Technische Universität Ilmenau 3. Juni 2024

Vorlesung

Security Engineering

Dr. Peter Amthor

NeXTeX

Vorlesungsmitschrift von
Adrian Schollmeyer

Inhaltsverzeichnis

1. Introduction

1.1. What is Security Engineering?

- Zugriffssteuerung
- Methoden zur Absicherung von Software
- Spezialisierte Methoden für Softwareentwicklung; zieht sich durch alle Schritte des Softwareentwicklungsprozesses, ist also ein *Cross-Cutting-Concern* in der Softwareentwicklung: Security Requirements, Security Model, Security Testing (z. B. SAST), SecDevOps

Konkret ist Security Engineering eine Spezialform der Softwareentwicklung, bei der besondere security-spezifische Teilmethodiken/Artefakte/Modelle benutzt werden, um Sicherheitsanforderungen umzusetzen. Ziel ist die Garantie nichtfunktionaler Sicherheitsanforderungen. Dies umfasst alles von der Technologie bis zum Projektmanagement.

Beispiel: Vertraulichkeit als Anforderung muss zu jedem Zeitpunkt berücksichtigt werden. Im Design müssen verschiedene Benutzerrollen vorgesehen werden, die mit Authentifizierung und Autorisierung umgesetzt werden müssen. Die korrekte Funktion dieser muss getestet werden.

Damit dieser Prozess beherrschbar bleibt, ist Automatisierung mit entsprechenden tools zwingend notwendig.

1.2. Why do we need Security Engineering?

Egal ob Code offen zugänglich oder versteckt ist, sollten wir davon ausgehen, dass Angreifer auf allen Phasen des Softwareentwicklungsprozesses unterwegs waren.

Angreifer laufen den Softwareentwicklungsprozess normalerweise rückwärts ab:

- Ausnutzung einer laufenden Software (z. B. infizierter E-Mail-Anhang wird geöffnet)
- Privilege Escalation: Einsatz eines Programms, um mehr Berechtigungen zu erlangen
- Implementierung von Schadcode auf der Opfer-Infrastruktur, um den Zugang dauerhaft zu erhalten.
- Entwurf von Zielen, die der Angreifer auf dem Opfer-System erreichen will

Wenn jedoch formal gezeigt werden kann, dass die Umsetzung von Sicherheitsanforderungen entlang des Softwareentwicklungsprozesses formal verifizierung lässt, kann man zeigen, dass es nicht am Sicherheitsmodell gelegen haben kann, wenn etwas schief geht. Dies ist allerdings schwierig bis unmöglich zu beweisen.

2. Model Engineering

Der Schritt der Modellierung wird jetzt aufgeweitet in mehrere Teilschritte:

- **Security Model**
- Iterativer Prozess zur Analyse und Verbesserung des Security Model
- ...
- UML

Ein zentraler Punkt des Security Engineering ist das Model Engineering.

2.1. Security Engineering Workflow

In diesem Workflow werden neben dem üblichen Modell basierend auf funktionalen Anforderungen auch eine Security Policy in die Modellierung aufgenommen. Eine **Security Policy** wird aus den Sicherheitsanforderungen erstellt und legt Regeln fest, um dieben diese Sicherheitsanforderungen umzusetzen. Die Security Policy wird nicht auf Basis natürlicher Sprache definiert, sondern mit einem formalen Sicherheitsmodell, welches sich später analysieren und idealerweise auch verifizieren lässt. So kann sichergestellt werden, dass bestimmte Vertraulichkeits- und Integritätsverletzungen nicht auftreten können.

Beispiel: beim Aufbau einer Patientendatenbank gehört es zur Security Policy, dass Rollen und Regeln festgelegt werden sollen, wer welche Daten lesen/schreiben darf (Zugriffssteuerungsregeln). Nicht zum Modell würde üblicherweise gehören, wie genau die Datenbank die Daten speichert.

2.2. Models We Know

ACF/ACM Access Control Function / Access Control Matrix. Es gibt Subjekte, Objekte und Operationen, die in Relationen zueinander stehen. Subjekte können Operationen auf Objekten ausführen. In einer ACM wird aufgetragen, welche Subjekte Operationen auf welche Objekte ausführen düfen. Formal ist eine ACM $m : S \times O \to \mathfrak{P}(Op)$, wobei S die Menge aller Subjekte, O die Menge aller Objekte und Op die Menge aller Operationen ist.

ACF/ACM ist abstrakt und extrem mächtig. Alle in der Vorlesung vorgestellten und in Betriebssystemen eingesetztn Modelle basieren auf ACF/ACM bzw. lassen sich darauf zurückführen. In der Praxis sind Spalten der ACM häufig in Metadaten (z. B. Berechtigungsinformationen in inodes des Dateisystems) hinterlegt.

HRU (Harrison, Russo, Ullman). Die ACM wird in einem Automaten verpackt, um auch Zustand bei der Berechtigungsprüfung zu berücksichtigen. Der Zustandsautomat kann vom Startzustand q_0 in andere Zustände wechseln, die ihrerseits wiederum mit einem **STS** (State Transition Scheme) in andere Zustände wechseln können. Die Zustände wiederum enthalten dann jeweils eine Menge von Subjekten, Objekten und eine dazugehörige ACM. So kann man bspw. abbilden, dass neue Nutzer angelegt werden können, wodurch die wirksame ACM um ein Subjekt erweitert wird (sprich: ein Zustandübergang). Dies ermöglicht Änderungen der Sicherheitspolitik in Reaktion auf Zustandsänderungen.

RBAC Role-base Access Control. Hier werden Benutzer durch Rollen abstrahiert, die die Berechtigungen mehrerer Nutzer auf einmal definieren.

2.3. HRU Model Safety

Eine Eigenschaft des HRU-Modells ist **Safety**. Es ist safe, wenn es von einem gegebenen Ausgangszustand unmöglich ist, durch Zustandsübergänge ein bestimmtes Recht in die ACM einzutragen, was nicht bereits im Ausgangszustand vergeben ist. Es darf sich kein Recht irgendwohin in die ACM „ausbreiten“.

Auf dem HRU-Modell ist die Frage, ob man irgendwie von einem Startzustand q_0 ein bestimmtes Recht erlangen kann, äquivalent zu der Frage, ob man vom Startzustand q_0 einen Zustand erreichen kan, in dem dieses Recht gegeben ist.

2.4. Analyzability Tradeoff

Modelle klönnen danach klassifiziert werden, wie gut sie sich analysieren lassen bzw. wie gut sich beliebige Semantiken der Security Policy modellieren lassen. Ein Beispiel für ein sehr ausdrucksstarkes Modell ist RBAC, bei dem man einfach eine Liste von Rollen mit erlaubten Operationen auf Objekten hinterlegt (leicht verständlich).

Allgemein gibt die **Ausdrucksstärke** an, wie nah das Modell an der Semantik der Security Policy ist. Bei den weniger ausdrucksstarken Modellen ist es meist deutlich einfacher, diese auf ihre Sicherheit zu analysieren. Allgemein ist es hier ein Tradeoff zwischen Ausdrucksstärke und Analysierbarkeit.

2.5. The Take-Grant Model

Ziel war es, ein möglichst domänenspezifisches Modell zu erstellen, um Sicherheitsanforderungen für einzelne Anwendungsbereiche zu erstellen. Das **Take-Grant Model** wurde speziell für die Anwendungsdomäne Betriebssystem und Access Control entworfen.

Wichtig! Generell ist Safety *nicht* entscheidbar. Take-Grant zeigt uns hingegen eine untere Grenze der Entscheidbarkeit, indem ein minimales Modell gebaut wird, dessen Safety-Eigenschaften noch in Polynomialzeit entscheidbar sind.

Im Take-Grant-Modell ist ein kompletter Graph Teil des Zustands. Zusätzlich ist Teil die Menge R, die **Rewriting Rules**, die erklären, wie sich Zustände verändern dürfen. Dies entspricht δ im HRU-Modell.

Die Frage ist jetzt, ob von einem Ausgangszustand q_1 mit Graph G_1 ein Pfad existiert, der diesen Graphen mithilfe der in R definierten Regeln ein Graph G_n erzeugt wird, der eine (sicherheitsrelevante) Eigenschaft X aufweist.

Im Take-Grant-Model werden Subjekte und Objekte zu **Entitäten** zusammengefasst.

2.5.1. Graph Rewriting Rules

take Delegation (empfangend).

Kopieren von Rechten von einer Entität auf eine andere. Wenn also Ann take auf Bob hat und Bob read auf Doc hat, kann Ann sich read auf Doc geben (das Recht erlangen).

grant Delegation (gewährend).

Eine Entität darf Rechte an eine andere Entität zu geben, die sie selbst hat. Wenn also Ann read auf Doc hat, darf Ann das Recht read auf Doc an Bob übertragen.

create Eine Entität darf neue Objekte mit optionalen Rechten erstellen. Problem hierbei ist, dass das Modell nicht abdeckt, dass möglicherweise bestimmte Berechtigungen nötig sind, um entsprechende Entitäten erstellen zu können (Beispielsweise ist das Erstellen von Dateien im Betriebssystem daran geknüpft, dass Schreibberechtigungen für das Verzeichnis existieren). Dies ist jedoch per Design so, da Take-Grant nur ein minimales Modell sein soll.

call Eine Entität darf ein bestimmtes Programm ausführen (sprich: ein neues Subjekt erstellen). Dabei kommt ein neuer Knoten hinzu (ein Prozess wird erstellt). Dieser hat read auf das Image des Programms (er darf den eigenen Code lesen) und er darf auf Argumente mit den gleichen Rechten zugreifen wie die Entität, die das Programm gestartet hat. Diese Semantik ist bereits sehr spezifisch auf Betriebssysteme ausgelegt.

Beispiel: ein erstellter (Kind-)Prozess darf auf eine gemeinsame Systemressource zugreifen.

remove Eine Entität darf Kanten löschen (Umkehroperation zu grant). Dies ist die einzige **nicht-monotone** Regel, d. h. sie kann als einzige Operation den Graphen verkleinern.

2.5.2. Achievements

Es enthält Operation mit mehr als einer Vorbedingung und mehr als einer Nachbedingung. Dies bedeutet bereits, dass Safety im HRU-Modell nicht allgemein entscheidbar ist. Durch die Spezifität des Take-Grant-Modells ist es jedoch in linearzeit entscheidbar,

ob das Modell safe ist, d. h. ob sich ein Zustand vom Ausgangszustand erreichen lässt, welcher nicht gewünscht ist.

In der Praxis ist Take-Grant allerdings auch nicht sehr ausdrucksstark, da man nur auf fünf Operationen beschränkt ist.

Es zeigt sich also, dass der Entwurf eines sehr spezifischen Modells zu entscheidbaren Safety-Eigenschaften führen kann. Nachteil ist natürlich das Problem, dass es sehr Domänenspezifisch ist. Bereits kleine Änderungen am Modell (bspw. zusätzliche Regeln) können schnell dazu führen, dass der Zustandsraum und damit die Komplexität des Zustandsgraphen stark (bspw. exponentiell) wächst bzw. sogar zu Turing-Vollständigkeit führt (was die Entscheidbarkeit erschwert). Es ist beim Entwurf des Modells auch darauf zu achten, ob das Modell tatsächlich die nötige Analysierbarkeit erzielen lässt.

2.6. The Typed Access Matrix Model (TAM)

2.6.1. Beschreibung

Ziel ist hier, eine generischere (größere) Ausdrucksstärke als Take-Grant zu erzielen und dabei trotzdem analysierbar zu bleiben. TAM wird auch im Gegensatz zu Take-Grant tatsächlich in der Praxis (bspw. in SELinux) umgesetzt.

Ide Idee ist die Ausnutzung eines Typsystems, um Einschränkungen auf Befehlsparametern herzustellen.

Das Modell erweitert die ACM um weitere Spalten für Subjekte. Dies erlaubt die Rechtezuweisung von Subjekten auf Subjekte. Subjekte werden als Teilmenge von Objekten angesehen. Dadurch lassen sich bspw. Administrationsszenarien wie die Erstellung/Löschung von Benutzern modellieren. Zur Unterscheidung werden die Elemente von $O \setminus S$ als **reine Objekte** bezeichnet. Zusätzlich erweitert TAM das HRU-Modell um ein Typsystem, was jedem Objekt (und damit auch jedem Subjekt) einen Typen $t \in T$ mittels einer Funktion $type : O \to T$ zuweist.

$$Q_{TAM} = \mathfrak{P}(S) \times \mathfrak{P}(O) \times TYPE \times M \tag{2.1}$$

$$q_{0,TAM} = \langle S_0, O_0, type_0, m_0 \rangle \tag{2.2}$$

$$type_0 : O_0 \to T \tag{2.3}$$

Zu beachten ist, dass T und R bei TAM **statisch** sind. Das bedeutet, dass sich diese Mengen zur Laufzeit (beim Zustandswechsel) *nicht* ändern. Was sich jedoch ändern kann, sind die Typen, die Objekten zugewiesen sind (d. h. der Typ eines Objekts kann sich ändern).

Die Prüfung im STS bei TAM wird erweitert um Typprüfungen $type_q(x_i) = t_i$. Die Rechteprüfung aus dem HRU-Modell wird unverändert übernommen. Somit lassen sich bei der Regelprüfung Operationen insofern kontrolleren, dass deren Objektparameter bestimmte Typen haben müssen. So kann erzwungen werden, dass ein Recht nur auf Objekte bestimmten Typs angewendet werden darf.

Die Nachbedingungen in δ sind als Konkatenation von Primitiven notiert. Diese sind vom Modell konkret vorgegeben:

- Create subject x_s of type t_s (Subjekt erstellen)
- Create object x_o of type t_o (Objekt erstellen)
- Enter r into $m(x_s, x_o)$ (Recht vergeben)
- Delete r from $m(x_s, x_o)$ (Recht entziehen)
- Destroy subject x_s (Subjekt löschen)
- Destroy subject x_o (Objekt löschen)

Diese Operationen sind isomorph zu Operationen aus dem HRU-Modell.

2.6.2. TAM Safety

For ein gegebenes TAM-Modell im Zustand q und Recht $r \in R$ gilt es zu entscheiden, ob es eine endliche Sequenz von Eingaben σ^* gibt, sodass unter Anwendung des STS kein Zustand $q' \in \delta^*(q, \sigma^*)$ eine Rechteausbreitung enthält.

Die Frage ist also, ob wir eine Folge Operationen finden, mit der es möglich ist, dass ein Subjekt ein bestimmtes Recht erhält, welches nicht bereits im Ursprungszustand vergeben war. Dabei stellt sich natürlich auch die Frage, ob alle Rechteausbreitung schlimm sind. In der Praxis ist natürlich nicht jede Rechteausbreitung schlecht, jedoch sind Safety-Fragen, bei denen nur bestimmte Zellen der ACM betrachtet werden müssen, lediglich Konkretisierungen der hier untersuchten Fragestellung.

2.6.3. Analysierbarkeit

Generell sind TAM-Modelle unentscheidbar (da sie eine Obermenge von HRU sind). Durch Hinzufügen weiterer Bedingungen lässt sich jedoch Entscheidbarkeit, bei weiterer Einschränkung sogar in Polynomialzeit, erreichen:

Monotonität

Mono-Konditionalität Jede Rechteprüfung prüft höchstens ein Recht

Ternär Jeder Befehl arbeitet mit höchstens drei Objekten.

Azyklisch Der TCG enthält keine Schleifen (Achtung: Eigenschleifen zählen auch!).

Der **Type Creation Graph** (TCG) hat Typen als Knoten und Kanten dazwischen. Da es nur endlich viele Typen gibt und die Menge statisch ist, kann der Graph nicht (in der Anzahl Knoten) wachsen.

Es werden nun Beziehungen zwischen Typen aufgestellt:

- Wenn ein Subjekt vom Typ t_1 ein Objekt vom Typ t_2 erstellen kann, ist t_1 **parent type** von t_2, t_2 ist **child type** von t_1.
- Im TCG werden Kanten von Elterntypen zu Kindtypen eingezeichnet. Dabei kann es auch Eigenkanten geben.

Durch Umformulierung von TAMs lassen sich viele praktisch relevante Szenarien so Umformulieren, dass die Modelle ternär und azyklisch werden. Das macht Entscheidbarkeit für TAMs in der Praxis möglich, bspw. für SELinux. TAM wird gerne als obere Schranke für die polynomielle Komplexität von Sicherheitsanalysen benutzt.

Offene Fragen:

- Was bringt es, ein Typsystem einzuführen?
- Warum ist Azyklizität des TCG für die Safety-Entscheidbarkeit ausreichend?

2.7. Role-Based Access Control (RBAC)

Lücke

2.8. Loose Ends

(1)

(2) Dynamische Komponenten (GitClub, nicht RBAC96):
   - Organisationen und Nutzer können hinzugefügt werden
   - Protection State enthält User, Objekte, UA (Relation User-Rolle)

(3) Safety: Es stellt sich die Frage, ob ein Element in UA erstellt werden kann, welches nicht existieren soll (bspw. ob ein Admin Rechte an Nutzer vergeben kann, die er nicht vergeben können sollte).

   Allgemein ist die Frage, ob einem Nutzer eine Rolle gegeben werden kann, die ihm nicht gegeben werden soll.

$$\begin{aligned} \mathit{safe}(q, r) \iff & \exists \sigma^* \in \Sigma^* : \exists q' \in \delta^*(q, \sigma^*) : \\ & \forall u \in U_q \cap U_{q'} : \langle u, r \rangle \in UA_{q'} \implies \langle u, r \rangle \in UA_q \end{aligned}$$

   Natürlich stellt sich jetzt noch die Frage der Safety Analyzability, also ob es (algorithmisch) möglich ist, die Safety zu prüfen. Leider lässt sich eine Evolution von Zuständen bei RBAC nicht richtig ausdrücken, wodurch eine Analyse eines Automaten nicht möglich ist. Es ist nicht einmal möglich, σ^* und δ^* richtig auszudrücken.

   Allerdings kann RBAC sehr gut ausdrücken, welche Subjekte welche Rechte haben. RBAC96 wurde nicht für eine gute Analysierbarkeit entwickelt, sondern für gute Implementierbarkeit. Für bessere Analysierbarkeit muss RBAC noch um einen Automaten erweitert werden.

2.9. Attribute-Based Access Control (ABAC)

ABAC und RBAC möchten das Problem der schlechten Skalierbarkeit lösen und einen geringen Abstraktionslevel erzielen. ABAC erweitert dabei RBAC, um offene Systeme zu modellieren. Das Prinzip der Rolle wird auf beliebie Attribute/Metainformationen generalisiert, die an Benutzer/Objekte angeheftet werden können (bspw. ein Rollen-Attribut in einem Zertifikat, ein Login-Timestamp, etc.).

Beispiel Kindersicherung: Das Kind (Subjekt) hat als Attribut ein Alter. Der Film (Objekt) hat als Attribut eine Altersbeschränkung. Die Berechtigung, den Film zu schauen, hängt nur vom Alter und der Altersbeschränkung ab.

2.9.1. CoreABAC

- Es gibt mehrere Subjektattribute und Objektattribute. A_S und A_O enthalten alle möglichen Kombinationen von allen Attributen als Tupel.
- Abbildungen ordnen Subjekt- und Objektattribut-Tupel Subjekten und Objekten zu.
- Autorisierungs-Ausdrücke können beliebige Ausdrücke der Prädikatenlogik erster Ordnung sein und werden als aussagenlogischer Ausdruck aufgeschrieben. Die ACF entscheidet nun, ob es ein geeignetes Prädikat gibt, welches für gegebenes Subjekt, Objekt und Operation *true* zurückgibt.
- AAR ordnet Prädikate Operationen zu, sodass für eine Operation ein Prädikat entscheidet, ob die Berechtigung für eine Kombination aus Subjekt und Objekt vergeben wird.

RBAC lässt sich in CoreABAC simulieren, indem Sessions/Rollen als Subjekte und Attribute modelliert werden. Beispielsweise erstellen User ihre Sessions, die wiederum mit Rollen attributiert sind.

Kosten

(1) Eigentlich sollte man möglichst wenig beliebige boolsche Ausdrücke benutzen, da diese nur sehr schwer händisch auf ihre Safety prüfen lassen. Hier stellt sich jetzt die Frage, wie das umsetzbar ist.

In Dacquiri [2] wird bspw. zwischen Entities und Guards unterschieden. Es wird definiert, welche Entitities Zugriff auf was haben sollen. Guards definieren, welche Eigenschaften Objekte haben sollen (bspw. Besitzer o. Ä.). Funktionen werden in einem Kontext ausgeführt, die von einer Berechtigungsprüfung umgeben sind. Innerhalb des Kontexts kann auf nichts zugegriffen werden, was nicht zuvor spezifiziert (und damit auf Berechtigung geprüft) wurde. Zugriffe außerhalb des Kontexts werden bereits zur Compilezeit erkannt und verhindert. Dacquiri ist insofern beschränkt, dass Policies **aufzählbar** sind, d. h. dass es nur endlich viele Werte gibt. Dies erlaubt effiziente Implementierbarkeit, schränkt aber die Ausdrucksstärke ein.

(2) Auch in CoreABAC fehlt der Zustandsautomat wie in RBAC. Also stellt sich auch hier wieder die Frage, welche Teile des Modells sich dynamisch ändern können.

- Subjekte, Objekte sind fast immer dynamisch.
- Attribute sollten sich ändern können (bspw. Änderungen des Kontomodells und der damit verbundenen Berechtigungen, Sessions ändern sich und werden über Attribute abgebildet)
- Je nach Anwendungsfall können möglicherweise auch A_S und A_O dynamisch sein.
- AAR dynamisch zu machen, ist vom Anwendungfall abhängig sinnvoll oder nicht. In so einem Fall muss man zusätzlich Zugriffe definieren und regeln, die die AAR ändern können (also administrative Zugriffe). Solche Überlegungen sind vor allem bei Mandatory Access Control wichtig.

(3) Die Saftey-Frage ist hier, ob die Attributierungsfunktion an einer bestimmten Stelle einen bestimmten Wert annehmen kann. Wäre allerdings die AAR dynamisch, muss man jedoch auch eigentlich entscheiden, ob die Prädikate nicht in einen „gefährlichen“ Zustand gebracht weden können. Allgemein ist die Safety-Frage hier nicht entscheidbar.

HRU lässt sich recht einfach mit ABAC emulieren. Also ist die Safety Analyzability mindestens so schwer die für HRU.

$\mathbf{ABAC}_\alpha$

ABAC_α ist ein *dynamisches* Access Control Modell. Statt Attribute als Vektoren darzustellen, gibt es eine Menge Attributierungsfunktionen, die User (Principals) und Subjekte (jetzt unabhängig von Usern) auf Attribute abbilden. Dies macht auch die Simulation von RBAC in ABAC_α einfacher, da Rollen als Attributierungsfunktion und Sessions als Subjekte modelliert werden können. Durch die Auftrennung von Benutzern und Subjekten ist es wieder möglich, bspw. für Sessions oder Intentionen des Benutzers Berechtigungen zusätzlich zu beschränken. Zu guter letzt gibt es natürlich auch für Objekte Attributierungsfunktionen. Die Attributierungsfunktionen müssen nicht zwangsläufig jeweils nur einen Wert pro Attribut zuordnen, sondern auch eine Menge von Werten. Hinzu kommt eine Funktion, die anzeigt, welches Subjekt durch welchen User erzeugt wurde. Statt der Zuordnung auf Operationen, gibt es eine Menge von Autorisierungsfunktionen für jede Permission. Das bedeutet jedoch auch, dass jede Permission zu genau einer Operation gehören muss.

ABAC_α ist sogar safety-entscheidbar. Hier zeigt sich wieder, dass eine Reduktion der Ausdruckskraft auch Entscheidbarkeit gewonnen werden kann.

2.10. Core-based Security Models

Kernidee ist die Bildung eines Kern-Sicherheitsmodells, welches aus gemeinsamen Eigenschaften anderer Komponenten entsteht. Dieses wird dann um weitere Komponenten

entsprechend des Anwendungsfalls erweitert. Natürlich stellt sich erst einmal die Frage, was die gemeinsamen Eigenschaften der Modelle sind. Das ist für die bisher gezeigten Modelle der Automat $\langle Q, \Sigma, \delta, \lambda, q_0 \rangle$. Dieser lässt sich selbst für die Modelle bilden, die nicht explizit einen Automaten haben (bspw. ABAC).

Erweiterungen des Modells ergeben sich dann im Prinzip aus dem, was nicht bereits Teil des Automaten ist. Diese Erweiterungen müssen jedoch statisch definiert sein, da sie sonst Teil des Automaten wären.

Dynamische Komponenten müssen irgendwie in den Zustand eingebaut werden, bspw. für Objekte und Subjekte. Teilweise sind in den Zuständen hier Teilmengen dieser Obermengen enthalten.

Abschließend muss noch nach Bedarf das STS definiert werden. Hierbei lässt sich auch ein Gefühl dafür bekommen, wie fehleranfällig eine Implementierung sein kann, da insbesondere sehr heterogene Vor- und Nachbedingungen der STS dazu führen kann, dass Programmierfehler leichter passieren können. Es zeigen sich hier also auch Red Flags, die mögliche Probleme in den Anforderungen oder im Sicherheitsmodell andeuten können.

Die Bildung eines Core-basierten Models für ABAC_α ist in Anhang A gezeigt.

2.11. Model Analysis

2.11.1. Model Simulation

Es ist selten möglich, die Safety eines Modells zu entscheiden. Jedoch ist es möglich, die Existenz von Fehlern mithilfe von Simulationen aufzuzeigen. Natürlich muss dafür zunächst wieder definiert werden, was „safe“ bedeutet. Hierzu können die bereits bekannten Definitionen genutzt werden. In der Praxis werden diese Definitionen auch häufig wieder eingeschränkt, um dem Anwendungsfall zu entsprechen. Allerdings kann dies wieder dazu führen, dass das Modell jetzt nicht mehr dem entspricht, was ursprünglich gefordert war.

Allgemein ist die Safety-Frage, ob gewisse Rechteausbreitungen auftreten können, die nicht auftreten sollen. Um dies zu analysieren, stellt sich zunächst die Frage, wo überall die Komponenten des Modells verändert werden, die entscheiden, welche Rechte vergeben sind. Über alle möglichen Zugriffsprüfungen quantifiziert muss definiert werden, wie eine Rechteausbreitung zustande kommen kann.

Bei der Analyse ist es nicht immer möglich, Safety zu entscheiden. Allerdings sind manche Safety-Probleme noch semi-entscheidbar. Der Trick ist, eine relevante Teilmenge von Zuständen zu finden, die das Modell unsicher machen. Somit kann man zwar nicht mehr prüfen, ob ein Modell sicher ist, aber man kann Unsicherheiten im Modell finden. Mit der Analyse lässt sich herausfinden, weshalb es zu einer unzulässigen Rechteausbreitung gekommen sein kann und damit wo der Fehler im Modell ist. Eine komplette Suche in allen Zuständen ist dabei praktisch nicht möglich, da der Zustandsraum theoretisch unendlich und praktisch extrem groß (z. B. 10^{16}) ist. Nach jeder neuen Eingabe vergrößert sich der Suchraum meist exponentiell, was eine naive Simulation zu komplex macht. Stattdessen müssen Heuristiken angewendet werden, um Modelle zu analysieren. Lücke

2.11.2. Dissecting Model Safety Expressions

Grundstruktur besteht immer aus vier Komponenten:

- Safety-Prädikat (Recht in Bezug auf den Zustand)
- Quantifikation (Herstellung eines zulässigen Ausführungskontexts mithilfe von Quantoren, bspw. Nutzer, Sessions, etc.)
- Notwendige Bedingung (Erreichung eines „höherprivilegierten“ bzw. „verbotenen“ Berechtigungslevels, muss über dynamische Komponenten argumentieren)
- Konsequenz (Formulieren, dass in so einem Fall das hohe Privileg auch schon im Grundzustand gegeben war)

Das Safety-Prädikat muss dabei über alle Operationsfolgen vom Startzustand q aus argumentieren.

$$
\begin{aligned}
&safe_{\mathrm{ABAC}_\alpha}\left(q', p\right) : \Longleftrightarrow\ \forall s \in S_{q'}, o \in O_{q'} : \exists sa_{q'} \in SA_{q'}, oa_{q'} \in OA_{q'} : \\
&\left\langle sa_{q'}(s), oa_{q'}(s)\right\rangle \models Authorization_p\left(s, o\right) \implies s \in S_q \wedge o \in O_q : \exists sa_q \in SA_q, oa_q \in OA_q : \\
&\langle sa_q(s), oa_q(s)\rangle \models Authorization_p\left(s, o\right)
\end{aligned}
$$

Beachte, dass sich dieses Safety-Prädikat nur auf ein bestimmtes Recht p beschränkt. Bei der Safety-Analyse bleibt es somit grundsätzlich möglich, dass andere Rechte dennoch erlangt werden könnten. Lücke

3. Specification Engineering

Ziel dieses Kapitels ist jetzt das Umsetzen eines Modells in sehr sprachspezifische Komponenten, um eine *Executable Security Policy* zu erhalten. Dafür werden häufig eigene Spezifikationssprachen eingesetzt. Damit ist es möglich, mit wenig Aufwand die formale Modelldefinition in Quellcode zu übersetzen.

3.1. Security Policy Specification Languages

Da Ziel ist es, eine Brücke zwischen dem formalen Modell und der Implementierung zu bilden. Dies ist die Grundlage für formale Verifikation und automatische Codegenerierung.

Auf beiden Seiten diese Brücke gibt es verschiedene Bereiche. Einerseits können verschiedene Modelle (z. B. HRU, TAM, core-basierte Modelle) die Eingabe sein. Andererseits ist das Ergebnis sehr implementierungsspezifisch und die Policy kann an verschiedenen Orten im Tech-Stack durchgesetzt werden (z. B. OS, Middleware, Software). Ein Compiler für die Policy muss dafür auch passende Schnittstellen bereitstellen, die zum entsprechenden System kompatibel sind. Die Sprache muss also generisch genug sein, die Anbindung in verschiedenen Umgebungen möglich zu machen.

3.1.1. DynaMo

DynaMo ist eine Spezifikationssprache für Core-basierte ABAC- und RBAC-Modelle, also Modellen, die mit der Attributierung von Entitäten arbeiten. Unter DynaMo können auch weiter Zustandsautomaten benutzt werden, sodass die Analyse dynamischer Eigenschaften weiter möglich bleibt. Die Sprache unterstützt nicht nur die üblichen Subjekte, Objekte, Principals, etc., sondern auch die spezifikation des STS.

DynaMo erlaubt die Wiederverwendung und Vererbung bestehender Modelle, um neue Modelle zu spezifizieren. So können auch bestehende Analyseergebnisse wiederverwendet und Modelle iterativ weiterentwickelt werden. Die DynaMo-Spezifikation kann zwei Typen haben.

Eine *Model Class* beschreibt eine ganze Modellfamilie, also eher ein Modellkalkül als ein spezifisches Modell für einen einzelnen Anwendungsfall. Diese bestehen aus den mathematischen Strukturen für das Modell sowie Vor- und Nachbedingungen für Zustandsübergänge. In Programmiersprachen entspricht eine Model Class etwa einer abstrakten Klasse.

Die Anwendung einer Model Class auf einen spezifischen Anwendungsfall wird in einer *Model Instance* definiert. In diesem wird der State Space, das STS und die Extensions

definiert. Eine *Model Instance* muss dabei von einer Model Class vererben und nutzt dann die Features der Model Class.

DynaMo wird mit einem Transpiler in andere Sprachen (C++, Rust) übersetzt. So kann das Modell direkt in die Software integriert und verwendet werden. Außerdem kann das Modell als XML für die Analyse in WorSE exportiert werden.

DynaMo ermöglicht direkt die Definition der Komponenten eines Core-basierten Modells. Die Komponenten des Core-basierten Modells (State-Space, Eingabevektor, STS, Startzustand, Extensions) können direkt in der Model Class aufgeschrieben werden. Das STS wird definiert, indem für jede Operation Vorbedingungen als `pre` und Nachbedingungen als `post` definiert werden. Argumente für das STS werden mit Datentypen belegt, die im Eingabevektor definiert wurden. Die Vorbedingungen sind immer ein einziger Ausdruck, während Nachbedingungen Blöcke aus sequentiell angewandten Zustandsänderungen sind.

Die Model Class gibt dann die Komponenten (z. B. Mengen, Relationen) an und definiert die Prädikate und Anweisungen für die Vor- und Nachbedingungen. Für logische Ausdrücke können die üblichen mathematischen Operatoren (wenn auch mit anderen Zeichen) benutzt werden, sodass sich die Definitionen des theoretischen Modells leicht in DynaMo-Ausdrücke umformen lässt.

3.1.2. The SELinux Policy Specification Language

Die *SELinux Policy Specification Language* ist speziell für SELinux, die MAC-Erweiterung des Linux Kernels, entwickelt worden. Kernkonzepte sind Typsierung von Subjekten (Prozesse) und Objekten (Dateien, Sockets, Pipes, etc.) sowie eine Rollensemantik wie bei RBAC und eine Labeling-Semantik ähnlich zu ABAC. So besitzen Entitäten (z. B. Subjekte, Objekte) Labels. Die Entscheidungen der ACF basieren dann nicht auf den konkreten Entitäten, sondern auf den Labels. Labels sind als Attribute Entitäten zugeordnet und beschreiben dabei „Sicherheits-Kontexte“, also bspw. Objektklassen, Typen, Rollen oder auch User-IDs. Damit ist es möglich, sowohl Type Enforcement als auch RBAC-Semantiken in der Policy durchzusetzen.

In der Policy gibt es verschiedene Sprachelemente:

- Typen werden entsprechend der gewünschten Semantik deklariert und werden für fundamentale AC-Regeln benutzt. In der SELinux-Literatur wird teilweise zwischen Typen und Domains unterschieden (wobei die Domain einen Kontext beschreibt, indem ein Programm ausgeführt wird). Diese sind jedoch im Modell auch nur Typen und bewegen sich im selben Namensraum.
- Berechtigungen werden mit `allow`-Statements definiert, die den Zugriff vom Subjekt eines Typs auf ein Objekt eines anderen Typs und einer Objektklasse mit bestimmten Operationen erlaubt. Der Policy-Compiler kann an dieser Stelle bereits prüfen, ob bestimmte Zugriffe auf bestimmte Objektklassen überhaupt zulässig/-sinnvoll sind (bspw. kann ein Socket nicht ausgeführt werden).
- Die Domain eines Prozesses kann nach `allow`-Regeln ermöglicht werden, um bestimmte Aktionen mit anderen Berechtigungen ausführen zu können.

- Entrypoint-Regeln erlauben einem Prozess den Wechsel in einen Typen nur dann, wenn eine Datei mit bestimmten Typen ausgeführt wird. Damit kann unterbunden werden, dass nach einem Typübergang ein Programm ausfgeführt wird, welches eigentlich nicht ausgeführt werden darf.
- Automatische Domain Transitions können durchgeführt werden, wenn Prozesse mit bestimmten Typen von Prozessen mit bestimmten Typen ausgeführt werden.

Da diese Semantiken recht mühsam zu schreiben und schwer nachzuvollziehen sind, gibt es mittlerweile Transpiler, die wiederum anderen Spezifikationssprachen in die SELinux Policy Specification Language übersetzen.

Rollen werden in SELinux auch über Typen abgebildet. Rollen können dabei andere Typen gruppieren und Regeln auf Typen der Rollen können festlegen, welche Zugriffe die Rollen haben. Für Rollen kann auch gesteuert werden, welche Rollenübergänge zulässig sind. Diese Übergänge müssen von Anwendungsentwickler manuell durchgeführt werden.

A. Workshop: Core-Basierte Modelle

In diesem Workshop wird Core-based Modelling genutzt, um $ABAC_\alpha$ als Automat auszudrücken. Zunächst wird sich überlegt, welche Mengen wir benötigen. Davon muss geklärt werden, welche dynamisch und welche statisch sind:

A.1. Definition des Cores

- U (Benutzer, dynamisch)
- S (Subjekte, dynamisch)
- O (Objekte, dynamisch)
- V (Attributwerte, statisch)
- UA (Attributierungsfunktionen $ua : U \to V$, dynamisch)
- SA (Attributierungsfunktionen $sa : S \to V$, dynamisch)
- OA (Attributierungsfunktionen $oa : S \to V$, dynamisch)
- P (Berechtigungsbezeichner, $P \subseteq OP$)
- $AUTH$ (Autorisierungsprädikate)
- $screat$ (Ersteller eines Subjekts)

Die dynamischen Mengen tauchen als (Teilmengen) im Zustand auf. Je nachdem, ob in einem Zustand eine Teilmenge auftauchen soll oder nicht (insbesondere taucht eine Teilmenge auf, wenn sich die Anzahl ändert), müssen Potenzmengen (oder nicht) gewählt werden:

$$Q = \mathfrak{P}(U) \times \mathfrak{P}(S) \times \mathfrak{P}(O) \times \mathfrak{P}(UA) \times \mathfrak{P}(SA) \times \mathfrak{P}(OA) \times SCREAT \tag{A.1}$$

$$q_0 = \langle U_0, S_0, O_0, UA_0, SA_0, OA_0, screat_0 \rangle \tag{A.2}$$

Alle statischen Mengen tauchen im Core auf:

$$ABAC_\alpha = \langle Q, \Sigma, \delta, q_0, P, AUTH \rangle \tag{A.3}$$

Ein paar Definition der obigen Sachen:

$$AUTH = \{ Auth_p : S \times O \to \mathbb{B} \mid p \in P \} \tag{A.4}$$

$$screat : S \to U \tag{A.5}$$

In diesem Core fehlen Constraints, die in $ABAC_\alpha$ vorkommen. Diese können als Extensions definiert werden. Sie werden in der Zustandsüberführungsfunktion aufgerufen. Sie sind aber generell erst einmal eine Menge boolescher Ausdrücke, die sich zur Laufzeit nicht ändern:

$$ABAC_\alpha = \langle Q, \Sigma, \delta, q_0, P, AUTH, CONS \rangle \tag{A.6}$$

$$CONS = \{ConstrSub, ConstrObj, ConstrObjMod\} \tag{A.7}$$

A.2. Definition des STS

Als nächstes muss das STS definiert werden. Die Frage ist dabei zunächst, welche vordefinierten Operationen es gibt:

- CreateSubject (Erstellung von Subjekten, gibt initiale Werte für die Attributierungsfunktionen mit an)
- CreateObject (Erstellung von Objekten, dito)
- DeleteSubject (Löschung von Subjekten)
- ModifySubjectAtt (Modifikation von Subjektattributen)
- ModifyObjectAtt (Modifikation von Objektattributen)
- AddUser (Erzeugung von Benutzern, gibt initiale Werte für die Attributierungsfunktion mit an)
- DeleteUser (Löschung von Benutzern)
- ModifyUserAtt (Modifikation von Benutzerattributen)

Bei der Unterscheidung ist wichtig, weche Sachen Operationen sind (die in der Policy modelliert werden müssen) und was Primitiven sind, die Teil des Models sind. Beachte, dass hier noch keine Operation für die Löschung von Objekten definiert wurde.

Beachte, dass bei der Erstellung von Entitäten direkt Attribute mitgeliefert werden. Dies ist notwendig, da man sonst keine Berechtigungen (auch nicht Berechtigungen zur Veränderung von Attributen) auf diesen Entitäten definieren kann. Bei den administrativen Funktionen (Benutzer erstellen, u. Ä.) werden Operationen ohne Subjekt als Aufrufer ausgeführt. Diese laufen etwas außerhalb des Kontexts von $ABAC_\alpha$ und deren Autorisierungsmechanismen sind im Paper [3] nicht näher definiert. $ABAC_\alpha$ definiert also nur die innere Autorisierungsebene, nicht die äußere (die festlegt, wer Admin ist und wer User).

Man beachte weiter, dass bei den nicht-administrativen Funktionen immer das Subjekt mit als Aufrufer übergeben wird (außer bei CreateSubject, wo das Subjekt kein Aufrufer sein kann). Dies ist notwendig für die Berechtigungsprüfung.

Die Menge der Operationen ist so immer noch nicht vollständig. P muss noch hinzugefügt werden. Diese verändern jedoch niemals den Zustand. So lässt sich die Menge der Operationen in *state-modifying* und *not state-modifying* unterschieden.

Es ergibt sich dann als STS für $op \in P$:

$$\delta\left(q, \langle op, (x_s, x_o)\rangle\right) = \text{if } f_{ABAC_\alpha}\left(x_s, x_o, op\right) \text{ then } true. \tag{A.8}$$

$$f_{ABAC_\alpha}\left(s, o, op\right) = \begin{cases} true & Authorization_p\left(s, o\right) = true \\ false & \text{otherwise} \end{cases} \tag{A.9}$$

Für ABAC_α sind die Vor- und Nachbedingungen bereits in [3] vorgegeben. Einzig veränderlich sind dabei die Constraints, die wieder vom Autor der Policy definiert werden können.

Es zeigt sich also, dass ABAC_α durchaus als Core-basiertes Modell mit Automaten definiert werden kann.

Offene Fragen:

- Wo werden die Attribute der Principals geprüft? Diese werden nur bei der Veränderung von Subjekten durchgeführt, da anderswo die Subjekte mit den erlaubten Subjektattributen als Stellvertreter für Benutzer existieren. Es ist nötig, in ConstrSub die Constraints richtig zu definieren.
- Wie werden Attributmodifikationen autorisiert? Auch hier wird muss über Attribute eine Policy festgelegt werden, nach der bspw. Owner gespeichert werden können, die dann Berechtigungen für die Modifikation haben.

B. Workshop: Model Engineering for a Hospital Information System (HISSEC)

B.1. Schritt 0: Vorbereitung

Zunchst ist zu überlegen, welches Modell als Grundlage gewählt werden soll. ABAC_α bietet sich als Basis an, da sich Rollen gut abbilden lassen. Da gibt es als Mengen:

- U... User-IDs
- O_{EHR}... EHRs
- O_{Sen}... Sensor-IDs
- $C = \mathbb{N}$... Case-IDs
- R... Role IDs
- W... Ward IDs

Je nach Designentscheidungen können Patienten als User über Attribute dargestellt werden, in einer separaten Menge U_{pat} zu erfassen oder eben auch einfach gar nicht als Benutzer zu modellieren (sondern nur als Objekte), da sie nie irgendwelche Operationen durchführen. Die letzte Entscheidung steht zwar in Konflikt zur Sprechweise der Policy, dass Benutzer als Patienten erstellt werden können, aber das erfordert nicht zwingend, dass ein Patient auch wirklich ein User sein muss. Die Führung einer Menge U_{pat} erlaubt bei der Spezfikation der ACF, durch Schnittmengenbildung danach zu filtern, dass Operationen nicht von Patienten durchgeführt werden können (statt Attribute auszuwerten). Die Policy für *read* schreibt vor, dass User, die Patienten sind, gar keinen Zugriff mehr auf EHR-Daten haben, deren Fälle ihnen nicht zugewiesen sind. Das macht eine Abbildung über Objekte unmöglich. Eine Abbildung über ein Attribut ist ausreichend, da eine Zuordnung weiter über andere Attribute möglich ist und die Entscheidung, ob jemand Patient ist, nur für eine einzige Rechteprüfung notwendig ist.

Die Sensoren werden getrennt von den EHRs geführt, da sie finit sind, während die EHRs sich häufig ändern.

Die Case-IDs können zwar theoretisch aus einer dynamischen Menge C entspringen, aber das gibt keinen wirklichen Vorteil, da C eine Attributmenge ist und nicht jeder Wert aus der Attributmenge auch tatsächlich irgendwo zugewiesen sein muss. Daher reicht $C = \mathbb{N}$ aus.

Die Sensoren müssen zur Berechtigungsprüfung ausgelesen werden. Jedoch ist es nur nötig, eine binäre Entscheidung zu treffen, ob der Sensorwert über einen Schwellwert

geht. Der genaue Wert ist für die Rechteprüfung nicht direkt notwendig, sondern kann abstrahiert werden. Entsprechend gibt es auch keine Funktion, die dem Sensor einen Wert gibt.

- $v_{max} : O_{Sen} \to \mathbb{B}$

Attribute können als eine Attributierungsfunktion oder auch als mehrere Funktionen modelliert werden. Mehrere Funktionen zu benutzen macht die spätere Handhabung einfacher. Da alle User immer nur genau einem Ward zugewiesen sein können, wird hier nicht die Potenzmenge benötigt.

- $roles : U \to \mathfrak{P}(R)$
- $wards : U \to W$
- $ucases : U \to \mathfrak{P}(C)$
- $cases : O_{Ehr} \to \mathfrak{P}(C)$
- $patient : U \to \mathbb{B}$

Zusätzlich muss natürlich noch die ACF definiert werden.

B.2. Core Specializations and Extensions

Dynamische Komponenten (Core):

- U
- O_{EHR}
- U_{Pat}
- v_{max}
- $roles$
- $wards$
- $ucases$
- $patient$
- $ocases$

Statisch (Extension):

- O_{Sen}
- R
- W
- C

B.3. Glue

Zum Schluss muss noch geklärt werden, wie Zugriffe geregelt (ACF) werden und Zustände geändert (STS) weden. In unserem Modell lässt sich dies nicht direkt über eine Matrix realisieren, wie es bei TAM gemacht wird. Bei RBAC prüft die ACF Rollenzugehörigkeit und dazugehörige Permissions, was bereits nah an dem HISSEC-Modell liegt.

$$f_{HISSEC} : U \times (O_{EHR} \cup O_{Sen}) \times OP \to \mathbb{B}$$

Diese Funktion ist jetzt noch nicht direkt abhängig vom Zustand, aber es wird Zugriff auf den aktuellen Zustand benötigt, um die korrekten zustandsabhängigen Mengen und Funktionen auswählen können. Dafür muss stattdessen die Funktion λ definiert werden, z. B.:

$$f_{HISSEC}(u, o, \text{fetch}) \iff roles(u) \cap \{\text{nurse}, \text{physician}\} \neq \varnothing \vee (\text{paramedic} \in roles(u) \wedge v_m ax(o))$$

Eine erste Definition für das Lesen von Daten wäre:

$$\begin{aligned} f_{HISSEC}(u, o, \text{read}) \iff & \\ & \exists i, j \in C, \exists u, u' \in U : \\ & \neg patient(u) \wedge \\ & i \in ocases(o) \wedge \\ & i \in ucases(u) \wedge \\ & j \in ucases(u') \wedge \\ & j \in ocases(o) \\ & ward(u) = ward(u') \end{aligned}$$

Hier ist jedoch das Problem, dass die Operation sich eigentlich schon auf einen Case i bezieht, man sich also eigentlich nicht einfach per Existenzquantor einen beliebigen Case herbeten kann. Eigentlich wäre die Lösung jetzt, das Modell zu verfeinern, um die ACF anzupassen, sodass Operationen im Kontext eines Falls stattfinden. Die pragmatische Lösung wäre hier jedoch, tatsächlich einfach Zugriff zu gewähren, solange ein entsprechender Case existiert (sprich: kein Zugriff im Kontext eines Cases).

$$\begin{aligned} f_{HISSEC}(u, o, \text{read}) \iff & \\ & \neg parient(u) \wedge \\ & ocases(o) \cap ucases(u) \neq \varnothing \\ & \exists u' \in U \setminus \{u\} : (\\ & ward(u') = ward(u) \wedge \\ & ocases(o) \cap ucases(u') \neq \varnothing) \end{aligned}$$

Was sich hier zeigt, ist, dass es sich anbieten würde, Autorisierungsprädikate für jede Operation zu definieren, sodass man nicht alle Bedingungen der verschiedenen Operationen zu einer Funktion vereinigen müsste. Das funktioniert allerdings wieder nur

für User-Level-Operationen. Für administrative Operationen reicht die ACF wiederum nicht aus, da bspw. bei der Erstellung von Usern kein Objekt existiert. Die Umsetzung administrativer Operationen muss dann im STS definiert werden.

B.4. Definition

Model $\texttt{HISSEC}_{24}$ is a deterministic automaton $\langle Q, \Sigma, \delta, q_0, O_{Sen}, W, R, C, AUTH \rangle$ where

$$
\begin{aligned}
Q &= \mathfrak{P}(U) \times \mathfrak{P}(O_{EHR}) \times UCASES \times OCASES \times ROLES \\
&\quad \times PATIENT \times WARDS \times VMAX \\
UCASES &= \{ucases^q : U^q \to \mathfrak{P}(C) \mid q \in Q\} \\
OCASES &= \left\{ocases^q : O^q_{EHR} \to \mathfrak{P}(C) \;\middle|\; q \in Q\right\} \\
ROLES &= \{roles^q : U^q \to \mathfrak{P}(R) \mid q \in Q\} \\
PATIENT &= \{patient^q : U^q \to \mathbb{B} \mid q \in Q\} \\
WARDS &= \{ward^q : U^q \to W \mid q \in Q\} \\
VMAX &= \{v^q_{max} : O_{Sen} \to \mathbb{B} \mid q \in Q\} \\
C &= \mathbb{N} \\
R &= \{\text{paramedic}, \text{nurse}, \text{manager}, \text{clerk}, \text{physician}\} \\
W &= \{\text{internal}, \text{surgery}, \text{ICU}, \text{maternity}\} \\
\Sigma &= OP \times X
\end{aligned}
$$

and δ, Σ, q_0 are automaton components as defined by the model core.

Authorization rules:

C. Workshop: Proof Theory

C.1. TUI CW Model

Für jeden Zustand muss geprüft werden, dass er weiterhin read-secure und write-secure ist. Das TUI-CW-Modell ist secure, wenn q_0 secure ist und jeder von q_0 aus durch endlich viele Zustandsübergänge erreichbare Zustaand secure ist.

C.1.1. BLP-BST-like theorem

Wir schauen uns an, was modifiziert werden könnte beim Zustandsübergang. Hier sind das $M, \mathfrak{P}(C), \mathfrak{P}(H)$. Als nächstes werden möglicherweise schädliche Zustandsübergänge (also Dinge, die „schief gehen“ könnten) identifiziert werden:

- Änderungen der ACL
- Änderungen an der History
- Änderungen an den Konfikten zwischen Objekten

A TUI CW model $\langle Q, \Sigma, \delta, q_0, S, O \rangle$ is secure iff

(1) q_0 is secure

(2) The STS δ is built such that for each state q reachable from q_0 by a finite input sequence, where $q = \langle m, C, H \rangle$ and $q' = \delta(q, \sigma) = \langle m', C', H' \rangle$, $\forall s \in S, o \in O, \sigma \in \Sigma$, the following holds:

   (a) Read-security conformity:

   $$r \notin m(s,o) \wedge r \in m'(s,o) \implies \forall o' : \langle o, o' \rangle \in C' : \langle s, o' \rangle \notin H'$$

   Konflikt wird eingetragen:
   $r \in m(s,o) \wedge \neg \left(\forall o' \in O : \langle o, o' \rangle \in C : \langle s, o' \rangle \notin H' \right) \implies r \notin m'(s,o)$

   (b) Write-security Conformity ähnlich, aber hier muss für alle Objekte entsprechnd Zeug gefordert werden.

C.1.2. Proof

Wünschenswert ist jetzt, die Äquivalenz der Ausdrücke mit der TUI-CW-Security zu zeigen.

For $R :=$ „$read \in m(s,o)$“, $R' :=$ „$read \in m'(s,o)$“, $C' :=$ „$\forall o' \in O, \langle o, o' \rangle \in C' : \langle s, o' \rangle \notin H'$“:

$$\underbrace{\left(\neg R \wedge R' \implies C'\right)}_{(a1)} \wedge \underbrace{\left(R \wedge \neg C' \implies \neg R'\right)}_{(a2)} \overset{!}{\iff} R \implies C'$$

Durch Auflösen der Implikation, Auflösen von Klammern und Minimierung der logischen Ausdrücke ergibt sich diese Äquivalenz.

C.1.3. STS for TUI-CW

$\triangleright init_r(s,o) ::=$

IF $\forall o' \in O : \langle o, o' \rangle \in C : \langle s, o' \rangle \notin H$

THEN $H' := H \cup \{\langle s, o \rangle\}$

$m'(s,o) := m(s,o) \cup \{r\}$

$\triangleright init_w(s,o) ::=$

IF

THEN

Alternativ ließe sich vielleicht auch in POST die Konfliktrelation ändern, sodass die Read-Security wieder erfüllt ist. Das würde aber bedeuten, dass man die Konflikte, die die Bedingung verletzen würden, einfach aus der Relation streicht. Das ist aber praktisch nicht wirklich sinnvoll. Ähnlich könnte man auch Elemente aus der History löschen, die Konflikte erzeugen. Auch das ist praktsich nicht sinnvoll.

Hier wurde bereits die History modifiziert, wo nur die Initialisierung der Operationen durchgeführt wird. Eigentlich ist es aber erst nötig, die History zu modifizieren, sobald die Operation tatsächlich ausgeführt wird. Dann muss man aber nochmal Prüfungen für die Lese- und Schreiboperationen definieren.

$\triangleright add_c(o_1, o_2) ::=$

IF $\forall s \in S : \langle s, o_1 \rangle \notin H \vee \langle s, o_2 \rangle \notin H$ THEN $C' := C \cup \{\langle o_1, o_2 \rangle\}$

Beim Einfügen neuer Konflikte kann es auftreten, dass die bisherige History diesen neuen Konflikt verletzt. Hier muss sich entschieden werden, ob solche Konflikte nicht erlaubt werden (sprich: nicht eingetragen werden können) oder ob man in POST die Berechtigungen oder History abändert, dass die Konflikte nicht mehr exisitieren.

Stichwortverzeichnis

Literatur

[1] Peter Amthor. “Security Engineering”.

[2] *dacquiri - Rust*. URL: https://docs.rs/dacquiri/latest/dacquiri/ (besucht am 24. 04. 2024).

[3] Xin Jin, Ram Krishnan und Ravi Sandhu. “A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC”. In: *Data and Applications Security and Privacy XXVI*. Hrsg. von Nora Cuppens-Boulahia, Frédéric Cuppens und Joaquin Garcia-Alfaro. Berlin, Heidelberg: Springer, 2012, S. 41–55. ISBN: 978-3-642-31540-4. DOI: 10.1007/978-3-642-31540-4_4.