achelos

Whitepaper

Security by Design – Positiv- und Negativbeispiele aus der Praxis

Abstract

„‚Security by Design‘ beschreibt eine Art der Soft- und Hardware-Entwicklung, bei der IT-Sicherheit schon in der Entstehung von Produkten und Lösungen berücksichtigt wird. Das Ziel ist es, das Produkt so unempfindlich wie möglich gegen Angriffe zu machen.“

(https://www.computerwoche.de/a/security-by-design-umsetzen,3546232)

„‚Security by Design‘ steht für integrierte Softwaresicherheit und setzt voraus, Sicherheit als explizite Anforderung in den Entwicklungsprozess aufzunehmen sowie ganzheitliche Sicherheitsmaßnahmen von der Initialisierung an zu berücksichtigen, umzusetzen und zu testen.“

(https://www.heise.de/developer/meldung/Sichere-Softwareentwicklung-nach-dem-Security-by-Design-Prinzip-752337.html)

Security by Design – wieder ein neues Buzzword in der Entwicklung? Das die Lösung aller Sicherheitsprobleme verspricht? Brauchen wir das auch noch?

Bisher ging es doch auch ohne – oder?

Agnes Diller | Senior Consultant | agnes.diller@achelos.de

1 SECURITY NOT BY DESIGN – TYPISCHE SZENARIEN

Der Klassiker: Sicherheitslücken durch Weiterentwicklung

„Eigentlich müssten wir ein komplettes Redesign machen."

„Wie das mit den Rechten funktioniert? Das hat damals der Externe gemacht."

„Können wir nicht eben schnell die neuen Zugriffsmethoden noch einbauen?"

Klingt vertraut? Überarbeitungen, die (aus Unkenntnis, Unachtsamkeit oder schlicht Zeitdruck) die bisherige Sicherheitsarchitektur beschädigen oder umgehen und damit Sicherheitslücken einbauen, sind wahrscheinlich allen schon begegnet.

In den letzten Jahren bekommen diese Klassiker eine neue, größere Dimension: Durch einen Wechsel der Betriebsumgebung werden Anwendungen in ein Umfeld versetzt, in dem plötzlich ein viel höherer Sicherheitsbedarf besteht als ursprünglich notwendig.

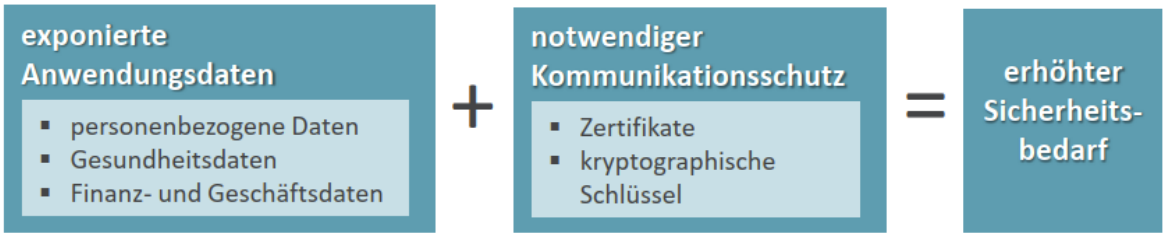
Jetzt neu: Schlecht gesicherte Daten erreichen ein größeres Publikum.

Beispiele:

- Verlagerung von bisher lokal betriebenen Anwendungen ins Internet („YourBusinessCriticalApplication as a Service")
  - Mehrere (konkurrierende) Player nutzen gemeinsame Ressourcen.
  - Angriffsfläche aus dem Internet
- Outsourcing des Betriebs („YourClient'sSensitiveDataCloud")
  - Betrieb durch (vertrauenswürdige?) Dritte
  - Auch hier: Mehrere Player nutzen gemeinsame Ressourcen
  - Angriffsfläche auf dem Übertragungsweg (meist Internet)
- Neue Zugriffswege zu Anwendungen vom unsicheren Endgerät („Easy Smartphone Access to your Health Data for you, Google and your Phone Company")
  - Niedrige Awareness der Nutzer – Smartphone wird nicht als kritisch wahrgenommen
  - Administrationsrechte liegen in der Regel bei Google Store (oder äquivalentem Dienst) und dem Telefonbetreiber.
  - Sicherheitsarchitektur des Telefons (bisher) meist rudimentär
  - Betrieb in wechselnden Netzen

In allen diesen Szenarien ist nun zusätzlicher Schutz der Übertragungswege notwendig, um die Vertraulichkeit der Inhalte und Authentifizierung der Kommunikationspartner zu gewährleisten.

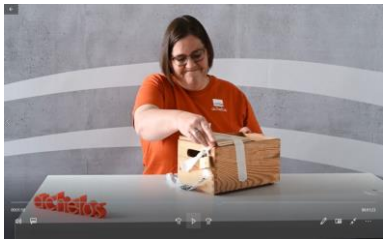
Im neuen Umfeld:



Die häufige Antwort auf diese Herausforderung:

Security not by Design

Die betroffenen Anwendungen sind für den veränderten Betrieb nicht ausgelegt und werden daher beim Wechsel sicherheitstechnisch „nachgerüstet“:



Video abspielen

Wie in der Praxis Security by Design eingesetzt wurde, um Sicherheitsunfälle zu vermeiden, bzw. wie Security not by Design schiefgegangen ist, erzählen die folgenden Kapitel.

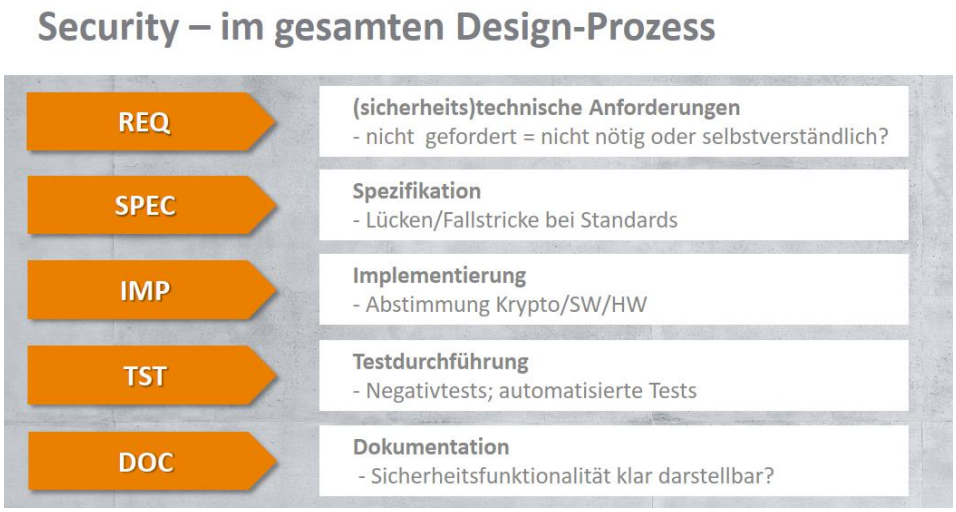


Abbildung 1 – Security im gesamten Design-Prozess

1.1 Requirements: (sicherheits)technische Anforderungen – nicht gefordert = nicht nötig oder selbstverständlich?

Back to the roots – Beispiel: restriktives Gruppenkonzept

Ausgangslage: Anforderungen an die Sicherheitsarchitektur werden allgemein formuliert – für die Umsetzung fehlen konkrete Anforderungen:

„Gruppenkonzept für Zugriffsrechte implementieren“ statt „Folgende Assets sollen durch ein Gruppenkonzept für Zugriffsrechte geschützt werden: ...“.

Insbesondere bei Umbau, längeren Entwicklungsphasen oder Einbau zusätzlicher Funktionen wird das Rechtekonzept weiter aufgeweicht:

Früher konzentrierte sich dieses Problem meist auf die schlichte Frage: „Wer hat Rootrechte?“ Heute sind die Zugriffskonzepte differenzierter, aber am Ende einzelne Gruppen oft genauso mächtig wie „root“ – und werden oft noch genauso schlampig behandelt, zum Beispiel im folgenden Szenario: Es gibt zwar ein Konzept für Zugriffsrechte mit mehreren Gruppen, aber nur eine Gruppe hat „heikle" Zugriffsrechte.

Aber: Bei einer Aufstellung der Gruppenzusammensetzungen im Rahmen der Sicherheitsdokumentation erweist sich, dass mehrere Hundert Prozesse aus ungeklärten („historischen“) Gründen dieser Gruppe zugewiesen sind.

Eine kurzfristige Änderung der Zuordnungen ist wegen schlecht nachverfolgbarer Seiteneffekte nicht möglich (Software stürzt bei Änderung ab).

Hinterfragen beim Requirements Engineering:

- Werden eine passende Sicherheitsarchitektur und Modularisierung als selbstverständlich angesehen?

Der Klügere kann’s nicht retten – Beispiel: „schwachbrüstige“ Kommunikationspartner

Ausgangslage: Die formalen Anforderungen an das Produkt erlauben schwache Kryptographie oder Alternativen zu kryptographischer Sicherung. Die fehlende Forderung nach (starker) Kryptographie verleitet zum Einsatz billigerer Hardware-Komponenten, die keine oder nur schwache Krypto-Algorithmen unterstützen.

In unserem Fall wurde ein „Plausibilitätscheck“ als Alternative zur kryptographischen Absicherung in den Vorgaben für die Produktzertifizierung ausdrücklich erwähnt. Allerdings gab es dazu keinerlei weitere Hinweise oder Erfahrungen. Im Projektverlauf zeigte eine frühzeitige Sicherheitsanalyse, dass die „klügere“ Komponente die Schwäche der Kommunikationspartner nicht ausgleichen kann. Der Einsatz schwacher Kryptographie bzw. das Ausweichen auf Plausibilitätschecks zieht Probleme bei der Robustheit des Systems (falsch-negatives Anschlagen des Plausibilitätschecks) und bei der Wartung / im Betrieb (Justierung der Plausibilitätsparameter) nach sich, da die nicht ausreichenden Sicherheitsmechanismen ausgeglichen werden müssen. Es zeichnete sich die Gefahr ab, dass der Einsatz der billigeren Komponenten zu höheren Aufwänden im Betrieb führen würde.

Hinterfragen beim Requirements Engineering:

- Wie werden die Anforderungen des Betriebs berücksichtigt? Werden erhöhte Aufwände aufgrund „kreativer“, vermeintlich billigerer Lösungen erzeugt?

1.2 Spezifikation – Lücken/Fallstricke bei Standards

Vom Lottogewinn zu „Fast jedes Los gewinnt“ – Beispiel: „Lucky 13“-Timing-Angriff auf TLS-1.2-Implementierung

Ausgangslage: Zur Implementierung der Sicherheit werden Standards eingesetzt, z. B. TLS 1.2. Dadurch entsteht das Gefühl, diesen Aspekt in der Spezifikation vollständig abgedeckt zu haben. Die Spezifikationen (z. B. RFCs) beinhalten aber oft Optionen und Implementierungshinweise zu einzelnen Details. Diese Lücken werden bei der Spezifikation nicht ausgefüllt. Zudem existieren bekannte Angriffe gegen die verbreiteten Protokolle, zu denen es (ebenfalls bekannte) Gegenmaßnahmen bei der Implementation gibt. Diese Gegenmaßnahmen fehlen ebenfalls in der Spezifikation.

Das führt zu folgendem Szenario:

Bei der Entwicklung werden eigene Implementierung und Bibliotheken (oft in mehreren Versionen) eingesetzt, die Entscheidung für bestimmte Optionen fällt zufällig (i. d. R. durch Defaultwerte der Bibliothek). Ob die verwendete Version die für das Produkt relevanten Gegenmaßnahmen enthält, wird nicht bewertet.

Beispiel: „Lucky 13“-Timing-Angriff auf das Padding in TLS 1.2, betroffen sind mehrere Cipher-Suites mit CBC-Mode (MAC-Encode-Encrypt) [http://www.isg.rhul.ac.uk/tls/TLStiming.pdf].

Bei diesem Angriff handelt es sich um eine Timing-Attacke. Dabei wird durch gezielte Veränderung/Verkürzung der Nachricht der Empfänger dazu verleitet, Teile der eigentlichen Nachricht als Padding zu interpretieren. Je nachdem wie sehr der Nachrichtentext einem korrekten Padding gleicht, wird das Zeitverhalten bei der Verarbeitung der Nachricht beeinflusst. Durch mehrfache Versuche und deren statistische Analyse ist es möglich, Teile der Klartext-Nachricht bis hin zur gesamten Nachricht zu rekonstruieren.

Der RFC gibt Hinweise zur sicheren Implementierung, aber die RFC-Hinweise

- sind schwierig umsetzbar (konstanter Zeitverbrauch) und
- reichen in bestimmten Fällen nicht aus.

Allgemein ist dieser Angriff relativ ungefährlich, da die benötigte Anzahl der manipulierten Nachrichten und TLS-Sessions sehr hoch ist. Kennt man aber bereits einen Teil der Nachricht bzw. sind die Nachrichteninhalte durch Formate und Vorgaben eingeschränkt, ändert sich das. Im einem unserer Projekte im eHealth-Bereich etwa werden über die TLS-verschlüsselte Verbindung

- Nachrichten in festgelegten Formaten übertragen (i. e. SOAP),
- nur eine eingeschränkte Anzahl von Nachrichtentypen ausgetauscht,
- die Inhalte der Nachrichten oft bis auf wenige Felder global oder für die jeweiligen Kommunikationspartner statisch festgelegt.

Dadurch reduziert sich die Menge der notwendigen Sessions beträchtlich, und der Angriff wird realistisch. In unserem Beispiel konnten rechtzeitig zusätzliche Gegenmaßnahmen ergriffen werden; die Gefahr einer Projektverzögerung war aber gegeben.

Hinterfragen bei der Spezifikation:

- Sollten Vorgaben für die genutzten Bibliotheken gemacht werden?
- Werden bekannte Angriffe durch die verwendeten Versionen abgewehrt?
- Werden Angriffe durch statische Dateninhalte erleichtert?

1.3 Implementation – Vertrauenswürdigkeit des Codes, sichere Speicher

Trau, schau, wem! – Beispiel: Docker und TPM

Ausgangslage: Der Betrieb sensitiver Anwendungen soll in eine Cloud verlagert werden. Dadurch entfällt die direkte Kontrolle darüber, ob (absichtlich oder unabsichtlich)

- manipulierte oder unvollständige Software oder
- Testversionen der Applikation

ausgeführt werden.

In unserem Beispiel wurde dieses Problem frühzeitig erkannt und in der Implementierungsphase die Software in mehreren Schichten (Betriebssystem, Docker Daemon, Docker Container) jeweils mit einer Signatur versehen. Die verwendeten Server werden mit einem TPM ausgerüstet, in dem ein Vertrauensanker zur Signaturprüfung abgelegt wird. Dadurch kann mithilfe eines „Authenticated Boot"-Verfahrens schichtweise geprüft werden, dass nur korrekt signierte Software in Betrieb genommen wird. Zusätzlich wurde ein „Remote Attestation"-Mechanismus eingebaut, der es erlaubt, die Software während der Ausführung zu überprüfen.

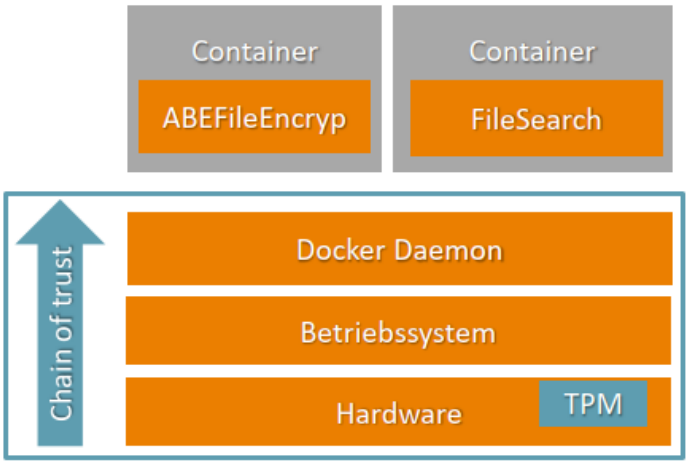


Abbildung 2 – „Chain of Trust" in einer sicheren Applikation

Durch das Zusammenspiel mehrerer Komponenten wurde so die Gefährdung durch einen evtl. nicht vertrauenswürdigen Betreiber verringert.

[Johannes Blömer, Peter Günther, Volker Krummel, Nils Löken: Attribute-Based Encryption as a Service for Access Control in Large-Scale Organizations, 2017]

Hinterfragen bei der Implementation:

- Wie kann eine unsichere Betriebsumgebung ausgeglichen werden?

Teile und herrsche – Beispiel: geeignete Verfahren und Aufteilung der Prozesse

Ausgangslage: Im gleichen Projekt wurde zwar ein Teil der Anwendung in eine „klassische" Cloudumgebung verlagert, ein weiterer Teil aber in einer besser gesicherten Betriebsumgebung („Trusted Cloud") realisiert, deren Performance allerdings begrenzt ist. Die Herausforderung bestand nun in der geeigneten Aufteilung der Anwendung, um die Vorteile der Cloud sicher nutzen zu können und die Auslastung der gesicherten Umgebung zu begrenzen.

In unserem Beispiel konnte durch die Implementierung geeigneter Kryptoverfahren und die Ausnutzung von Invarianz-Eigenschaften die Anwendung in rechenintensive Teile in der Cloudumgebung und weniger rechenintensive Teile im sichereren Bereich aufgeteilt werden.

Ebenfalls: [Johannes Blömer, Peter Günther, Volker Krummel, Nils Löken: Attribute-Based Encryption as a Service for Access Control in Large-Scale Organizations, 2017]

Hinterfragen bei der Implementation:

- Passende Software-Architektur zur Verteilung der Prozesse auf die Betriebsumgebung? Können geeignete Kryptoverfahren unterstützen?

1.4 Testdurchführung – Negativtests, vollständige Tests, automatisierte Tests

Ein Test sagt mehr als tausend Code-Zeilen – schneller Nachweis durch Negativtests

Ausgangslage: TLS wird an mehreren Stellen in der Anwendung genutzt, z. T. werden unterschiedliche Bibliotheken kombiniert. Die Implementierung ist unübersichtlich, u. a. durch den Einsatz zweier verschachtelter, komplexer State Machines. Nun muss nachgewiesen werden, dass die TLS-Implementierung gegen bestimmte Angriffe immun ist. Einer dieser Angriffe ist „Early CSS“, d. h. die Verarbeitung einer ChangeCipherSpec-Message vor Schlüsselaushandlung. Dabei wird ausgenutzt, dass diese ChangeCipherSpec-(CSS-)Message nicht Teil des TLS-Handshake-Protokolls zur Schlüsselaushandlung ist und daher kein bestimmter Zeitpunkt dafür vorgesehen ist. Beispielsweise war die OpenSSL-Implementierung bis 2014 gegen diesen Angriff anfällig.

[http://www.ieee-security.org/TC/SP2015/papers-archived/6949a535.pdf]

In unserem Beispiel konnte durch den Einsatz passender Negativtests einfach und schnell nachgewiesen werden, dass vorzeitige CSS-Messages nicht verarbeitet werden.

Hinterfragen beim Test:

- Welche häufigen Sicherheitslücken (z. B. bekannte Angriffe) können durch Negativtests einfach geprüft werden, wie B. durch automatisierte Tests mit dem TLS Inspector von achelos?

Last-minute-Fix = next month testing? – Automatisierung zahlt sich aus

Ausgangslage: Kurz vor Projektende werden Fehler in der Implementierung festgestellt. Ein Bugfix ist relativ schnell möglich. Die Tests werden allerdings manuell durchgeführt, da eine Automatisierung als zu aufwendig verworfen wurde.

In unserem Beispiel benötigt eine komplette Wiederholung der manuellen Tests aufgrund begrenzter Testbeistellungen mehrere Wochen und sprengt dadurch den Abgabetermin. Die Alternative sind reduzierte Tests, die die Gefahr bergen, unerwünschte Seiteneffekte des Bugfix zu übersehen. Auch die Testdokumentation muss aufwendig manuell angepasst werden – ein erhöhtes Risiko für Fehler. Durch die Entscheidung gegen die vermeintlich teureren automatisierten Tests kommt es zu Sicherheitsrisiken und Verzögerungen.

Hinterfragen beim Test:

- Werden, soweit möglich, automatisierte Tests eingesetzt?
- Können Beistellungen durch Simulationen ersetzt werden?

1.5 Dokumentation – Sicherheitsfunktionalität klar darstellbar?

Automatisch generierter Aufwand – Code- und Dokumentationsgenerierung muss zur Sicherheitsarchitektur passen

Ausgangslage: Zur Implementierung wird Code-Generierung eingesetzt. Die Sicherheitsdokumentation wird erst gegen Projektende erstellt.

In unserem Beispiel wurde die Abstimmung der Software-Modelle zur Code-Generierung mit der Sicherheitsfunktionalität in der Design-Phase vernachlässigt. Dadurch konnten die abstrakten Modelle und die generierte Dokumentation die Umsetzung der Sicherheitsmechanismen nicht ausreichend nachweisen. Eine wochenlange manuelle Nachdokumentation der Software wurde notwendig, wesentlich erschwert durch die Unübersichtlichkeit des generierten Codes.

Hinterfragen bei der Dokumentation:

- Wurde die notwendige Dokumentation der Sicherheitsfunktionalität beim Design berücksichtigt?
- Deckt die automatisierte Dokumentation die Sicherheitsaspekte ausreichend ab?

2 FAZIT

Security ist keine gekapselte Funktionalität – die Einbettung in den gesamten Entwicklungsprozess ist entscheidend:

- Viele Teile der Software-Entwicklung tragen zur Sicherheit bei – oder eben nicht.
- Insbesondere: Isolierte Fokussierung auf Sicherheit in Spezifikation / Architektur / Test ersetzt nicht das Sicherheitsbewusstsein und das Security-Know-how aller Beteiligten.

→ Security by Design*er*

Wie kann man das umsetzen? Ideal wäre es, den Blick auf die Sicherheit der Anwendung als selbstverständlichen Bestandteil der Entwicklung zu etablieren:

So wie die „Handwerkerehre“ in der Entwicklung einen Einsatz veralteter Methoden (z. B. „GOTO“ statt Schleifen) oder Bibliotheken oder eine unvollständige Implementierung der Funktionalität verhindert, sollte sie auch die Erfüllung der Sicherheitsziele in allen Entwicklungsphasen sicherstellen. Genau wie die spezifizierte Funktionalität allen Beteiligten vollständig geläufig ist, müssen auch die Sicherheitsziele rechtzeitig definiert und allen Beteiligten bekannt sein.

„Wenn mir keiner sagt, dass ich eine aktuelle Krypto-Bibliothek verwenden muss, kann ich doch auch eine alte Version nehmen?“

„NEIN, das wäre doch so PEINLICH wie der Einsatz von FORTRAN 77!“

Weitere Links:

Ein ähnlicher Ansatz wie hier findet sich in:

https://www.microsoft.com/en-us/securityengineering/sdl/practices

Die OWASP Foundation legt einen stärkeren Fokus auf die Implementierung:

https://wiki.owasp.org/index.php/Security_by_Design_Principles

achelos GmbH
Vattmannstraße 1
33100 Paderborn

+ 49 5251 14212-0

www.achelos.de