Security Testing: Myths, Challenges, and Opportunities
Experiences in Integrating Security Testing “End-to-End” Into the Software Life-Cycle at SAP

Achim D. Brucker
achim.brucker@sap.com  http://www.brucker.ch/
SAP SE, Vincenz-Priessnitz-Str.  1, 76131 Karlsruhe, Germany

SECTEST Keynote
6th international Workshop on Security Testing (SECTEST)
Graz, Austria, April 13, 2015
Security Testing: Myths, Challenges, and Opportunities
Experiences in Integrating Security Testing “End-to-End” Into the Software Life-Cycle at SAP

Abstract

Security testing is an important part of any security development lifecycle (SDL) and, thus, should be a part of any software (development) lifecycle. Still, security testing is often understood as an activity done by security testers in the time between “end of development” and “offering the product to customers.”

On the one hand, learning from traditional testing that the fixing of bugs is the more costly the later it is done in development, security testing should be integrated into the daily development activities. On the other hand, developing software for the cloud and offering software in the cloud raises the need for security testing in a “close-to-production” or even production environment. Consequently, we need an end-to-end integration of security testing into the software lifecycle.

In this talk, we will report on our experiences on integrating security testing “end-to-end” into SAP’s software development lifecycle in general and, in particular, SAP’s Secure Software Development Lifecycle (S²DL). Moreover, we will discuss different myths, challenges, and opportunities in the area security testing.
A Security Testing Taxonomy

- Manual
  - Black-Box
    - Dynamic: Manual Penetration Testing
    - Static: Manual Binary Analysis
  - White-Box
    - Dynamic: Manual Penetration Testing
    - Static: Manual Code Review
- Automated
  - Black-Box
    - Dynamic: Web Vulnerability Scanning
    - Static: Static Binary Analysis
  - White-Box
    - Dynamic: Runtime Memory Analysis
    - Static: Static Source Analysis
In this talk, security testing refers to all kind of methods that find security vulnerabilities in systems, including (but not limited) to:

- static approaches (e.g., SAST, code reviews)
- dynamic approaches (e.g., DAST, fuzzing)
- combined approaches (e.g., IAST, concolic testing)
Agenda

1. SAP in a Nutshell
2. Motivation
3. The Beginning: Large Scale Introduction of SAST
5. SAP’s Secure Software Development Lifecycle ($S^2$DL)
6. Myths and Lesson’s Learned
Die SAP SE

• Leader in Business Software
  • Cloud
  • Mobile
  • On premise

• Many different technologies and platforms, e.g.,
  • In-memory database and application server (HANA)
  • Netweaver for ABAP and Java

• More than 25 industries

• 63% of the world’s transaction revenue touches an SAP system

• approx. 68 000 employees worldwide

• Headquarters: Walldorf
  (close to Heidelberg, Germany)
De-centralized development model:

- **Central security expert team** (S²DL owner)
  - Organizes security trainings
  - Defines product standard “Security”
  - Defines risk and threat assessment methods
  - Defines security testing strategy
  - Selects and provides security testing tools
  - Validates products
  - Defines and executes response process

- **Local security experts**
  - Embedded into development teams
  - Organize local security activities
  - Support developers and architects
  - Support product owners (responsibles)
My Background

• I wear two hats:
  • Research Expert/Architect
  • *(Global) Security Testing Strategist*

• Background:
  Security, Formal Methods, Software Engineering

• Current work areas:
  • Static code analysis
  • *(Dynamic) Security Testing*
  • Mobile Security
  • Security Development Lifecycle
  • Secure Software Development Lifecycle

http://www.brucker.ch/
Agenda

1. SAP in a Nutshell
2. Motivation
3. The Beginning: Large Scale Introduction of SAST
5. SAP’s Secure Software Development Lifecycle ($^2$DL)
6. Myths and Lesson’s Learned
Costs of Vulnerabilities (Attacks on IT Systems)

- TJX Company, Inc. (2007) $250 million
- Sony (2011) $170 million
- Heartland Payment Systems (2009) $41 million

A hack not only costs a company money, but also its reputation and the trust of its customers. It can take years and millions of dollars to repair the damage that a single computer hack inflicts.

(http://financialedge.investopedia.com/financial-edge/0711/Most-Costly-Computer-Hacks-Of-All-Time.aspx)
Vulnerability Types of CVE Reports Since 1999

- Execute Code: 28%
- Denial of Service: 17%
- Overflow: 12%
- XSS: 11%
- SQL Injection: 8%
- Gain Information: 5%
- Bypass Something: 4%
- Other: 15%

Causes for most vulnerabilities are:
- programming errors
- configuration errors

Patching:
- is expensive
- may introduce new bugs

How can we help developers to avoid these mistakes?
Agenda

1. SAP in a Nutshell
2. Motivation
3. The Beginning: Large Scale Introduction of SAST
5. SAP’s Secure Software Development Lifecycle (S²DL)
6. Myths and Lesson’s Learned
How We Started: What We Wanted to Find
Programming Patterns That May Cause Security Vulnerabilities

Mainly two patterns
Local issues (no data-flow dependency), e.g.,

• Insecure functions

1 \[\text{var } x = \text{Math.random}();\]

• Secrets stored in the source code

1 \[\text{var } password = 'secret';\]

Data-flow related issues, e.g.,

• Cross-site Scripting (XSS)

1 \[\text{var } docref = \text{document.location.href};\]
2 \[\text{var } input = \text{docref.substring(docref.indexOf("default")}+8);\]
3 \[\text{var } fake = \text{function (x) }\{\text{return x;}\}\]
4 \[\text{var } cleanse = \text{function (x) }\{
5 \text{\quad return 'hello_world';}\}\]
6 \[\text{document.write(fake(input));}\]
7 \[\text{document.write(cleanse(uinput));}\]

• Secrets stored in the source code

1 \[\text{var } foo = 'secret';\]
2 \[\text{var } x = \text{decrypt(foo, data);}\]
How We Started: What We Wanted to Find
Programming Patterns That May Cause Security Vulnerabilities

Mainly two patterns
Local issues (no data-flow dependency), e.g.,

- Insecure functions

1 var x = Math.random();

- Secrets stored in the source code

1 var password = 'secret';

Data-flow related issues, e.g.,

- Cross-site Scripting (XSS)

1 var docref = document.location.href;
2 var input = docref.substring(docref.indexOf("default=")+8);
3 var fake = function (x) {return x;}
4 var cleanse = function (x) {
5 return 'hello_world';}
6 document.write(fake(input));
7 document.write(cleanse(uinput));

- Secrets stored in the source code

1 var foo = 'secret';
2 var x = decrypt(foo,data);

We trust our developers, i.e., we are focusing on finding “obvious” bugs.
SAST at SAP

• Since 2010, mandatory for all SAP products
• Multiple billions lines analyzed
• Constant improvement of tool configuration
• SAST tools used at SAP:

<table>
<thead>
<tr>
<th>Language</th>
<th>Tool</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAP</td>
<td>CVA (SLIN_SEC)</td>
<td>SAP</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Checkmarx CxSAST</td>
<td>Checkmarx</td>
</tr>
<tr>
<td>C/C++</td>
<td>Coverity</td>
<td>Coverity</td>
</tr>
<tr>
<td>Others</td>
<td>Fortify</td>
<td>HP</td>
</tr>
</tbody>
</table>

• Further details:
So Everything is Secure Now, Right?

Our tool reports all vulnerabilities in your software – you only need to fix them and you are secure.

Undisclosed sales engineer from a SAST tool vendor.
Our tool reports all vulnerabilities in your software – you only need to fix them and you are secure.

Undisclosed sales engineer from a SAST tool vendor.

Yes, this tools exists! It is called Code Assurance Tool (cat):
Our tool reports all vulnerabilities in your software – you only need to fix them and you are secure.

Undisclosed sales engineer from a SAST tool vendor.

Yes, this tools exists! It is called Code Assurance Tool (cat):

- The cat tool reports each line, that might contain a vulnerability:
So Everything is Secure Now, Right?

Our tool reports all vulnerabilities in your software – you only need to fix them and you are secure.

Undisclosed sales engineer from a SAST tool vendor.

Yes, this tools exists! It is called Code Assurance Tool (cat):

- The cat tool reports each line, that might contain a vulnerability:
- It supports also a mode that reports no false positives:
Agenda

1. SAP in a Nutshell
2. Motivation
3. The Beginning: Large Scale Introduction of SAST
5. SAP’s Secure Software Development Lifecycle ($^2$DL)
6. Myths and Lesson’s Learned
Combining Multiple Security Testing Methods and Tools

- Risks of only using only SAST
  - Wasting effort that could be used more wisely elsewhere
  - Shipping insecure software
- Examples of SAST limitations
  - Not all programming languages supported
  - Covers not all layers of the software stack
Combining Multiple Security Testing Methods and Tools

- Risks of only using only SAST
  - Wasting effort that could be used more wisely elsewhere
  - Shipping insecure software
- Examples of SAST limitations
  - Not all programming languages supported
  - Covers not all layers of the software stack
Combining Multiple Security Testing Methods and Tools

• Risks of only using only SAST
  • Wasting effort that could be used more wisely elsewhere
  • Shipping insecure software

• Examples of SAST limitations
  • Not all programming languages supported
  • Covers not all layers of the software stack
Combining Multiple Security Testing Methods and Tools

- Risks of only using only SAST
  - Wasting effort that could be used more wisely elsewhere
  - Shipping insecure software
- Examples of SAST limitations
  - Not all programming languages supported
  - Covers not all layers of the software stack
A Risk-based Test Plan

- Combines multiple security testing methods, e.g., code scans, dynamic analysis, manual penetration testing or fuzzing
- Selects the most efficient test tools and test cases based on the risks and the technologies used in the project
- Re-adjusts priorities of test cases based on identified risks for the project
- Monitors false negative findings in the results of risk assessment
Agenda

1. SAP in a Nutshell
2. Motivation
3. The Beginning: Large Scale Introduction of SAST
5. SAP’s Secure Software Development Lifecycle ($S^2DL$)
6. Myths and Lesson’s Learned
SAP’ Secure Software Development Lifecycle (S²DL)

Start of development

Preparation
- Training
  - Security awareness
  - Secure programming
  - Threat modelling
  - Security static analysis
  - Data protection and privacy
  - Security expert curriculum

Risk Identification
- SECURIM (Security Risk Identification and Management)
- Data Privacy Impact Assessment
- Threat Modeling

Development
- Plan Security Measures
  - Plan product standard compliance
  - Plan security features
  - Plan security tests
  - Plan security response

Secure development
- Secure programming
- Static code scan
- Code review

Security testing
- Dynamic testing
- Manual testing
- External security assessment

Transition
- Security validation

Utilization
- Security response
- Independent security assessment
- Execute the security response plan

Release decision

Figure: SAP SSDL
SAP’ Secure Software Development Lifecycle (S²DL)


Security Measure Plan

Security Testing Plan
• Based on Security Risk Identification and Mitigation Report (Threat Modelling, SECURIM)
• Describes planned security testing activities
• Completeness and plausibility check by validation or security enablement team

Security Measure Report

Security Testing Report
• Result of executed security testing activities (e.g., code scan report)
• Describes deviations from plan
• Input for validation and operation (cloud)
Agenda

1. SAP in a Nutshell
2. Motivation
3. The Beginning: Large Scale Introduction of SAST
5. SAP’s Secure Software Development Lifecycle ($^2$DL)
6. Myths and Lesson’s Learned
Continuously Measure Your Work and Improve Your Setup

But How to Measure and What to Expect?

**What we do:**
- Externally reported vulnerabilities/found by validation: check why we missed it earlier
- Potential reasons for missing a vulnerability (and actions)
  - Vulnerability not detected by our tools (strategy)
    - could be detected in principle by our tools
      ⇒ analyze necessary changes (with tool vendor) and decide if risk justifies effort for enhancing tool
    - cannot be detected in principle by our tools
      ⇒ research for suitable tools and decide if risk justifies effort for introducing new tool
  - Vulnerability can be detected by our tools
    - With recent configuration but not configuration at release date
      ⇒ no immediate actions necessary
    - With configuration at release date
      ⇒ analyze why it was not detected and take further actions

**What we expect**
- Issues not covered by current tool configuration should increase (ideally to 100%)

**What we observe**
- Increase of logic-based flaws
Penetration Tests at the End of Development

... test/ensure the security of the developed product, right?

Main purpose of penetration tests at end of development is:

- to check for “flaws” in the S²DL (and not the product)
- Ideally, they only find:
  - no issues that can be fixed/detected earlier (e.g., configuration)

**Note,** penetration tests in productive environments are different:

- They test the actual configuration
- They test the productive environment (e.g., cloud/hosting)
False Positives are not Your Biggest Concern

A Pragmatic Solution for Too Many Findings: Prioritize Them

- What needs to be audited
- What needs to be fixed
  - as security issue (response effort)
  - quality issue
- Different rules for
  - old code
  - new code
False Positives are not Your Biggest Concern

A Pragmatic Solution for Too Many Findings: Prioritize Them

- What needs to be audited
- What needs to be fixed
  - as security issue (response effort)
  - quality issue
- Different rules for
  - old code
  - new code
False Positives are not Your Biggest Concern
A Pragmatic Solution for Too Many Findings: Prioritize Them

- What needs to be audited
- What needs to be fixed
  - as security issue (response effort)
  - quality issue
- Different rules for
  - old code
  - new code
False Positives are not Your Biggest Concern
A Pragmatic Solution for Too Many Findings: Prioritize Them

- What needs to be audited
- What needs to be fixed
  - as security issue (response effort)
  - quality issue
- Different rules for
  - old code
  - new code
We are often talking about a lack of security awareness and, by that, forget the problem of lacking development awareness.

Always keep in mind:
Building a secure system more difficult than finding a successful attack.

We need:

• Easier to use security APIs
• More tools that make it easy to implement system securely
• Frameworks that make it hard to implement insecure systems
• ...

And, btw, this also holds for DevOps (Cloud)
Thank you!

[Comic text]

http://xkcd.com/327/
Ruediger Bachmann and Achim D. Brucker.
Developing secure software: A holistic approach to security testing.

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
Formal firewall conformance testing: An application of test and proof techniques.

Achim D. Brucker and Uwe Sodan.
Deploying static application security testing on a large scale.
ISBN 978-3-88579-622-0.

Achim D. Brucker and Burkhart Wolff.
On theorem prover-based testing.
ISSN 0934-5043.