Ruediger Bachmann and Achim D. Brucker

Developing Secure Software

A Holistic Approach to Security Testing

Building secure software requires a well-selected combination of security testing techniques during the whole software development lifecycle.

Security vulnerabilities are a serious threat to software vendors and their customers: they can result in both monetary loss as well as loss of reputation. Thus, implementing a rigid secure software development lifecycle is a competitive advantage for a software vendor.
A *holistic security testing approach* must cover the whole software development lifecycle across all software products and all security threats.

In this article, we discuss a holistic security testing approach that was developed at SAP, a large vendor of enterprise software.

1. Security Testing – An Overview

Most security vulnerabilities are caused by one of the following four reasons [1, 2]:

- bad programming patterns such as missing checks of user-influenced data that can cause, e.g., in SQL injections vulnerabilities,
- misconfiguration of security infrastructures, e.g., too permissible access control or weak cryptographic configurations,
- functional bugs in security infrastructures, e.g., access control enforcement infrastructures that inherently do not restrict system access,
- logical flaws in the implemented processes, e.g., resulting in an application allowing customers to order goods without paying.

The vast majority of successful attacks against IT applications do not attack core security primitives such as cryptographic algorithms. Attackers much more often exploit bad programming or misconfigurations [1, 2]. Thus, in this article, we focus on security testing techniques that help to detected vulnerabilities caused by programming errors and misconfiguration.

1.1 A Categorization

From a high-level perspective, (security) testing techniques are often classified as follows:

- *Black-box-testing vs. white-box-testing:* In *black-box-testing*, the tested system is used as a black-box, i.e., no internal details of the system implementation are used. In contrast, *white-box-testing* takes the internal system details (e.g., the source code) into account.
- *Dynamic testing vs. static testing:* Traditionally, testing is understood as a *dynamic testing*, i.e., the system under test is executed and its behaviour is observed. In contrast, *static testing techniques* analyse a system without executing the system under test.
- *Manual testing vs. automated testing:* In manual testing the test scenario is guided by a human while in automated testing the test scenario is executed by a specialized application.

As Figure 1 illustrates, these categories can be combined, e.g., *static code analysis* can be classified as automated, white-box, and static.

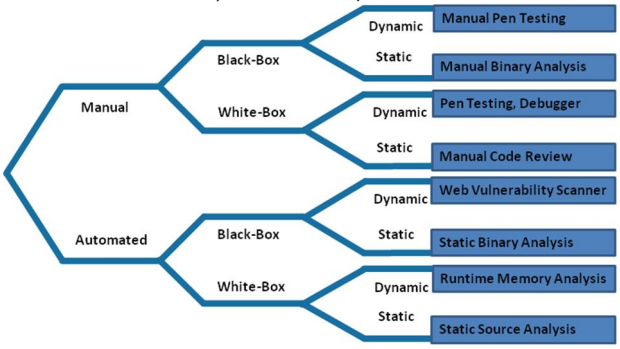

Figure 1: Security Testing Techniques

While these categories are important to differentiate security testing techniques, they are not sufficient to select the most appropriate security testing strategy for a specific application.

1.2 Selection Criteria

When selecting a security testing method or tool, we need to consider many aspects, e.g.:

- *Attack surface:* different security testing methods find different vulnerability types.
- *Application type:* different security testing methods behave differently when applied to different application types.
- *Quality of results and usability:* security testing techniques and tools differ in usability (e.g., fix recommendations) and quality (e.g., false positives rate).
- *Supported technologies:* security testing tools usually only support a limited number of technologies (e.g., programming languages) and if a

tool supports multiple technologies, it does not necessary support all of them equally well.
- *Performance and resource utilization:* different tools and methods require different computing power or different manual efforts.
- *Costs for licenses, maintenance and support:* besides direct costs, this includes, e.g., effort for integrating bug-trackers or reporting tools.

In this paper, we will focus on the first two aspects which, from a security perspective, are the most important ones.

2 Security Testing Methods

It is commonly accepted that fixing bugs and security vulnerabilities late in the software development is usually more costly compared to fixing them as early as possible [3]. Therefore, security testing techniques should be applied as early as possible in a secure software development lifecycle (see, e.g., [4]) and not, as an afterthought, during (or shortly before) shipment.

2.1 During Planning and Design

Strictly speaking, the security review of the architecture and threat modelling are not security testing methods. Still, we will discuss them as they are an important prerequisite for subsequent security testing efforts, respectively, the selection of security testing techniques:

- *Architecture Security Reviews*: A manual review of the product architecture to ensure that it fulfils the necessary security requirements.
  Prerequisites: architectural model.
  Benefit: detecting architectural violations of the security standard.
- *Threat modelling* is a structured manual analysis of an application specific business case or usage scenario. This analysis is guided by a set of precompiled security threats.
  Prerequisites: business case or usage scenario
  Benefits: identification of threats their impact and potential countermeasures that are specific to the development of the software product.

These methods help to identify the attack surface and, thus, the most critical components. This allows focusing the security testing activities and, thus, the up-front investment in, e.g., threat modelling, will save more effort in later on.

2.2 During Application Development

In development stages in which an application is not yet executable in a test environment, the following techniques are already applicable:

- *Static Source Code Analysis (SAST) and Manual Code Review*: Analysis of the application source code for finding vulnerabilities without actually executing the application.
  Prerequisites: Application source code.
  Benefits: Detection of insecure programming, outdated libraries, and misconfigurations.
- *Static Binary Code Analysis and Manual Binary Review*: Analysis of the compiled application (binary) for finding vulnerabilities without actually executing the application. In general, this is similar to the source code analysis but is not as precise and fix recommendation cannot be provided.

At SAP, static source code analysis is mandatory requirement for all products [5].

2.3 Executable in a Test Environment

At later stages of the development, when the software can actually be executed, further security testing approaches can be applied, e.g.:

- *Manual or automated Penetration Testing* simulates an attacker sending data to the application and observes its behaviour.
  Benefits: Identification of a wide range of vulnerabilities in a deployed application.
- *Automated Vulnerability Scanners* test an application for the use of system components or configurations that are known to be insecure. For this, pre-defined attack patterns are executed as well as system fingerprints are analysed.
  Benefits: Detection of well-known vulnerabilities, i.e., detection of outdated frameworks and misconfigurations.
- *Fuzz Testing* tools send random data, usually in larger chunks than expected by the application, to the input channels of an application to provoke a crashing of the application.
  Benefits: Detection of application crashes (e.g., caused by buffer overflows) that might be security critical.

All these techniques require a deployed and configured application on an *isolated* test system, including back-ends or external services.

While dynamic techniques usually do not achieve a similar high coverage of the analysed application as static approaches, they are particularly well suited for detecting vulnerabilities that involve data flows across system (or technology) boundaries.

2.4 System Operation and Maintenance

During the operation of an application, the security testing techniques discussed in the last section can be applied to ensure that the system configuration is still secure and that assumptions (e.g., a virus protection shall be installed or a correct authorization concept is implemented) are not violated accidentally. Additionally, passive security testing techniques that monitor system behaviour or analyse systems logs (e.g., monitoring system, intrusion detection systems) are generally recommended.

From a software maintenance perspective, the security testing of patches is particularly important: patches need to be security tested thoroughly (i.e., against all possible attacks and all system configurations the patch can be applied) to ensure that a customer that fix bugs in their system are not, accidentally, exposed to new vulnerabilities.

3 Application Types

As a thorough security analysis of an application requires the analysis in the context of the system landscape the application is used in, we need to have a closer look on the different applications types. The product portfolio form a ranges software vendor, such as SAP, often ranges from rather small applications (e.g., a mobile App) to complex multi-tiered client-server applications. Figure 2 illustrates the most common applications types that we will discuss in the following in more detail.

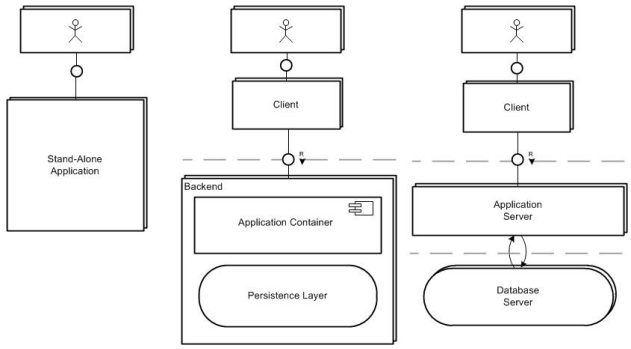

Figure 2: Stand-Alone, 2-Tier-, 3-Tier- Architecture

3.1 Client and Stand-alone Applications

The emphasis lies on the following subtypes: Mobile vs. desktop and native vs. non-native. Based on the structure (e.g., native implementation, large JavaScript parts, large HTML parts, browser plugin-based) different test recommendations can be derived.

3.2 Mobile Applications

Mobile applications and infrastructures have particular characteristics like local storage on a mobile client, password entry, location based data (GPS), different usability aspects, device management. Moreover, mobiles usually contain a lot of privacy relevant data both from personal as well as business life.

3.3 Application Servers

Application servers are one layer of multi-tier architectures. Pure application servers without any applications are hardly developed and shipped. Tomcat for instance always contains management and administration applications.

3.4 Storage

Storage (e.g., database servers) provides persistence for data which is stored in a certain form (e.g., as key-value pairs or as tables in a relational database). The data can be accessed and manipulated via well-defined interfaces.

3.5 Three-tiered Architecture

This is one representative of a multi-layer architecture. This document only focusses on 3 layers since it is sufficient to cover all necessary security testing aspects.

3.6 Two-tiered Architecture

From a client perspective there is no difference between two-, three- or multi-tier. Two-tier applications – compared to three tiers – bring database and application server closer together.

3.7 Cloud Applications

From a technical and from a security testing perspective similar structures as in multi-tier applications can be found. Principally a Cloud application should implement a strict tenant architectural concept, so that all data and behaviours performed by one tenant should be isolated from others. Cloud applications may also be used as a single tier in wider application deployments; for example to act as a consumable platform to allow services and applications to be written above it, or as a consumable infrastructure to allow platforms to be developed above it.

4 Best Practices

In this section we briefly discuss best practices, i.e., which combination of security testing techniques should be considered for which application types.

4.1 Client and Standalone Applications

Nowadays, pure standalone applications are seldom. Most applications allow access to the network. Even if this is not the case, an attacker can use networking features of the operating system, e.g., by providing a link to a malicious image which is automatically opened by an local image viewer I the user clicks in that link. Thus, while *automated static analysis* provides already a very good coverage of security issues for clients and standalone applications, we strongly recommend using *fuzz testing* for testing the file handling function as well as functions receiving data from the network [6].

In addition to this rather general remark, we would like to address two areas of client applications in particular:

- Today, many client applications are actually web clients, i.e., written in HTML5/JavaScript. For these clients, the well-known web vulnerabilities (e.g., XSS) need to be considered particularly which, often, requires additional manual penetration testing.
- Client technology seems to change particularly frequently which often creates periods in which a technology is already used for building applications but not yet supported well by automated security testing tools. In this case, again, manual security testing needs done in addition.

4.2 Mobile Applications

On the one hand, modern mobile applications are not significantly different from client and standalone applications and, thus, the recommendation discussed in Section 4.2 are equally important for mobile applications. On the other hand, the usability (limited screen size, lack of a keyboard) becomes a much more influencing factor on security. To mitigate this, architectural review as well as penetration and usability testing should be applied routinely as well.

4.3 Application Servers

An application server is merely a container for applications that provides network connectively. As such, security testing of application servers needs to focus on ensuring that the core networking and security (e.g., access control enforcement, validation of cryptographic certificates) is implemented correctly. Thus, fuzzing as well as static code analysis for detecting, e.g., buffer overflows in the networking stack is particularly important.

Testing the core security functionality of an application server requires usually the development of dedicated test applications that are used during a dynamic penetration test and that, due to their specific nature, concentrate the penetration test on the application server itself.

4.4 Sever Applications

A server application, i.e., an application that is deployed in an application server should be tested using static code analysis. An effective and efficient static analysis of such applications requires that the interfaces provided by the application server are known to the static analysis tool. Additionally, dynamic tests (in particular penetration testing and fuzzing) should be used for ensuring that the interplay of the protection mechanism of the application server and the application itself does not contain vulnerabilities.

4.5 Storage

Storage servers, e.g., database servers, are can be treated similar to application servers. There are defined interfaces which pass the data into the actual storage. Moreover, similar to applications servers, core security features such as access control is usually provided.

Thus, in general, the same general approach should be followed (see Section 4.4). As storage servers process data (compared to just passing data to an application) the overall risk of buffer overflow related vulnerabilities is much higher. Consequently, such issues should receive a special consideration during static code analysis as well as during fuzz testing. As storage is, per se, stateful, dynamic tests (e.g., penetration tests) need to observe longer system traces which makes them more complex.

4.6 Three-tiered Architecture

First, a three-tiered application consists out of the three tiers: client, server, and storage and, thus, the recommendation discussed in Sections 4.1-4.5 apply here as well.

Second, a thorough security approach for multi-tiered application requires also a holistic consideration of the complete stack: in particular, we recommend threat modelling to identify the attack surface and the critical components as well as the most critical communications paths between the components. Dynamic testing (e.g., penetration testing) should be used for ensuring the security of the complete stack.

4.7 Two-tiered client server architecture

A two-tiered application is mainly a three-tier application that tightly integrates the database tier and the application tier. Thus, the recommendations for three-tier applications apply, in principle, to a two tier as well. Due to the tight integration of the application tier and the database tier, a testing of the application and the database “in isolation” is usually not possible. On the one hand, this tight integration allows static analysis tools to more precisely analyse the data-flow between application and database. On the other hand the setup for dynamic tests requires more attention as the risk of modifying application data during the tests is, compared to a three-tier architecture, higher.

4.8 Cloud Applications

With respect to security testing, the speciality of cloud applications is their provisioning model as well as its continuous development model. Moreover, cloud applications are usually updated more frequently (e.g., releasing a new version every two weeks is rather typical) and updates can be pushed to customers comparatively easily.

Thus, as first step, the architecture of a cloud application should be analysed and testing rec-

ommendations for all identified application types should be followed.

Moreover, the frequent releases are both a blessing and a curse: on the one hand, security fixes can be pushed to customers quickly. On the other hand, security testing needs to be integrated very smoothly into the development and release cycle to avoid a slowdown of the release cycle.

5 Conclusion

Developing software that adheres to high security standards requires a holistic approach. The security testing techniques that we discussed in his article offer a rich toolbox that supports software vendors in improving the security of their products.

While our experience [5] as well as independent research [7] shows that static code analysis is the most efficient and effective security testing method *if only one method* is applied during software development, we are convinced that applying *only* static source code analysis is not sufficient. While security testing techniques complement each other (e.g., there are vulnerabilities that cannot be detected by static source code analysis that can be detected by penetration testing), finding many problems at a late stage of the development (e.g., during a penetration test) is a clear indication that there are serious problems in earlier stages.

For reaching a high standard in application security, a carefully selected combination of testing techniques is necessary. Selecting the best combinations of tools from this toolbox is challenging. In this article, we only scratched the surface with respect to rather technical selection criteria. In reality, many other aspects, such as the required effort in training and using security testing tools or maintenances costs need to be considered as well.

It is important to understand that security testing as such is not a guarantee for secure software. First, a security testing approach needs to be turned into a *holistic security testing strategy* that is tightly integrated into a secure software development lifecycle. A successful security testing strategy needs to ensure at least that, for each developed product, the selection of security testing techniques is justified, the security testing is carried out properly, the test results are carefully analysed and weaknesses found are fixed. It is also recommended to include tests for detected weaknesses into the regular regression test activities to ensure that weaknesses are not re-introduced accidentally. In addition, the decision taken (e.g., the selection of test methods) needs to be revised regularly to ensure that the selected tools and techniques are always the best possible choice.

Still, even the best security testing strategy and its careful implementation cannot guarantee the absence of all security weaknesses. For example, other activities a secure software development lifecycle needs to cover are security awareness programs and secure programming trainings as well as effective security response process. The latter needs not only to react on externally reported issues, monitoring publicly available vulnerability database (e.g., [2]) for third party components is equally important.

Finally, as we need to accept that the fact that we will not able to deliver the totally secure product, an effective and efficient patch process and the necessary communication to customers' needs to be considered.

Acknowledgements

We would like to thank our colleagues Lars Brueckner, Luca Compagna, Markus Ehrnsperger, Paul el Khoury, Andrey Hoursanov, Gerold Huebner, Christian Koschmieder, Benedict Kwok, Holger Mack, Svetoslav Manolov, Maik Mueller, Philip Miseldine, Juergen Schneider, Uwe Sodan, Jochen Wickel, and Christian Wippermann for valuable comments and suggestions that improved the quality of the paper.

Bibliography

[1] Mitre. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/, 2011. Site visited on 2014-01-06.

[2] National Institute of Standards and Technology (NIST). National Vulnerability Database. http://nvd.nist.gov/. Site visited on 2014-01-06.

[3] M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report Planning Report 02-03, National Institute of Standards & Technology, May 2002.

[4] Howard, Michael; Lipner, Steve (June 2006). The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software. Microsoft Press.

[5] Achim D. Brucker and Uwe Sodan. Deploying Static Application Security Testing on a Large Scale. In GI Sicherheit 2014.Lecture Notes in Informatics, GI, 2014.

[6] Patrice Godefroid, Michael Y. Levin, David A. Molnar: SAGE: whitebox fuzzing for security testing. Commun. ACM 55(3): 40-44 (2012)

[7] Riccardo Scandariato, James Walden, and Wouter 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, November 2013.