Proactive Application Security

Weg von der Büchse der Pandora

1 Einleitung

Wir leben im Zeitalter von Dynamite Phishing, Erpressungstrojanern, Passworthacks und Kreditkartenleaks. Dennoch wird Applikationssicherheit heute häufig immer noch erst in den finalen Phasen relevant: Dem Test oder dem Betrieb. Doch damit öffnet man die Büchse der Pandora – was den gesamten Entwicklungsprozess lang verborgen blieb, wird nun offenbar, durch einen Penetrationstest oder (schlimmer!) einen Hack!

Wir zeigen, dass dies auch anders geht –in der Designphase wird dabei mit Threat Modeling ein Bedrohungsszenario entworfen, um Bedrohungen und Gegenmaßnahmen zu identifizieren. Während der Implementierungsphase sorgt eine werkzeuggestützte Static Source Code Analyse des Quellcodes für die Identifizierung der Sicherheitslücken.

2 Shift Left

Die Idee eines Secure Software Development Lifecycle ist nicht neu – der „Microsoft Security Development Lifecycle" ist in einer ersten Version um 2004 entstanden (vgl. [MSFT]). Parallel dazu setzte sich aber die Erkenntnis durch, dass der in traditionellen Phasenmodellen praktizierte Dreisprung *Design, Implement, Test* den Anforderungen der IT des 21. Jahrhunderts nicht mehr gerecht wird. Insbesondere im Hinblick auf die Qualitätseigenschaft „Sicherheit" erscheint ein Ansatz, in dem erst das fertige Produkt mittels Durchführung von Penetrationstests geprüft wird hochriskant – im schlimmsten Fall treten erst dort unklare Anforderungen oder Schwachstellen im Design zutage, die enorm aufwendige Modifikationen der Software erfordern.

Nicht zuletzt daher hat sich die Erkenntnis durchgesetzt, dass Security bereits möglichst früh im Softwarelebenszyklus Berücksichtigung finden muss[1] – sowohl mittels konstruktiver Methoden, als auch im Hinblick auf analytische bzw. qualitätssichernde Maßnahmen. Darüber hinaus laden moderne, iterativ-inkrementelle Modelle aus dem Spektrum der Agilität gerade dazu ein, Sicherheitsrisiken möglichst frühzeitig, ganzheitlich und gemeinsam auszuschalten. Hierzu bieten sich zwei Techniken an, die nach unserer Erfahrung im Softwarelebenszyklus heute noch zu selten eingesetzt werden und daher ein großes Potential bieten, die Risiken von Sicherheitslücken im Code weiter zu verringern.

3 Der Secure Software Development Lifecycle

Der Software Development Lifecycle unterstützt Entwickler darin, ihre Software vor - gezielten oder ungezielten - Angriffen zu schützen. Oft wird Security ja nur mit dem Thema Penetrationstest assoziiert der Software Development Lifecycle jedoch bietet für jede Entwicklungsphase eine geeignete (andere) Methode, um das Sicherheitsniveau zu erhöhen. Dieses methodische Vorgehen sorgt weiterhin dafür, dass sich die verwendeten Methoden ohne weitere Schwierigkeiten in den jeweils verwendeten Entwicklungsprozess integrieren lassen.

---

[1] Daher auch der Begriff *Shift Left*, der die Tätigkeiten möglichst weit nach links, d.h. an den Anfang des Lebenszyklus verschieben möchte.

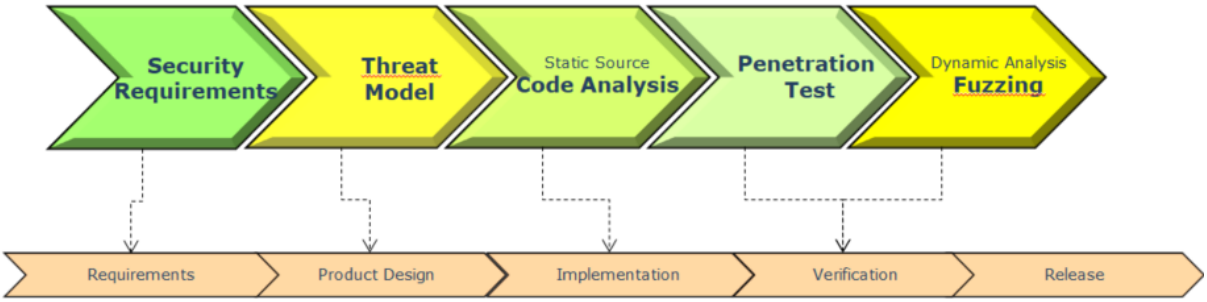

Abbildung 1: Software Development Lifecycle

Abbildung 1 zeigt die Zuordnung der passenden Methoden zu der jeweiligen Entwicklungsphase. Die erste Entwicklungsphase ist die Bestimmung der Security-Anforderungen an das Produkt. Neben funktionalen Anforderungen wird hier das Augenmerk insbesondere auf die Security Requirements gelegt, um möglichst frühzeitig im Entwicklungsprozess die benötigten Sicherheitsmaßnahmen zu definieren.

Während der Designphase wird die Methodik *Threat Modeling* empfohlen. Hierbei handelt es sich um eine Bedrohungsanalyse, die das Design systematisch auf Bedrohungen (Threats) untersucht und sie identifiziert. Für die identifizierten Threats werden Mitigierungsmaßnahmen definiert und anschließend in das Design aufgenommen. Die Methode wird weiter unten noch näher beleuchtet.

Das nun sichere Design wird in der Implementationsphase umgesetzt. Um auch in dieser Phase IT-Sicherheit zu gewährleisten, wird der erstellte Programmcode durch eine Code Analyse auf Sicherheitslücken untersucht. Wie genau dies geschieht und welche Sicherheitslücken eine Bedrohung darstellen, wird ebenfalls unten noch näher behandelt.

In der vierten Phase werden die beiden Methoden Penetration Testing und Fuzzing angewandt. Penetration Testing ist die wohl bekannteste und derzeit am häufigsten eingesetzte Methode. Sie dient dazu, Sicherheitslücken in dem (fast-) fertigen Produkt zu identifizieren. Die Durchführung erfolgt in der Regel durch eine Mischung aus toolgestützten Angriffen und manuellen Untersuchungen des Angriffsziels. Hierbei unterscheidet man u.a. zwischen einem Black-Box und einem White-Box Test. Bei einem Black-Box Test erhält der Tester fast oder gar keine Informationen über sein Zielsystem. Bei einem White-Box Test kann der Tester sogar der dokumentierte Quellcode zur Verfügung gestellt werden.

Unter Fuzzing wird das gezielte und automatisierte Senden einer Vielzahl von Angriffsstrings und ungültige Eingaben an das Ziel verstanden. Hierfür werden im ersten Schritt alle relevanten Interfaces / Ein- und Ausgabeschnittstellen des Zielsystems identifiziert. Für diese Interfaces werden passende Angriffswerte bestimmt und anschließend an das System gesendet. Das angegriffene System wird bei der Bearbeitung dieser Werte auf fehlerhaftes Verhalten und Anomalien hin überwacht. Treten solche Anomalien auf, werden Sie von Experten untersucht, mit dem Ziel Sicherheitslücken zu identifizieren.

Schon vor 15 Jahren hat Microsoft mit seinem Trustworthy Computing Security Development Lifecycle (SDL) ein Konzept vorgestellt, welches Software-Entwickler dabei unterstützen soll die von ihnen entwickelte Software vor böswillige Angriffen zu schützen. Dieser SDL umfasst noch weitere Maßnahmen wie etwa eine gezielte Schulung im Thema IT-Security.

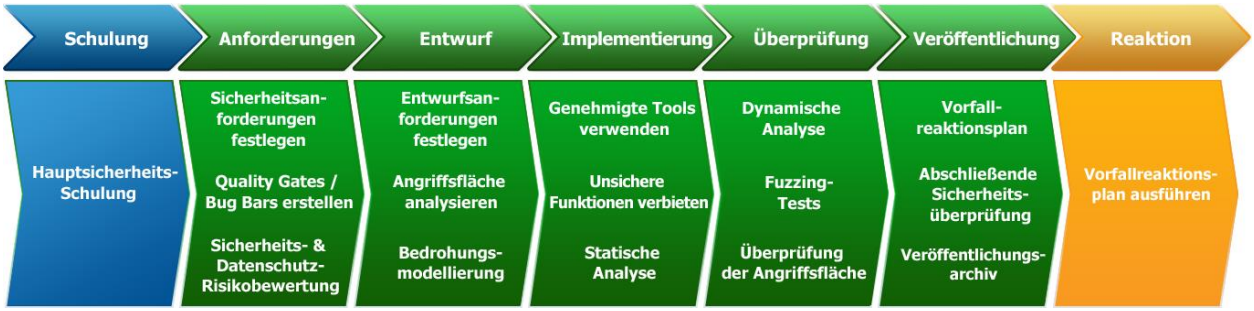

Abbildung 2: Trustworthy Computing Security Development Lifecycle

Abbildung 2 stellt den vereinfachten Microsoft Security Development Lifecycle da. Der vollständige SDL besteht aus über 20 verschiedenen Tätigkeiten in den einzelnen Phasen.

4 Threat Modeling

Threat Modeling unterstützt die methodische Entwicklung eines sicheren Systementwurfs und einer Architektur in der Designphase der Softwareentwicklung. Threat Modeling kann sowohl für die Darstellung von Programmabläufen wie auch für Netzwerkumgebung genutzt werden.

Der Threat Modeling Prozess besteht aus sechs Phasen (Abbildung 3).

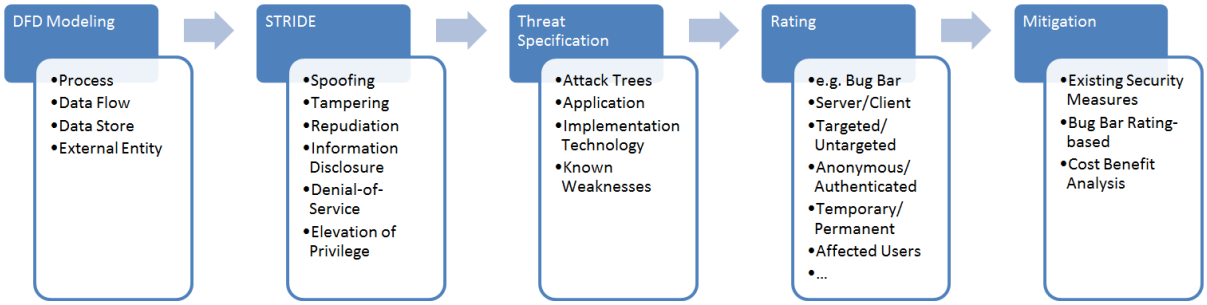

Abbildung 3: Threat Modeling Prozess

Vor Beginn der ersten Phase werden die verfügbaren Dokumente analysiert. Nach dem Auswerten der vorhandenen Dokumentation wird in der ersten Phase ein Data Flow Diagram (Datenflussdiagramm) erstellt.

Data Flow Diagram (DFD) Modeling

Das Datenflussdiagramm besteht aus fünf verschiedenen Elementen, welche zur Darstellung der fließenden Daten verwendet werden. Bei den Elementen handelt es sich um Data Flow, Data Store, External Entity, Process und Trust Boundary, welche grafisch in Abbildung 4 zu sehen sind.

Data Flow	
Data Store	
External Entity	
Process	
Trust Boundary	

Abbildung 4: DFD Elemente

Der Data Flow (Datenfluss) zeigt an, wie Daten innerhalb der Anwendung fließen. Hierbei kann es sich einerseits um Funktionsaufrufe, andererseits um das Versenden von Netzwerkdaten handeln. Wie der Name schon sagt, handelt es sich beim Data Store (Datenspeicherung) um Objekte, die Daten enthalten. Dies kann zum Beispiel eine Datenbank, eine Datei oder etwa gemeinsam genutzter Speicher (Shared Memory) sein. Das nächste Element ist die External Entity (externe Entität). Es bildet sämtliche Entitäten, die von außen mit der Anwendung oder dem Netzwerk kommunizieren ab, etwa externe Server oder Software von Drittanbieter, aber auch User die von außen auf das System zugreifen. Mit dem Element Process werden die internen Aktivitäten abgebildet, indem Daten verarbeitet werden. Das letzte Element ist die Trust Boundary (Vertrauensgrenze). Diese zeigt, wo innerhalb der Anwendung oder des Systems und nach außen hin ein Unterschied der Rechte vorliegt.

STRIDE

Mit Hilfe des erstellten Datenflussdiagramms werden nun die Threats nach der STRIDE Methodik identifiziert. STRIDE ist ein Akronym aus den Anfangsbuchstaben der Kategorien, die den Bedrohungen zugewiesen werden:

- **S**poofing Identity (Identitätsfälschung)
  Der Angreifer gibt sich als vertrauenswürdige Person oder Programm aus, um Zugriff auf das System zu erhalten.
- **T**ampering Data (Datenmanipulation)
  Ein nicht autorisierter Angreifer kann Daten verändern und manipulieren. Dies kann zu verschiedenen schwerwiegenden Angriffen wie Injections führen.
- **R**epudiation (Nicht-Zuordenbarkeit von Ereignissen)
  Ein Angreifer verschleiert seine Aktionen in dem er zum Beispiel die Log-Dateien verändert.
- **I**nformation Disclosure (unberechtigte Weitergabe von Daten)
  Ein Angreifer erlangt Einblick in Daten, für die er keine Leseberechtigung besitzt. Auch eindeutige Fehlermeldungen oder Informationen über das Betriebssystem fallen unter Information Disclosure.
- **D**enial of Service (Einschränkung der Verfügbarkeit)
  Ein Angreifer schränkt die Verfügbarkeit eines laufenden Prozesses ein, in dem etwa die Bandbreite oder andere Ressourcen so stark belastet werden das, dass System nicht mehr erreichbar ist.
- **E**levation of Privilege (Erweitern von Berechtigungen)
  Ein Angreifer übernimmt einen Account eines bestehenden Benutzers mit limitierten Berechtigungen und versucht höherwertige Berechtigungen zu erlangen.

Aus den verschiedenen Kombinationen wie die Daten innerhalb des DFDs zwischen den einzelnen Elementen fließen entstehen Threats welche den jeweiligen Kategorien zugewiesen sind.

Threat Specification

Die dritte Phase des Threat Modeling Prozesses ist die Threat Specification. Nachdem in der vorigen Phase allgemeingültige Bedrohungen identifiziert wurden, wird nun in dieser Phase auf die Besonderheiten der Anwendung oder des Systems eingegangen. Dies dient dazu, eindeutige Threats zu bestimmen. Hierzu werden Attack Trees erstellt, um den Zusammenhang zwischen konkreten Angriffen und den identifizierten Threats her zustellen.

Rating

In der vorletzten Phase geht es um die Bewertung der identifizierten Threats hinsichtlich ihrer Kritikalität (High, Medium, Low). Dies erfolgt anhand der Eintrittswahrscheinlichkeit und dem Schweregrad des Threats. Hierbei wird eine Vielzahl von Bewertungsparametern, die zum Teil zustandsabhängig sind und auch an die Benutzerumgebung angepasst werden können, berücksichtigt:

- Erkennbarkeit für Dritte und Angreifer
- Reproduzierbarkeit
- Lokal nutzbar oder (auch) im Netzwerk und aus dem Internet
- Erforderliche Zugriffsrechte zur Ausnutzbarkeit
- Art und Umfang der Konsequenzen auf die Integrität, Verfügbarkeit und Vertraulichkeit der Daten, Server und Client Systemen
- Gezielte oder ungezielte Angriffen
- Anzahl betroffener User, Wichtigkeit von Systemen
- Kurzzeitige oder andauernde Angriffe - auch nach Neustart

Mitigation

Sind alle Threats identifiziert und priorisiert, werden in der letzten Phase die Behebungsmaßnahmen (Mitigierungsmaßnahmen) für die einzelnen Threats ausgearbeitet. Jeder Threat hat einen Mitigierungszustand. Es werden die folgenden Zustände unterschieden:

- Mitigiert: Die Maßnahmen des aktuellen Entwicklungsstandes mitigieren den Threat. Somit ist der Threat im aktuellen Design schon behoben.
- Nicht mitigiert: Ein Threat ist mit den derzeit verfügbaren Maßnahmen nicht mitigiert. Es muss das Design dahingehend angepasst werden, um den Threat zu mitigieren.
- Nicht zutreffend: Der Threat trifft aus verschiedenen Gründen, wie etwa der Struktur, nicht zu.

Abschließend lässt sich sagen, dass sich mit Hilfe von Threat Modeling viele - möglicherweise auch erst entstehende - Sicherheitslücken schon in der Designphase beheben lassen.

5 Static Application Security Testing

Wenn die Anwendungsarchitektur in groben Zügen steht, beginnt in aller Regel die eigentliche Programmierarbeit. Wird Quellcode geschrieben, so vergeht normalerweise eine gewisse Zeit, bis er Teil einer integrierten, ausführbaren Software wird. Daher können dynamische Tests, die den Code zur Ausführung bringen, auch erst dann durchgeführt

werden. Ist der geschriebene Code allerdings valide, so kann er bereits mit anderen Mitteln untersucht werden, selbst wenn z.B. Abhängigkeiten noch nicht erfüllt sind.

Hier setzt die Methode SAST an. Sie nutzt statische Analysewerkzeuge, um bestimmte Programmierkonstrukte im Code zu finden. Der Vorteil liegt auf der Hand – im Sinne eines Shift Left können Probleme bereits sehr früh identifiziert werden und im Gegensatz zu dynamischen Prüfungen oder Tests entfällt das Bereitstellen von Treibern oder Stubs in der Regel vollständig.

Der Analysator nimmt hierzu den Quelltext als Eingabe, verarbeitet ihn ähnlich wie ein Compiler und extrahiert daraus Informationen, die im Anschluss zur Prüfung mit bekannten Sicherheitsproblemen verglichen werden. Abbildung 5 zeigt dieses Funktionsprinzip. Hier extrahiert das Werkzeug Fakten (hier Metriken) zu einer Reihe von Qualitätsmerkmalen aus dem Quellcode.

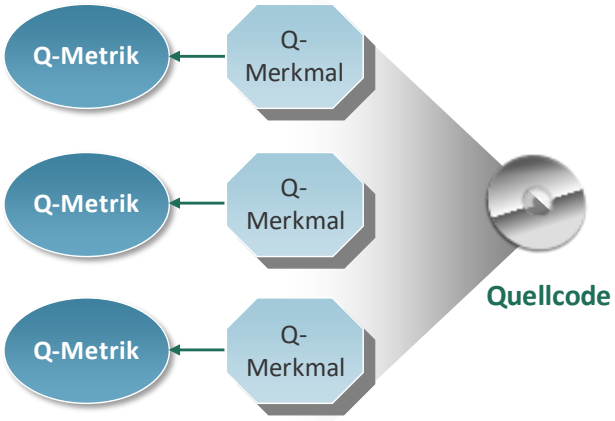

Abbildung 5: Funktionsweise statischer Analysatoren.

Dies beschränkt sich nicht nur auf rein syntaktische Prüfungen innerhalb kleiner Einheiten wie einer Methode – der Analysator kann aus Vererbungs- und Aufrufbeziehungen auch architekturelle Zusammenhänge zwischen Methoden, Klassen und Paketen erkennen. Mit Hilfe solcher Funktionen lässt sich sicherstellen, dass bestimmte sicherheitsrelevante Architekturmuster eingehalten werden, beispielsweise die zwingende Nutzung einer festgelegten Komponente zur Ver- und Entschlüsselung von Daten (zu weiteren Details betreffen statische Codeanalyse vgl. [Sim06]).

Hierbei ist es allerdings wichtig, die Arbeitsweise und damit die Beschränkungen statischer Analysatoren zu bedenken. So können solche Werkzeuge häufig konkrete Antipattern wie die Nutzung einer unsicheren Bibliotheksfunktionalität wie `Math.Random.*` sehr einfach und mit hoher Zuverlässigkeit erkennen. Schwieriger wird es allerdings, wenn auch Daten- und Kontrollfluss berücksichtigt werden müssen, wie z.B. bei der Erkennung einer potentiellen SQL-Injection.

Schließlich entsteht diese Gefahr erst bei der Ausführung des SQL-Statements auf der Datenbank, die Eingabe ungültiger Daten ist hiervon in modernen Softwarearchitekturen aber durch eine Reihe von Schichten getrennt (vgl. Abbildung 6). Das Analysewerkzeug muss dabei nachstellen, wie die Daten eines Angreifers be- und verarbeitet werden, bevor sie in der Datenbank zur Ausführung kommen.

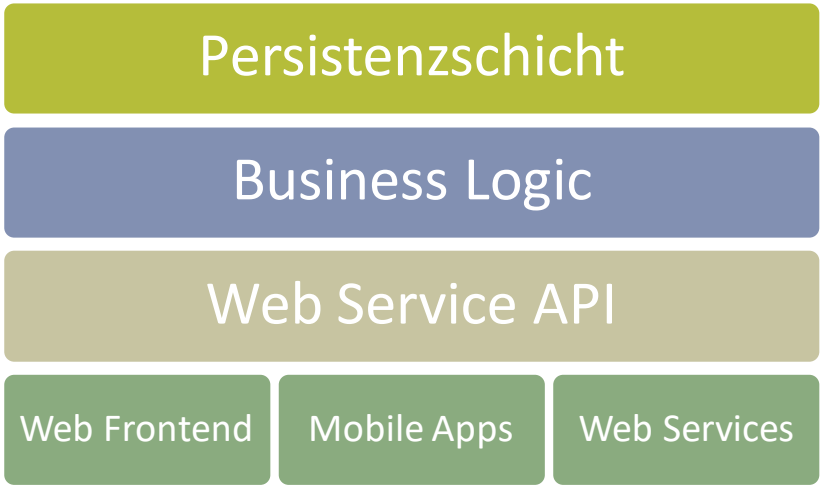

Abbildung 6: Typische Anwendungsarchitektur aus Sicht des Angreifers.

Erschwert wird diese Analyse immer dann, wenn Daten- oder Kontrollflüsse statisch nicht zugänglich sind sondern zur Laufzeit entschieden werden, wie bei Nutzung des *Dependency Injection* Patterns. Hier muss das Analysewerkzeug Annahmen treffen. Dies geschieht über Heuristiken, gelegentlich können dynamische Beziehungen auch aus Konfigurationen (z.B. Spring) ausgelesen werden.

Nichtsdestoweniger entstehen an diesen Stellen aber Unschärfen, die sich später im Prozess als falsch-Positive oder falsch-Negative äußern. Bei einem falsch-positiven Befund identifiziert das Analysewerkzeug ein Finding, wo kein Problem existiert. Umgekehrt äußert sich ein falsch-negativer Befund darin, dass das Analysewerkzeug ein real existierendes Problem nicht findet oder berichtet.

Zusätzlich zu blinden Flecken, die durch die statische Natur der Analyse bedingt sind, stoßen statische Analysatoren dort an Grenzen, wo Sachverhalte aus der Semantik oder der Pragmatik überprüft werden. So kann ein statischer Analysator im Allgemeinen nicht entscheiden, ob ein gewählter Verschlüsselungsalgorithmus konzeptuelle Sicherheitslücken aufweist. Werkzeuge lösen solche Aufgaben derzeit über die Pflege von White- oder Blacklists. Im Falle individuell implementierter Algorithmen hilft dies in der Regel aber nicht weiter.

Das *Open Web Application Security Project* (OWASP) veröffentlicht in regelmäßigen Abständen eine *Top Ten* genannte Liste der schwerwiegendsten Sicherheitslücken in Webapplikationen. Abbildung 7 zeigt die aktuelle Version dieser Liste.

OWASP Top Ten 2017
A1 – Injection
A2 – Broken Authentication and Session Management
A3 – Cross Site Scripting (XSS)
A4 – Broken Access Control
A5 – Security Misconfiguration
A6 – Sensitive Data Exposure
A7 – Insufficient Attack Protection

A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with known Vulnerabilities
A10 – Underprotected APIs

Abbildung 7: Die OWASP Top Ten (vgl. [OWASP]).

Viele dieser Sicherheitslücken können mittels einer statischen Analyse des Codes abgedeckt werden. Allerdings unterscheidet sich der Grad dieser Abdeckung. Während Lücken wie A5 nur teilweise abgedeckt werden können, lassen sich A1, A3 und A9 durch marktgängige Tools meist sehr gut abdecken.

Um statische Analysatoren gezielt einzusetzen, kommen nun auch die Ergebnisse aus vorherigen Aktivitäten wie *Threat Modeling* ins Spiel. Sie bestimmen den Rahmen, in dem Sicherheitslücken auftreten und ggf. ausgenutzt werden können. In manchen Branchen existieren darüber hinaus noch regulatorische Anforderungen wie z.B. PCI-DSS im Zahlungsverkehr, die eigene Anforderungen an die Überprüfung des Codes stellen.

Gemäß dieser Vorgaben wird der Analysator gemäß des in Abbildung 8 abgebildeten, iterativen Prozesses eingesetzt.

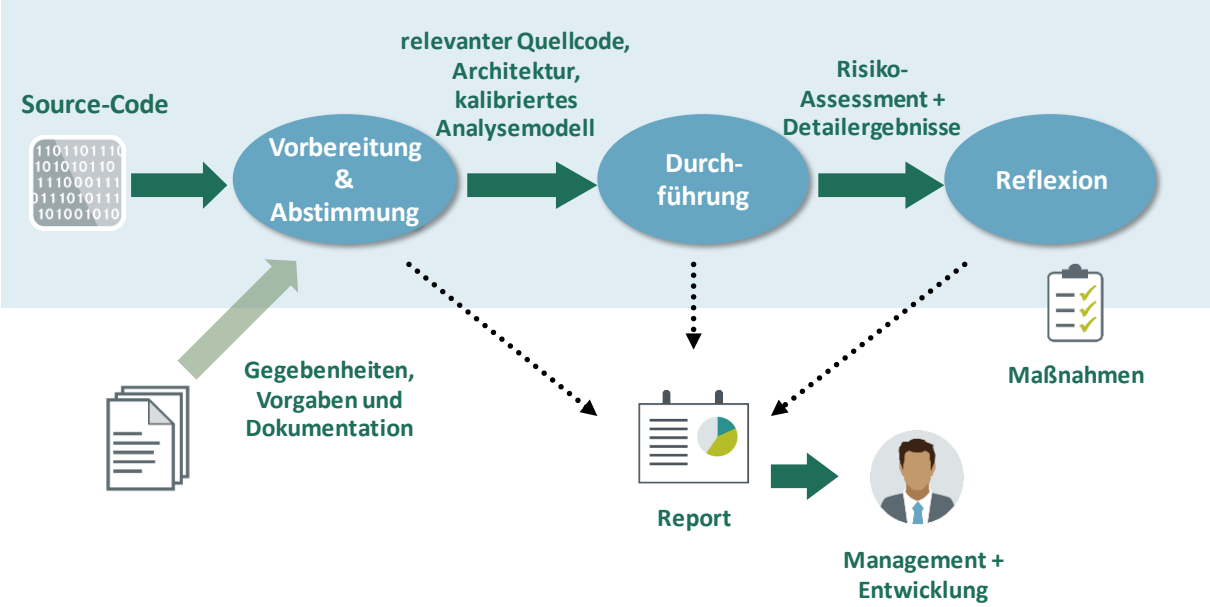

Abbildung 8: Prozess der einmaligen Durchführung einer statischen Analyse.

Dabei wird der Analysator aufgesetzt und auf Basis der Vorgaben parametrisiert, z.B: um Prüfungen auf spezifische Sicherheitslücken ein- oder auszuschalten. Gleichzeitig wird der Quellcode beim ersten Durchlauf einer Analyse einem manuellen Review unterzogen, um sicherzustellen, dass er vollständig und frei von Testcode wie Unit Tests ist. Gleichzeitig wird überprüft, ob die Lieferung nur die erwarteten Technologien enthält. Häufig zeigt sich hier, dass die Codebasis Code in weiteren Programmiersprachen enthält.

Sollen auch Architekturbeziehungen mitüberprüft werden, so wird dem Werkzeug eine Beschreibung der Architektur eingegeben. Im Idealfall liegt diese bereits in einer Sprache wie ArchiMate maschinenlesbar vor. Ist dies nicht der Fall, muss die Soll-Architektur ggf. manuell modelliert werden.

Nun wird der statische Analysator ausgeführt. In der Regel erfolgt dies mehrfach, da die Werkzeuge gerade zu Beginn eine große Zahl an *false Positives* erzeugen. Diese werden nach einem Review im Werkzeug als solche parametrisiert, damit die Ergebnismenge handhabbar bleibt.

Auf Basis dieser belastbaren Ergebnisse werden zielgruppengerechte Auswertungen erzeugt. In der Regel besteht dies aus:

- Einer risikobasierten aggregierten Sicht, die Entscheidern eine Übersicht über den Status Quo liefert
- Eine detaillierte Sicht der Befunde, angeordnet nach Architekturelementen, so dass die verantwortlichen Teams Sicherheitslücken schnell identifizieren und entfernen können.

Nach einem solchen Initialdurchlauf wird die statische Analyse eng mit dem Buildprozess verzahnt und in die CI-Umgebung integriert. Auf diese Weise stehen jederzeit aktuelle Analyseergebnisse zur Verfügung und können in *Definition of Done*[2], Tages- oder Sprintplanungen mit aufgenommen werden.

6 Zusammenfassung

Das Risiko mit Security erst zu warten, bis eine testbare Applikation vorliegt ist enorm und steigt mit der Zeit. Je später die Reaktion desto höhere Kosten und Risiken entstehen. Eine Hilfe ist das frühzeitige systematische Einbinden von Security in den gesamten Software-Lebenszyklus. Dies senkt einerseits die Kosten und minimiert anderseits das entstehende Risiko. Schon die ersten erstellten Artefakte, wie Anforderungen, Architektur oder Quellcode, müssen im Hinblick auf Security qualitätsgesichert werden. Mit Hilfe der Methode Threat Modeling werden konzeptuelle und architekturelle Sicherheitslücken identifiziert noch bevor überhaupt eine Zeile Code geschrieben wurde. Darauf aufbauend ermöglicht es die Methode SAST Abweichungen von der Architektur, Bad Smells und technische Sicherheitslücken im Code zu identifizieren und vor allem sofort du präzise zu korrigieren. Somit entlasten diese beiden Methoden die nachfolgenden dynamischen Analysen – wie Penetrationstest und sorgt für sichere Software zu niedrigeren Kosten.

7 Autoren

	Sascha Böhm (sascha.boehm@softscheck.com) ist Berater bei der IT-Sicherheitsberatung softScheck GmbH. Seine beruflichen Schwerpunkte umfassen IT-Security Audits von Netzwerken, Soft- und Hardware–Produkte sowie die Entwicklung, Prüfung und Bewertung von Security Requirements, Sicherheitskonzepten und Sicherheitsarchitekturen unter Einsatz des softScheck–Security Testing Process.

[2] Hier hat es sich bewährt, in der DoD zu fordern, dass für neu erzeugten bzw. modifizierten Code keine (kritischen) Findings des SAST-Tools mehr vorliegen.

	Sven Euteneuer (sven.euteneuer@expleogroup.com) verantwortet bei der expleo Group als Practice Lead Technical Quality unter anderem das Thema IT Security. Dank über 20 Jahren Erfahrung als Entwickler, Architekt und Codeanalyst kennt er die konstruktive wie die analytische Seite des Softwarelebenszyklus.
	Prof. Dr. Hartmut Pohl (hartmut.pohl@softscheck.com), geschäftsführender Gesellschafter der IT-Sicherheitsberatung softScheck GmbH. Security Consulting mit dem ISO 27034 basierten Entwicklungsprozess sicherer Software und dem Security Testing Process zur Identifizierung unerkannter Sicherheitslücken (Zero-Day-Vulnerabilities): Security Requirements Analysis, Threat Modeling, Static Source Code Analysis, Dynamic Analysis (Fuzzing) und Penetration Testing in Köln/Sankt Augustin.

8 Literaturverzeichnis

[Lip06] Lipner, S.; Howard, M.: The Security Development Lifecycle. Redmond 2006

[Ada14] Adam, S.: Threat Modeling: Designing for Security. Indianapolis 2014

[MSFT] Microsoft Corp: About Microsoft SDL, https://www.microsoft.com/en-us/securityengineering/sdl/about (abgerufen 08.02.2019)

[OWASP] Open Web Application Security Project, OWASP Top Ten, https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (abgrerufen 08.02.2019)

[Sim06] Simon, F., Seng O., Mohaupt T.: Code Quality Management. Heidelberg 2006