end that is implemented as rich-client using modern web development techniques, i.e., HTML5 and JavaScript, e.g., using frameworks such as AngularJS or JQuery.
- *Second tier:* A typical middle-tier implemented in Java (e.g., using Java Servlets hosted in an application server such as Apache Tomcat).
- *Third tier:* A third-party database that provides persistency for the business data that is processed in the second tier.

Figure 7 illustrates this example architecture where the dotted vertical lines mark the trust boundaries of the application.

6.2.1 The First Tier: Web Applications

Web applications are predominantly affected by code injection vulnerabilities, such as SQL Injection [52] or Cross-Site Scripting (XSS) [50]. This is shown in Figure 6: as it can be seen, besides buffer overflows (and similar memory corruption vulnerabilities), which in general do not occur in Web applications, injection vulnerabilities represent the vast majority of reported issues.

In consequence, security testing approaches for Web applications are primarily concerned with detection of such code injection vulnerabilities. In the remainder of this section, we utilize the field of Web applications as a case study in security testing and comprehensively list academic approaches in this area.

Code-Based Testing and Static Analysis. As introduced in Section 5.2, static analysis of source code is a powerful tool to detect vulnerabilities in source code. Static analysis allows to analyze the code of a given application without actually executing it. To detect injection vulnerabilities with such an approach, in most cases the static analysis attempts to approximate the data flows within the examined source code. This way, data flows from *sources* that can be controlled by the attacker (e.g., the incoming HTTP request) into security sensitive *sinks* (e.g, APIs that create the HTTP response of send SQL to the database).

Different mechanisms and techniques proposed in the program analysis literature can be applied to Web applications. The actual implementation of such an analysis can utilize different techniques such as model checking, data-flow analysis, or symbolic execution, typically depending on the desired precision and completeness. Three different properties are relevant for the analysis phase. First, the static analysis of the source code itself can be performed on different levels, either only within a given function (*intraprocedural analysis*) or the interaction of functions can be analyzed as well

(*interprocedural analysis*). Second, the execution context can be examined in detail or neglected, which then typically reduces the precision of the analysis. A third aspect of static analysis deals with the way how the flow of data or code is analyzed.

One of the first tools in this area was *WebSSARI* [60], a code analysis tool for PHP applications developed by Huang et al. based on a CQual-like type system [39, 40].

Pixy is a static taint analysis tool presented by Jovanovic et al. for automated identification of XSS vulnerabilities in PHP web applications [72, 73]. The tool performs an interprocedural, context-sensitive, and flow-sensitive data flow analysis to detect vulnerabilities in a given application. Combined with a precise alias analysis, the tool is capable of detecting a variety of vulnerabilities. In a similar paper, Livshits and Lam demonstrated how a context-sensitive pointer alias analysis together with string analysis can be realized for Java-based web applications [83].

Xie and Aiken introduced a static analysis algorithm based on so-called block and function summaries to detect vulnerabilities in PHP applications [137]. The approach performs the analysis in both an intraprocedural and an interprocedural way. Dahse and Holz refined this approach and demonstrated that a fine-grained analysis of built-in PHP features and their interaction significantly improves detection accuracy [29, 28].

Wasserman and Su implemented a static code analysis approach to detect XSS vulnerabilities caused by weak or absent input validation [134]. The authors combined work on taint-based information flow analysis with string analysis previously introduced by Minamide [90].

A tool for static analysis of JavaScript code was developed by Saxena et al. [110]. The tool named Kudzu employs symbolic execution to find client-side injection flaws in Web applications. Jin et al. employ static analysis[70] to find XSS vulnerabilities within HTML5-based mobile applications. Interestingly, they found new ways to conduct XSS attacks within such apps by abusing the special capabilities of mobile phones.

Further static analysis approaches to detect injection vulnerabilities have been proposed by Fu et al.[42], Wassermann and Su [133], and Halfond et al. [53]

Dynamic analysis and black-box testing. A complementary approach is dynamic analysis, in which a given application is executed with a particular set of inputs and the runtime behavior of the application is observed. Many of these approaches employ dynamic taint tracking and consist of two different components. A detection component to identify potentially vulnerable data flows and a validation component to eliminate false positives.

SecuBat [74] is a general purpose Web vulnerability scanner that also has XSS detection capabilities. To achieve its goals, SecuBat employs three different components: a crawling component, an attack component, and an analysis component. Whenever the crawling component discovers a suspicious feature, it passes the page under investigation to the attack component, which then scans this page for web forms. If a form is found, appropriate payloads are inserted into the fields of the form and submitted to the server. The response of the server is then interpreted by the analysis component. Snuck [30] is another dynamic testing tool. In a first step, Snuck uses a legitimate test input to dynamically determine a possible injection and the corresponding context via XPATH expressions. Based on the determined context, the tool chooses a set of predefined attack payloads and injects them into the application.

While SecuBat and Snuck focus on server-side security problems, FlashOver [2] focuses on a client-side problem. More specifically, FlashOver detects client-side reflected XSS in Adobe Flash applets. It does so by decompiling the source code and statically detecting suspicious situations. Then it constructs an attack payload and executes the exploit via dynamic testing.

Recently, Bau et al. [11] and and Doupe et al. [3] prepared comprehensive overviews on commercial and academic black-box vulnerability scanners and their underlying approaches.

Taint-tracking. As discussed previously, most Web-specific security vulnerabilities can predominantly be regarded as information flow problems, caused by unsanitized data paths from untrusted sources to security sensitive sinks. Hence, taint-tracking (see Section 5.3.3) is a well established method to detect injection vulnerabilities.

Dynamic taint tracking was initially introduced by Perl in 1989 [131] and since then has been adopted for numerous programming languages and frameworks [117]. Subsequent works describe finer grained approaches towards dynamic taint propagation. These techniques allow the tracking of untrusted input on the basis of single characters. For instance, Nguyen-Tuong et al [95] and Pietraszek and Vanden Berghe [100] proposed fine grained taint propagation for the PHP runtime to detect various classes of injection attacks, such as SQL Injection or XSS. Based on dynamic taint propagation, Su and Wassermann [119] describe an approach that utilizes specifically crafted grammars to deterministically identify SQL injection attempts.

The first taint tracking paper that aimed at automatically generating Cross-Site Scripting attacks was authored by Martin et al. [86]. The presented mechanism expects a program adhering to the Java Servlet specification and a taint-based vulnerability specification as an input, and generates a valid Cross-Site Scripting or SQL Injection attack payload with the help of dynamic taint tracking and model checking. While this approach requires a security analyst to manually write a vulnerability specification, Kieyzun et al. focus on the fully automatic generation of taint-based vulnerability payloads [76]. Furthermore, they extend the dynamic tainting to the database. Hence, as opposed to the first approach, this approach is able to also detect server-side persistent XSS vulnerabilities.

Besides server-side injection vulnerabilities, Web applications are also susceptible to injection problems on the client-side, i.e., Cross-Site Scripting problems caused by insecure JavaScript executed in the browser [77]. Through making the browser's JavaScript engine taint-aware, this vulnerability class can be detected during testing: Lekies et al. implemented a browser-based byte-level taint-tracking engine to detect reflected and persistent client-side XSS vulnerabilities [81]. By leveraging the taint-information, their approach is capable of generating a precise XSS payload matching the injection context. A similar approach was taken by FLAX [111], which also employs taint-tracking in the browser. Instead of using an exploit generation technique, FLAX utilizes a sink-aware fuzzing technique, which executes variations of a predefined list of context-specific attack vectors.

6.2.2 The Second Tier: Java-based Server Applications

Also the second tier, i.e., the Java-based business application, is predominantly affected by code injection vulnerabilities, such as SQL Injection [52] or Cross-Site Scripting [50].

Even if we secure the front-end against these attacks, we cannot rely on these protection as an attacker might circumvent the front-end by attacking the second tier directly (e.g., using WSDL-based, RESTful, or OData-based interfaced). Thus, we need to apply the same standards for secure development also to the second tier.

Code-Based Testing and Static Analysis. There are many static analysis approaches available for languages traditionally used for server systems, e.g., C/C++ or Java; both in the academic world as well as commercial offerings. Commercial tools are at least available since ten years [25, 88] and used widely since at least five years [14, 21]. Due to space reason, we only discuss a few selected works for Java: most notability, FindBugs [8] is a "ready-to-run" tool that finds a large number of potential security vulnerabilities in Java programs and, moreover, can be considered the first widely used static analysis tool for Java that included security checks. Already in 2005, Livshits presented a method based on taint-propagation for finding data-flow related vulnerabilities in Java programs [83]. Tripp et al. [123] improved this idea to make it applicable to large scale programs by using a optimized pointer analysis as prerequisite for building the call graph. An alternative to using a call-graph-based tainting analysis, there are also approaches, such as [92] that are based on slicing. Moreover, there are two different Open Source frameworks available that for building own static analysis tools for Java: Wala [130], which itself is written in Java, and Swaja [61], which is written in OCaml.

Penetration Testing and Dynamic Analysis. From an implementation perspective, the most important dynamic testing approach is – besides manual penetration testing – fuzzing. As we implement the second tier, we focus on gray-box testing approaches that combine white-box and black-box testing. The most well-known and industrially proven approach is SAGE [48], which is used by Microsoft. Besides SAGE, there are also other approaches that share the same basic concept: using instrumentation or static analyses of the implementation to improve the quality, efficiency, or effectiveness of dynamic security testing. For example, [12] uses a binary taint analysis to increase the fuzz testing coverage while [54] uses symbolic execution to achieve the same goal. Gray-box fuzzing is also successfully applied to commercial operating system [87].

When using a gray-box testing technique on a multi-tier architecture, one can decide to only test the second tier in isolation or to test all tiers "at once". In the latter case, only selected tiers might be subject to taint-tracking or symbolic execution, as already such a partial covering of the implementation with white-box techniques allows to improve the testing results. Moreover, even partial application of white-box techniques helps to diagnose the root cause of an vulnerability and, thus, usually helps to minimize the time required for fixing a vulnerability.

Until recently, thus gray-box testing approaches where not generally available in commercial security testing solutions. This has changed: first vendors are staring to integrate similar techniques in their tools, using the *Interactive Security Application Testing* (ISAT).

6.2.3 The Third Tier: Back-End Systems

In our example, we assume that the back-end system (i.e., the database) is supplied by a third-party vendor. Therefore, we only have access to the systems as black-box. Still, we need to security test it as we are implementing stored procedures (SQL) on top of it as well as need to configure its security (e.g., to ensure proper access control and a secure and trustworthy communication to the second tier). Moreover, we might want to assess the overall implementation-level security of the externally developed product.

Security Testing the Implementation. For increasing our confidence in the security of the database, a third-party component, itself, we can apply manual penetration testing as well as fuzzing for checking for buffer overflows in the interfaces that are exposed via the network. As we assume that we only have access to the binary, we can only use black-box fuzzers such as presented by Woo et al. [136] or gray-box fuzzers that are based on binary analysis, e.g., [79].

Moreover, all code that we develop on top of the core database, i.e., stored procedures written in an SQL-dialect such as PL/SQL should be analyzed for injection attacks using dedicated static analysis approaches such as [24] (also many commercially available static analysis tools support the analysis of SQL dialects). Moreover, we can apply several dynamic security testing approaches that specialize on testing SQL Injections: Appelt et al. [6] present a mutation based testing approach for finding SQL Injections while Wang et al. [132] use a model-based fuzzing approach. For an experience report on using fuzzers for finding SQL Injections see [44].

Security Testing the Configuration. Finally, even if all software components are implemented securely, we still need to ensure that the actual configuration used during operations is also secure. While we touch this topic only very briefly, we want to emphasize that it is important to test the secure configuration of the communication channels, the access control of all tiers, as well as to keep the systems up to date by patching known vulnerabilities. Testing the security of the communication channels includes approaches that check the correct validation of SSL/TLS certificates (e.g., Frankencerts [18]) as well as protocol fuzzers such as SNOOZE [10] or SECFUZZ [124]. For testing the correct access control, various model based approaches (e.g., [13, 19, 85]) haven been applied to case studies of different size. Finally, tools like Nessus [109] that rather easily allow to scan networks for applications with known vulnerabilities and, thus, applications that need to be updated or patched.

7 Summary

In this chapter, we provided an overview of recent security testing techniques and their practical application in context of a three-tiered business application. For this purpose, we first summarized the required background on software testing and security engineering. Testing consists of static and dynamic lifecycle activities concerned with evaluation of software products and related artifacts. It can be performed on the component, integration, and system level. With regard to accessibility of test design artifacts white-box testing (i.e., deriving test cases based on design and code information) as well as black-box testing (i.e., relying only on input/output behavior

of software) can be distinguished. Security testing validates software system requirements related to security properties of assets that include confidentiality, integrity, availability, authentication, authorization, and non-repudiation. Security requirements can be positive and functional, explicitly defining the expected security functionality of a security mechanism, or negative and non-functional, specifying what the application should not do. Due to the negative nature of many security requirements and the resulting broad range of subordinate requirements, it is essential to take testing into account in all phases of the secure software development lifecycle (i.e., analysis, design, development, deployment as well as maintenance) and to combine different security testing techniques.

For a detailed discussion of security testing techniques in this chapter, we therefore classified them according to their test basis within the secure software development lifecycle into four different types: (1) *model-based security testing* is grounded on requirements and design models created during the analysis and design phase, (2) *code-based testing and static analysis* on source and byte code created during development, (3) *penetration testing and dynamic analysis* on running systems, either in a test or production environment, as well as (4) *security regression testing* performed during maintenance. With regard to model-based security testing, we considered testing based on architectural and functional models, threat, fault and risk models, as well as weakness and vulnerability models. Concerning, code-based testing and static analysis we took manual code reviews as well as static application security testing into account. With regard to penetration testing and dynamic analysis, we considered penetration testing itself, vulnerability scanning, dynamic taint analysis, as well as fuzzing. Concerning security regression testing, we discussed approaches to test suite minimization, test case prioritization, and test case selection. To show how the discussed security testing techniques could be practically applied, we discuss their usage for a three-tiered business application based on a web client, an application server, as well as a database backend.

Overall, this chapter provided a broad overview of recent security testing techniques. It fulfills the growing need for information on security testing techniques to enable their effective and efficient application. Along these lines, this chapter is of value both for researchers to evaluate and refine existing security testing techniques as well as for practitioners to apply and disseminate them.

Acknowledgements

The work was supported in part by the research projects QE LaB - Living Models for Open Systems (FFG 822740) and MOBSTECO (FWF P 26194-N15).

References

[1] H. J. Abdelnur, R. State, and O. Festor. Advanced fuzzing in the voip space. *Journal in Computer Virology*, 6(1):57–64, 2010.

[2] S. V. Acker, N. Nikiforakis, L. Desmet, W. Joosen, and F. Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. May 2012.

[3] Adam Doupe and Marco Cova and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA 2010*, 2010.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008.

[5] M. Anisetti, C. Ardagna, and E. Damiani. A low-cost security certification scheme for evolving services. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 122–129, June 2012.

[6] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269, New York, NY, USA, 2014. ACM.

[7] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *Security & Privacy, IEEE*, 3(1):84–87, 2005.

[8] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Experiences using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008. Special issue on software development tools, September/October (25:5).

[9] R. Bachmann and A. D. Brucker. Developing secure software: A holistic approach to security testing. *Datenschutz und Datensicherheit (DuD)*, 38(4):257–261, apr 2014.

[10] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer, and G. Vigna. SNOOZE: toward a stateful network protocol fuzzer. In S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2006.

[11] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[12] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A taint based approach for smart fuzzing. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012*, pages 818–825. IEEE, 2012.

[13] A. Bertolino, Y. L. Traon, F. Lonetti, E. Marchetti, and T. Mouelhi. Coverage-based test cases selection for XACML policies. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*, pages 12–21. IEEE Computer Society, 2014.

[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[15] N. Bjørner and K. Jayaraman. Checking cloud contracts in microsoft azure. In R. Natarajan, G. Barua, and M. R. Patra, editors, *Distributed Computing and Internet Technology - 11th International Conference, ICDCIT 2015, Bhubaneswar,*

India, February 5-8, 2015. Proceedings, volume 8956 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2015.

[16] J. Botella, B. Legeard, F. Peureux, and A. Vernotte. Risk-based vulnerability testing using security test patterns. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 337–352. Springer, 2014.

[17] P. Bourque and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge Version 3.0 SWEBOK.* IEEE, 2014. http://www.computer.org/web/swebok.

[18] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 114–129, Washington, DC, USA, 2014. IEEE Computer Society.

[19] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. In *ACM symposium on access control models and technologies (SACMAT)*, pages 133–142, New York, NY, USA, 2011. ACM Press.

[20] A. D. Brucker, L. Brügger, and B. Wolff. Formal firewall conformance testing: An application of test and proof techniques. *Software Testing, Verification & Reliability (STVR)*, 25(1):34–71, 2015.

[21] A. D. Brucker and U. Sodan. Deploying static application security testing on a large scale. In S. Katzenbeisser, V. Lotz, and E. Weippl, editors, *GI Sicherheit 2014*, volume 228 of *Lecture Notes in Informatics*, pages 91–101, GI, mar 2014. GI.

[22] M. Bruno, G. Canfora, M. Penta, G. Esposito, and V. Mazza. Using test cases as contract to ensure service compliance across releases. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 87–100. 2005.

[23] M. Büchler, J. Oudinet, and A. Pretschner. Semi-automatic security testing of web applications from a secure model. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 253–262. IEEE, 2012.

[24] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, SEM '05, pages 106–113, New York, NY, USA, 2005. ACM.

[25] B. Chess and G. McGraw. Static analysis for security. *Security Privacy, IEEE*, 2(6):76–79, nov.-dec. 2004.

[26] B. Chess and J. West. *Secure programming with static analysis.* Addison-Wesley Professional, first edition, 2007.

[27] Committee on National Security Systems. 4009, "National Information Assurance Glossary". Technical report, Committee on National Security Systems, 2010.

[28] J. Dahse and T. Holz. Simulation of built-in php features for precise static code analysis. 2014.

[29] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. 2014.

[30] F. d'Amore and M. Gentile. Automatic and context-aware cross-site scripting filter evasion. *Department of Computer, Control, and Management Engineering Antonio Ruberti Technical Reports*, 1(4), 2012.

[31] A. C. Dias-Neto and G. H. Travassos. A picture from the model-based testing area: Concepts, techniques, and challenges. *Advances in Computers*, 80:45–120, 2010.

[32] F. Duchene, S. Rawat, J. Richier, and R. Groz. Kameleonfuzz: evolutionary fuzzing for black-box XSS detection. In E. Bertino, R. S. Sandhu, and J. Park, editors, *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, pages 37–48. ACM, 2014.

[33] D. Evans. Static detection of dynamic memory errors. *SIGPLAN Not.*, 31:44–53, May 1996.

[34] M. Felderer, B. Agreiter, and R. Breu. Evolution of security requirements tests for service-centric systems. In *Engineering Secure Software and Systems: Third International Symposium, ESSoS 2011*, pages 181–194. Springer, 2011.

[35] M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. In *The Third International Conference on Advances in System Testing and Validation Lifecycle(VALID 2011)*, pages 109–114, 2011.

[36] M. Felderer and E. Fourneret. A systematic classification of security regression testing approaches. *International Journal on Software Tools for Technology Transfer*, pages 1–15, 2015.

[37] M. Felderer, B. Katt, P. Kalb, J. Jürjens, M. Ochoa, F. Paci, L. M. S. Tran, T. T. Tun, K. Yskout, R. Scandariato, F. Piessens, D. Vanoverberghe, E. Fourneret, M. Gander, B. Solhaug, and R. Breu. Evolution of security engineering artifacts: A state of the art survey. *International Journal of Secure Software Engineering*, 5(4):48–97, 2014.

[38] M. Felderer and I. Schieferdecker. A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer*, 16(5):559–568, 2014.

[39] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. *SIGPLAN Not.*, 34(5), May 1999.

[40] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *SIGPLAN Not.*, 37(5), May 2002.

[41] E. Fourneret, J. Cantenot, F. Bouquet, B. Legeard, and J. Botella. Setgam: Generalized technique for regression testing based on uml/ocl models. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, pages 147–156, June 2014.

[42] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Computer Software*

and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, volume 1, pages 87–96. IEEE, 2007.

[43] M. Gallaher and B. Kropp. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-03, National Institute of Standards & Technology, May 2002.

[44] R. Garcia. Case study: Experiences on sql language fuzz testing. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 3:1–3:6, New York, NY, USA, 2009. ACM.

[45] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems*, ASAS '11, pages 24–33. ACM, 2011.

[46] P. Gerrard and N. Thompson. *Risk-based e-business testing*. Artech House Publishers, 2002.

[47] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[48] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[49] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

[50] J. Grossman, R. Hansen, P. Petkov, and A. Rager. *Cross Site Scripting Attacks: XSS Exploits and Defense*. Syngress, 2007.

[51] J. Grossmann, M. Schneider, J. Viehmann, and M.-F. Wendland. Combining risk analysis and security testing. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 322–336. Springer, 2014.

[52] W. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, pages 65–81. IEEE, 2006.

[53] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

[54] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 49–64, Berkeley, CA, USA, 2013. USENIX Association.

[55] T. He, X. Jing, L. Kunmei, and Z. Ying. Research on strong-association rule based web application vulnerability detection. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 237–241, Aug 2009.

[56] P. Herzog. *The Open Source Security Testing Methodology Manual 3*, 2010. `http://www.isecom.org/research/osstmm.html` [accessed: April 11, 2015].

[57] M. Howard and S. Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.

[58] C. Huang, J. Sun, X. Wang, and Y. Si. Selective regression test for access control system employing rbac. In J. Park, H.-H. Chen, M. Atiquzzaman, C. Lee, T.-h. Kim, and S.-S. Yeo, editors, *Advances in Information Security and Assurance*, volume 5576 of *Lecture Notes in Computer Science*, pages 70–79. 2009.

[59] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85(3):626 – 637, 2012. Novel approaches in the design and implementation of systems/software architecture.

[60] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *International Conference on the World Wide Web (WWW)*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[61] L. Hubert, N. Barré, F. Besson, D. Demange, T. P. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static analysis workshop for Java. In B. Beckert and C. Marché, editors, *FoVeOOS*, volume 6528 of *Lecture Notes in Computer Science*, pages 92–106, 2010.

[62] J. Hwang, T. Xie, D. El Kateb, T. Mouelhi, and Y. Le Traon. Selection of regression system tests for security policy evolution. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 266–269. ACM, 2012.

[63] IEEE. IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*, 2008.

[64] ISO. ISO/IEC/IEEE 29119 Software Testing, 2013. available at `http://www.softwaretestingstandard.org/` [accessed: April 7, 2015].

[65] ISO/IEC. *Information technology – open systems interconnection – conformance testing methodology and framework*, 1994. International ISO/IEC multi–part standard No. 9646.

[66] ISO/IEC. *ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model*, 2001.

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

[68] ISO/IEC. *ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, 2011.

[69] ISTQB. Standard glossary of terms used in software testing. version 2.2. Technical report, ISTQB, 2012.

[70] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. 2014.

[71] M. Johns. *Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting*. PhD thesis, University of Passau, 2009.

[72] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[73] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Workshop on Programming languages and analysis for security*, PLAS '06, pages 27–36, New York, NY, USA, 2006. ACM.

[74] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. WWW '06, pages 247–256, New York, NY, USA, 2006. ACM.

[75] M. Kassab, O. Ormandjieva, and M. Daneva. Relational-model based change management for non-functional requirements: Approach and experiment. In *Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on*, pages 1–9, May 2011.

[76] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. ICSE '09, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[77] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. [online], `http://www.webappsec.org/projects/articles/071105.shtml`, (05/05/07), Sebtember 2005.

[78] V. Kongsli. Towards agile security in web applications. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 805–808, New York, NY, USA, 2006. ACM.

[79] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari. A smart fuzzer for x86 executables. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007. Third International Workshop on*, pages 7–7, May 2007.

[80] M. Lehman. Software's future: Managing evolution. *IEEE software*, 15(1):40–44, 1998.

[81] S. Lekies, B. Stock, and M. Johns. 25 million flows later-large-scale detection of dom-based xss. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[82] H. K. N. Leung and L. White. Insights into regression testing (software testing). In *Proceedings Conference on Software Maintenance 1989*, pages 60–69. IEEE, 1989.

[83] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[84] M. S. Lund, B. Solhaug, and K. Stolen. *Model-driven Risk Analysis*. Springer, 2011.

[85] E. Martin. Testing and analysis of access control policies. pages 75–76, may. 2007.

[86] M. Martin and M. S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. SEC'08, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[87] S. B. Mazzone, M. Pagnozzi, A. Fattori, A. Reina, A. Lanzi, and D. Bruschi. Improving mac os x security through gray box fuzzing technique. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

[88] G. McGraw. *Software Security: Building Security In.* Addison-Wesley Professional, 2006.

[89] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[90] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *International Conference on the World Wide Web (WWW)*, 2005.

[91] MITRE. *Common Vulnerabilities and Exposures.*

[92] B. Monate and J. Signoles. Slicing for security of code. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *TRUST*, volume 4968 of *Lecture Notes in Computer Science*, pages 133–142, Heidelberg, 2008. Springer-Verlag.

[93] L. Morell. A theory of fault-based testing. *Software Engineering, IEEE Transactions on*, 16(8):844–857, Aug 1990.

[94] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 537–552, Berlin, Heidelberg, 2008. Springer-Verlag.

[95] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, May 2005.

[96] NIST. *The economic Impacts of inadequate Infrastructure for Software Testing*, 2002. available at `www.nist.gov/director/planning/upload/report02-3.pdf` [accessed: April 7, 2015].

[97] OWASP. OpepSAMM. `http://www.opensamm.org/`. Accessed: 2015-03-30.

[98] OWASP Foundation. OWASP Code Review Guide v1.1. `https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project`. Accessed: 2015-03-11.

[99] OWASP Foundation. OWASP Testing Guide v4. `https://www.owasp.org/index.php/OWASP_Testing_Project`. Accessed: 2015-03-11.

[100] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[101] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[102] B. Potter and G. McGraw. Software Security Testing. *IEEE Security & Privacy*, 2004.

[103] B. Potter and G. McGraw. Software security testing. *Security & Privacy, IEEE*, 2(5):81–85, 2004.

[104] A. Pretschner. Defect-Based Testing. In *Dependable Software Systems Engineering*. IOS Press, 2015. to appear.

[105] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar. A generic fault model for quality assurance. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 87–103. Springer Berlin Heidelberg, 2013.

[106] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19:1–19:29, July 2012.

[107] J. Radatz, A. Geraci, and F. Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990(121990), 1990.

[108] S. Rawat and L. Mounier. Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results. In *Fourth International IEEE Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 531–533. IEEE Computer Society, 2011.

[109] R. Rogers and R. Rogers. *Nessus Network Auditing*. Syngress Publishing, 2 edition, 2008.

[110] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[111] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. The Internet Society, 2010.

[112] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: a controlled experiment. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering,*, pages 1–10. IEEE, Nov. 2013.

[113] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh. Technical Guide to Information Security Testing and Assessment. Special Publication 800-115, National Institute of Standards and Technology (NIST), 2008.

[114] I. Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, 2012.

[115] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. *Proceedings 7th Workshop on Model-Based Testing*, 2012.

[116] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. In *Proceedings 7th Workshop on Model-Based Testing*, 2012.

[117] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[118] H. Shahriar and M. Zulkernine. Automatic testing of program security vulnerabilities. In *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 2, pages 550–555. IEEE, 2009.

[119] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of POPL'06*, January 2006.

[120] H. H. Thompson. Why security testing is hard. *IEEE Security & Privacy*, 1(4):83–86, 2003.

[121] G. Tian-yang, S. Yin-sheng, and F. You-yuan. Research on software security testing. *World Academy of Science, Engineering and Technology Issure*, 69:647–651, 2010.

[122] G. Tóth, G. Kőszegi, and Z. Hornák. Case study: Automated security testing on the trusted computing platform. In *Proceedings of the 1st European Workshop on System Security*, EUROSEC '08, pages 35–39. ACM, 2008.

[123] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. *SIGPLAN Not.*, 44:87–97, June 2009.

[124] P. Tsankov, M. T. Dashti, and D. A. Basin. SECFUZZ: fuzz-testing security protocols. In *7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2-3, 2012*, pages 1–7. IEEE, 2012.

[125] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[126] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab*, 22(2):297312, 2012.

[127] D. Verdon and G. McGraw. Risk analysis in software design. *Security & Privacy, IEEE*, 2(4):79–84, 2004.

[128] M. Vetterling, G. Wimmel, and A. Wisspeintner. Secure systems development based on the common criteria: The palme project. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 129–138. ACM, 2002.

[129] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 127–138. ACM, 2013.

[130] WALA. T. j. Watson Libraries for Analysis.

[131] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reilly, 3rd edition, July 2000.

[132] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye. A model-based fuzzing approach for dbms. *2013 8th International Conference on Communications and Networking in China (CHINACOM)*, 0:426–431, 2013.

[133] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of Programming Language Design and Implementation (PLDI'07)*, San Diego, CA, June 10-13 2007.

[134] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[135] M.-F. Wendland, M. Kranz, and I. Schieferdecker. A systematic approach to risk-based testing using risk-annotated requirements models. In *ICSEA 2012,*

The Seventh International Conference on Software Engineering Advances, pages 636–642, 2012.

[136] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 511–522, New York, NY, USA, 2013. ACM.

[137] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pages 179–192, 2006.

[138] D. Yang, Y. Zhang, and Q. Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In G. Min, Y. Wu, L. C. Liu, X. Jin, S. A. Jarvis, and A. Y. Al-Dubai, editors, *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 1070–1076. IEEE Computer Society, 2012.

[139] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 1(1):121–141, 2010.

[140] Y. T. Yu and M. F. Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2):179 – 202, 2012.

[141] J. Zander, I. Schieferdecker, and P. J. Mosterman. *Model-based testing for embedded systems*, volume 13. CRC Press, 2012.

[142] P. Zech, M. Felderer, and R. Breu. Security risk analysis by logic programming. In *Risk Assessment and Risk-Driven Testing*, pages 38–48. Springer, 2014.

[143] P. Zech, M. Felderer, B. Katt, and R. Breu. Security test generation by answer set programming. In *Software Security and Reliability, 2014 Eighth International Conference on*, pages 88–97. IEEE, 2014.

[144] J. Zhao, Y. Wen, and G. Zhao. H-fuzzing: A new heuristic method for fuzzing data generation. In E. R. Altman and W. Shi, editors, *Network and Parallel Computing - 8th IFIP International Conference, NPC 2011, Changsha, China, October 21-23, 2011. Proceedings*, volume 6985 of *Lecture Notes in Computer Science*, pages 32–43. Springer, 2011.

[145] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.

About the Authors

Michael Felderer is a senior researcher and project manager within the Quality Engineering research group at the Institute of Computer Science at the University of Innsbruck, Austria. He holds a Ph.D. in computer science. His research interests include software and security testing, empirical software and security engineering, model engineering, risk management, software processes and industry-academia collaboration. Michael Felderer has co-authored more than 70 journal, conference and workshop papers. He works in close cooperation with industry and also transfers his research results into practice as a consultant and speaker on industrial conferences.

Matthias Büchler is a Ph.D student at the Technische Universität München. He holds a master's degree in computer science (Information Security) from the Swiss Federal Institute of Technology Zurich (ETHZ). His research interests include information security, security modeling, security engineering, security testing, domain specific languages, and usage control.

Martin Johns is a research expert in the Product Security Research unit within SAP SE, where he leads the Web application security team. Furthermore, he serves on the board of the German OWASP chapter. Before joining SAP, Martin studied Mathematics and Computer Science at the Universities of Hamburg, Santa Cruz (CA), and Passau. During the 1990ties and the early years of the new millennium he earned his living as a software engineer in German companies (including Infoseek Germany, and TC Trustcenter). He holds a Diploma in Computer Science from University of Hamburg and a Doctorate from the University of Passau.

Achim D. Brucker is a research expert (architect), security testing strategist, and project lead in the Security Enablement Team of SAP SE. He received his master's degree in computer science from University Freiburg, Germany and his Ph.D. from ETH Zurich, Switzerland. He is responsible for the Security Testing Strategy at SAP. His research interests include information security, software engineering, security engineering, and formal methods. In particular, he is interested in tools and methods for modeling, building and validating secure and reliable systems. He also participates in the OCL standardization process of the OMG.

Ruth Breu is head of the Institute of Computer Science at the University of Innsbruck, leading the research group Quality Engineering and the competence center QE LaB. She has longstanding experience in the areas of security engineering, requirements engineering, enterprise architecture management and model engineering, both with academic and industrial background. Ruth is co-author of three monographs and more than 150 scientific publications and serves the scientific community in a variety of functions (e.g. Board Member of FWF, the Austrian Science Fund, Member of the

NIS Platform of the European Commission).

Alexander Pretschner holds the chair of Software Engineering at Technische Universität München. Research interests include software quality, testing, and information security. Master's degrees in computer science from RWTH Aachen and The University of Kansas; PhD degree from Technische Universität München. Prior appointments include a full professorship at Karlsruhe Institute of Technology, an adjunct associate professorship at Kaiserslautern University of Technolgoy, a group manager's position at the Fraunhofer Institute of Experimental Software Engineering in Kaiserslautern, a senior scientist's position at ETH Zurich, and visiting professorships at the Universities of Rennes, Trento, and Innsbruck.