# Formale Methoden als Werkzeug für Software Security

Peter X. Schwemmer

WIBOND Informationssysteme GmbH
peter.schwemmer@wibond.de

## Zusammenfassung

Es werden die zugrundeliegenden Schwachstellen einiger schwerwiegender IT-Sicherheitsvorfälle untersucht. Dabei wird gezeigt, dass sich sicherheitsrelevante Abläufe modellieren lassen und formale Methoden existieren, welche Programmanalysen erlauben. Diese Modelle erlauben in Kombination mit formalen Methoden eine Verifikation sicherheitsrelevanter Abläufe. Dabei wird konkret auf das Verifikationswerkzeug TESLA (University of Cambridge) eingegangen. Es wird aufgezeigt, dass Software Security durch Usable Security im Softwareentwicklungsprozess erreicht werden kann und Security by Design eine Kernvoraussetzung für sichere Software ist.

# 1 Motivation

Viele Software-Schwachstellen basieren auf dem Zusammenspiel voneinander abhängiger Anweisungen. Verteilte Systeme, Mobile Computing oder Cloud Computing – viele neue Ansätze führen zu dieser Art voneinander abhängiger Komponenten [Matt00, Delo17]. Dabei kann es durchaus vorkommen, dass eine geänderte, eventuell komplexere, skalierbarere Software-Architektur nach wie vor den gleichen Anwendungsfall abdecken soll. Die Implementierung dieser Anwendungsfälle kann stark variieren [Lüne14].
Das Sicherstellen der spezifizierten Funktionsfähigkeit ist nicht trivial und mit den üblichen Testverfahren oft nicht vollständig abzudecken [Mann03, Schn04, NiNi07, Somm12]. Der Aspekt, dass Software zunehmend verteilter und komplexer wird und als Ganzes gesehen bestimmte Eigenschaften (z.B. Sicherheitseigenschaften) zu erfüllen hat, sollte den Impuls für fortschrittlichere Entwurfs-, Entwicklungs-, Analyse-, Test- und Verifikationsverfahren geben [AWCG+14].

Wie gezeigt wird, basieren einige der folgenreichsten IT-Sicherheitsvorfälle darauf, dass klar einzuhaltende Abläufe nicht umgesetzt wurden – eventuell auch darum, weil diese (parallelen) Abläufe in komplexe Softwarestrukturen gepackt werden, die schwer zu testen und zu verifizieren sind. Bei Androids Stagefright-Schwachstellen erfolgte keine Prüfung nicht vertrauenswürdiger Eingaben [Drak15], Apples Goto-Bug (Schritt in TLS-Prüfung wurde mit *goto* übersprungen) war eine versehentlich vorhandene doppelte Codezeile [Duck14, Reiß14, Appl14]. Der Netzwerkausrüster Juniper vergaß die Prüfung von Zertifikaten und verifizierte lediglich anhand des Ausstellernamens [Böck16]. Es sollen praktikable Verfahren gezeigt werden, mit denen man diese bzw. ähnliche Vorfälle verhindern hätte können.
Anhand eines kurzen Überblickes über Software-Verifikation bzw. -Analyse im Allgemeinen und Beispielen für formale Methoden im Speziellen, soll ein Konzept für im Softwareentwicklungsprozess verankerte Werkzeuge, welche Schwachstellen vorbeugen sollen, aufgezeigt wer-

P. Schartner · A. Baumann (Hrsg.) · D•A•CH Security 2017 · syssec (2017) 350-361.

den. Oft driften Funktionalitäten industrienaher Werkzeuge und Möglichkeiten bzw. Erkenntnisse aus der Forschung weit auseinander, Sicherheit ist meist nur ein Bestandteil im Produktentstehungsprozess. Daher ist es wünschenswert, Tools (z.B. Plug-Ins für bestehende Entwicklungsumgebungen) zur Verbesserung von Software-Sicherheit direkt im Entwicklungsprozess zu integrieren. Diese Tools können Kenntnisse aus der Forschung (Verifikation, formale Methoden, Logik, Testverfahren, ...) für Anwender einfach nutzbar machen, sodass diese sich auf die Entwicklung des Gesamtproduktes konzentrieren können.

Anhand von Praxisbeispielen soll gezeigt werden, wie man sichere Software mit wenig Aufwand realisiert. Dabei wird das Verifikationswerkzeug *TESLA* vorgestellt, was sich der Tatsache annimmt, dass Eigenschaften, welche für die Sicherheit ausschlaggebend sind, von Ereignissen in der Vergangenheit, der Gegenwart und der Zukunft abhängen.

# 2 Programmanalysen

Um Computerprogramme zu analysieren bzw. zu verifizieren [NiNH05], gibt es verschiedene Verfahren, welche sich grundsätzlich in zwei Kategorien einteilen lassen:

**Statische Analysen:** Prüfen, ob die Funktionalität bzw. die Implementierung der Spezifikation entspricht, führt dazu aber keinen Programmcode aus. Es wird rein auf Sourcecode-Ebene geprüft.

**Dynamische Analysen:** Prüfen, ob die Funktionalität bzw. die Implementierung der Spezifikation entspricht und führt dazu entsprechenden Programmcode aus.

Dabei können verschiedene, grundlegende Fehlerklassen erkannt werden [Somm12], welche wiederum Basis für das Auffinden von Software-Schwachstellen sind [Schw15].

**Tab. 1:** Erkennbare Fehlerklassen

| Fehlerklassen | Fehler-Beispiele |
|---|---|
| Datenfehler | Vor der Initialisierung genutzte Variablen<br>Mögliche Verletzungen der Indexgrenzen |
| Steuerungsfehler | Unerreichbarer Code |
| Ein-/Ausgabefehler | Senden nicht anonymisierter Fehlerprotokoll-Daten |
| Schnittstellenfehler | Falsch zugeordnete Parametertypen |
| Speicherverwaltungsfehler | Speicherlecks |

## 2.1 Statische Analyse (Datenflussanalyse)

Mit der Datenflussanalyse [KhSK09] leitet man Informationen über das dynamische Verhalten aus dem statischen Programmcode ab. Dabei wird untersucht, wie Daten zwischen Programmteilen weitergegeben werden und welche Abhängigkeiten so entstehen. Ziel kann beispielsweise eine Analyse der aufgerufenen Funktionen und deren Reihenfolge sein. Häufig wird dabei ein Kontrollflussgraph verwendet.

**Lst. 1:** Programmcode

```
int logType = LOGTYPE_UNDEF;
char* logText = "";
int retCode = checkSomething(logText);
if (retCode) {
```

```
    doOk();
    logType = LOGTYPE_INFO;
} else {
    doNok();
    logType = LOGTYPE_ERROR;
}
log(logType, logText);
```

Der Kontrollflussgraph in Abbildung 1 entspricht vorherigem Programmcode.

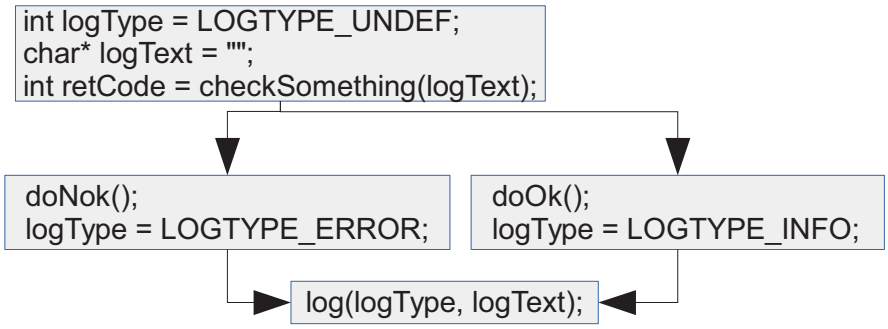


**Abb. 1:** Kontrollflussgraph

## 2.2 Dynamische Analyse

Für eine dynamische Analyse können verschiedene Verifikationswerkzeuge mit unterschiedlichen Darstellungsformen verwendet werden. Grundsätzlich geht es darum, Spezifikationen zu formalisieren bzw. zu modellieren. An dieser Stelle wollen wir durch Falsifikation Fehler mit entsprechenden Gegenbeispielen finden, also auf Ansätze mit Theorem Proving verzichten. So wird z.B. eine Beschreibung mit *Temporal Assertions* (Abschnitt 3.2) formuliert und verifiziert. Vereinzelt besitzen Verifikationswerkzeuge zugehörige Plug-Ins für verbreitete Entwicklungsumgebungen [CPRO17], was die Nutzung erheblich vereinfacht – eine fortschreitende Durchdringung der Entwicklungsumgebungen mit Verifikationswerkzeugen ist daher wünschenswert und soll durch dieses Paper angeregt werden.

# 3 Temporallogik und TESLA

Da das Verifikationswerkzeug TESLA auf der Temporallogik aufbaut, wird diese kurz dargestellt.

## 3.1 Temporallogik

Die temporale Logik (Zeitlogik) [KrMe08] wendet die Modallogik an, um zeitliche Abläufe durch Zustände bzw. Wahrheitswerte erfassen zu können. Damit kann man Situationen beschreiben, in denen die Gültigkeit eines Programmzustandes von vergangenen, aktuellen und zukünftigen Ereignissen und Zuständen abhängt.

Für die Spezifizierung dieser Temporallogiken können verschiedene Beschreibungsmöglichkeiten bestehen – diese variieren oft je nach eingesetztem Verifikationswerkzeug [Kleu09].

Abbildung 2 zeigt – unabhängig von Verifikationswerkzeugen und anhand einer konkreten Formel – die Ausdruckskraft der Temporallogik.

$$\exists\,(checkLogin \rightarrow \diamond denyLogin)$$

- $\diamond$ ['eventually', '(F)inally']
  irgendwann in der Zukunft
- $\exists x : P(x)$ ['there exists', 'there is at least one']
  es existiert zumindest ein ...

**Aussage:**
Es existiert zumindest ein Programmablaufpfad auf dem nach der Funktion $checkLogin$ die Funktion $denyLogin$ aufgerufen wird.

**Interpretation:**
Ein Login muss fehlschlagen können (z.B. wenn die Anmeldeinformationen falsch sind).

**Abb. 2:** temporaler Zusammenhang: Login ablehnen

## 3.2 TESLA

TESLA [AWCG+14] ist ein Werkzeug zur Beschreibung, Analyse und Verifikation zeitlicher Zusicherungen (engl.: temporal assertions), welches es ermöglicht, erwartete, zeitliche Verhaltensweisen und Zusammenhänge direkt im Programmcode zu beschreiben.

### 3.2.1 Funktionsweise

Auf Grundlage der *Temporal Assertions* werden Programmcode-zu-Programmcode Übersetzungen (engl.: instrumentation) durchgeführt, durch welche TESLA das Verhalten zur Laufzeit untersuchen kann. Während der dynamischen Analyse werden Zustandsübergänge in Zustandsautomaten durchgeführt, welche die *Temporal Assertions* repräsentieren. Ungültige Zustände führen zu entsprechenden Fehlerausgaben, wobei konkrete Gegenbeispiele Fehlerfälle gut nachvollziehbar machen. Die Funktion $tesla\_action$, deren Aufrufe durch Instrumentation hinzugefügt werden, verwaltet die Zustandsautomaten (Ereignisse prüfen, Übergänge durchführen, Fehlerzustände erkennen) während der dynamischen Analyse.

### 3.2.2 Beispiel

**Lst. 2:** Ein Login muss fehlschlagen können: Temporal Assertion

```
int tesla_login(struct loginForm* data, struct userCapability* cap){
    //Temporal assertion
    TESLA_WITHIN(tesla_login,
        optional(eventually(call(
            denyLogin(data)
        )))
    );
    int retCode = checkLogin(data, cap);
    if (retCode == LOGIN_ACCEPT) {
        //update user capability
        acceptLogin(data, cap);
    } else {
        //send error msg
        denyLogin(data);
```

```
    }
    return retCode;
}
```

**Lst. 3:** Instrumentation: vereinfacht dargestellter, zusätzlicher Programmcode

```
void denyLogin(struct loginForm* data) {
    //code added by instrumentation:
    tesla_action(
        ACTION_CALL, denyLogin, data
    );
    //original code [...]
}
```

Wenn TESLA die dynamische Analyse für die Funktion $tesla\_login$ abschließt und bei keinem Durchlauf $denyLogin$ aufgerufen wurde, wird eine Verletzung entsprechender *Temporal Assertion* erkannt. Feststellen kann TESLA dies dadurch, dass am Ende der dynamischen Analyse entsprechender Zustandsübergang fehlt, welcher durch $tesla\_action$ ausgelöst werden würde. Letztendlich wird nie der Zustand erreicht, der besagt, dass ein Login abgelehnt wurde. Die Verletzung könnte auch TESLAs statische Analyse [Schw15] erkennen, vorausgesetzt diese kann von vornherein ausschließen, dass $denyLogin$ aufgerufen wird (z.B., wenn diese Funktion gar nicht erreichbar wäre).

# 4 Anwendungsfall I – Zertifikatsprüfung

## 4.1 Problembeschreibung und Auswirkungen

Bei der Juniper-Sicherheitslücke von 2016, wurden Zertifikate lediglich anhand des Ausstellernamens akzeptiert, eine Signaturprüfung fand dann schlichtweg nicht mehr statt [Böck16]. Wir nehmen für unseren Anwendungsfall an, dass ein Fehler entsprechend Apples Goto-Bug ursächlich war [Duck14, Reiß14] und stellen dieses Szenario anhand folgender, vereinfachter Darstellung der Zertifikatsprüfung nach:

**Lst. 4:** Fehlerhaft implementierte Zertifikatsprüfung

```
int X509_verify(X509* cert, EVP_PKEY* pk) {
    int err = 0;
    if ((err = X509_verifyIssuer(cert, pk)) != 0)
        goto fail;
        goto fail;
    if ((err = X509_verifySignature(cert, pk)) != 0)
        goto fail;
    fail : return (err == 0);
    //returns 0 on successful check
}
```

**Fehlerfall:**

*goto* in Zeile 5 wird ausgeführt, wenn nur der Ausstellername stimmt (*if*-Bedingung in Zeile 3 nicht erfüllt), da dann $err == 0$ wahr ist, ist die Verifikation erfolgreich, obwohl das Zertifikat eventuell ungültig ist!

Listing 4 zeigt eine fehlerhafte Implementierung der Zertifikatsprüfung. Geltende Kodierrichtlinien, welche eine Verwendung von *goto* oder Zuweisungen in *if*-Bedingungen untersagen, wurden nicht eingehalten. Dennoch ist dieses Beispiel repräsentativ für die Software-Qualität produktiv genutzter Anwendungen [Appl14]. Code-Reviews kommen häufig nicht zur Anwendung.

**Soll:** Bei einer verschlüsselten Verbindung sendet der Angreifer das Zertifikat der legitimen Webseite, beim Validieren wird geprüft, ob man den privaten Schlüssel besitzt. Wenn die Validierung nicht erfolgreich ist, wird z.B. vom Browser eine Warnmeldung angezeigt.

**Ist:** Der Juniper-Bug ermöglicht einen Man-in-the-Middle Angriff bei verschlüsselten Verbindungen, da das Validieren erfolgreich ist, obwohl der Angreifer nicht im Besitz des privaten Schlüssels ist. Beispielsweise würde eine vom Angreifer nachgebildete Banking-Webseite vom Browser als gültig ausgewiesen.

## 4.2 Datenflussanalyse

In diesem Fall führt eine Datenflussanalyse (Abschnitt 2.1) zu einer nutzbaren Lösung. Der Kontrollflussgraph für die Funktion *X509_verify* ist in Abbildung 3 zu sehen.

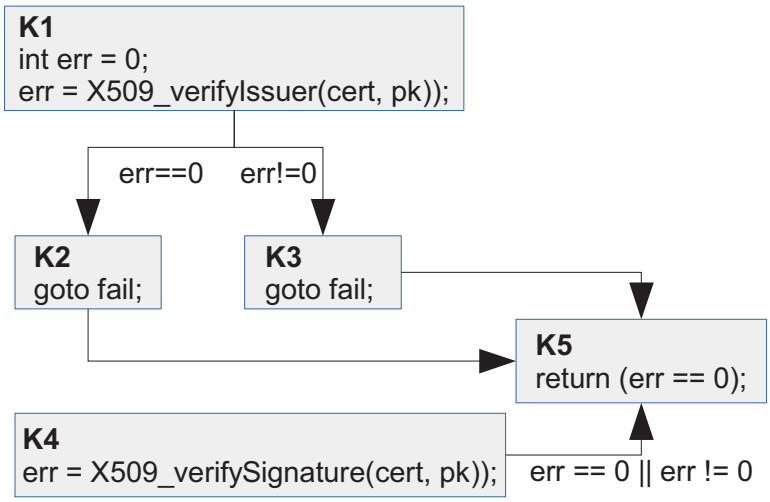


**Abb. 3:** Kontrollflussgraph der Zertifikatsprüfung

Wie zu erkennen ist, ist für die Knoten K2 und K3 jeweils die erste Anweisung ein unbedingter Sprungbefehl in Knoten K5. Damit ist die Funktion $X509\_verifySignature$ in Knoten K4 nicht erreichbar (keine eingehende Kante). Man kann also den Programmcode vereinfachen, indem nur tatsächlich erreichbare Anweisungen verwendet werden.

**Lst. 5:** vereinfachter Programmcode

```
int X509_verify(X509* cert, EVP_PKEY* pk) {
    int err = 0;
    err = X509_verifyIssuer(cert, pk));
    return (err == 0);
    //returns 0 on successful check
}
```

Es wurde also erkannt, dass die Funktion $X509\_verifySignature$ nicht erreichbar ist.

Eine statische Analyse kann – wie hier – potenzielle Fehler aufdecken, häufig sind aber dynamische Analysen nötig, um Aussagen über das konkrete Laufzeitverhalten und konkrete Fehlerfälle tätigen zu können. Beispielsweise wird nicht in jedem Programmausführungspfad auch jede erreichbare Anweisung einer Funktion ausgeführt.

## 4.3 Temporallogik

Viele sicherheitsrelevante Abläufe lassen sich mit Temporallogiken (Abschnitt 3.1) beschreiben, darunter auch unsere Zertifikatsprüfung. Dabei könnte man diese Beschreibungen sehr generalisiert halten, sodass diese auch bei einer Änderung der Softwarearchitektur bestand haben. Aber auch eine Spezifizierung direkt auf Basis konkreter Funktionsaufrufe ist denkbar. Der in Abbildung 4 gezeigte, temporale Zusammenhang hätte gereicht, um die Zertifikatsprüfung ausreichend zu spezifizieren und vor der entsprechenden Schwachstelle zu schützen [BBFL+01].

$$\exists\,(X509\_verifyIssuer \rightarrow \diamond X509\_verifySignature)$$

**Aussage:**
Es existiert zumindest ein Programmablaufpfad, auf dem nach der Funktion $X509\_verifyIssuer$ die Funktion $X509\_verifySignature$ aufgerufen wird.

**Interpretation:**
Es muss möglich sein, ein Zertifikat vollständig zu prüfen.

**Abb. 4:** temporaler Zusammenhang: Zertifikatsprüfung mit Signaturprüfung

Eine *Temporal Assertion* (Abschnitt 3.2), welche die Temporallogik aus Abbildung 4 umsetzt, lautet:

**Lst. 6:** konkrete *Temporal Assertion* zur Problemlösung

```
    TESLA_WITHIN(X509_verify,
        optional(eventually(call(
            X509_verifySignature(cert, pk)
        )))
    );
```

Diese kann beispielsweise direkt nach der Variableninitialisierung in Zeile 2 eingefügt werden.

**Lst. 7:** Angepasster Programmcode inklusive *Temporal Assertion*

```
int X509_verify(X509* cert, EVP_PKEY* pk) {
    int err = 0;
    TESLA_WITHIN(X509_verify,
        optional(eventually(call(
            X509_verifySignature(cert, pk)
        )))
    );
    if ((err = X509_verifyIssuer(cert, pk)) != 0)
        goto fail;
        goto fail;
    if ((err = X509_verifySignature(cert, pk)) != 0)
        goto fail;
```

```
    fail : return (err == 0);
    //returns 0 on successful check
}
```

TESLA führt die entsprechende Zertifikatsprüfung aus (dynamische Analyse), sobald ein Verstoß gegen die *Temporal Assertion* erkannt wird (Prüfung des Ausstellernamens erfolgreich, nachfolgende Signaturprüfung innerhalb von $X509\_verify$ findet nicht statt), wird der entsprechende Fehlerfall gemeldet und kann leicht nachvollzogen werden.

TESLAs statische Analyse könnte diese Tatsache feststellen ohne Programmcode auszuführen, indem es eine Datenflussanalyse entsprechend 4.2 durchführt.

# 5 Anwendungsfall II – Embedded Web-Server

*Interpretation des Internet of Things [Weis99, Bosc17, Micr17]:* Der Mensch soll zunehmend durch intelligente, miteinander kommunizierende Dinge unterstützt werden, wobei der Mensch Gegenstand oder Teilnehmer einer Kommunikation sein kann.

## 5.1 Problembeschreibung und Auswirkungen

In unserem Anwendungsfall soll der Mensch Teilnehmer der Kommunikation sein. Dabei sollen die Dinge – beispielsweise Sensoren, welche eine Produktion überwachen – über eine Weboberfläche angesprochen werden können. Des Weiteren werden Benutzer und Rechte in einer Datenbank verwaltet, in welcher ebenfalls alle Sensordaten zusammenlaufen. Es ist möglich, dass z.B. ein Instandhaltungsmitarbeiter in der Weboberfläche eines Sensors Servicedaten hinterlegt, welche dann in der Datenbank abgelegt werden und anderen Mitarbeitern ebenfalls bereitgestellt werden.

Die Weboberfläche bietet einige potentielle Schwachstellen, wovon wir hier Platz 1 der "OWASP Top 10 – 2017 rc1" [OWAS17] genauer betrachten und ein Beispiel geben:

**Injection:** Eingaben werden direkt als teil einer SQL-Abfrage an die Datenbank verwendet. So beispielsweise wenn der Nutzer seinen Benutzernamen und sein Passwort eingeben muss.

**Hauptursache:** Nicht vertrauenswürdige Daten.

Betrachten wir folgenden exemplarischen Programmcode, welcher zeigt, wie eine SQL-Injection zustande kommt:

**Lst. 8:** Login des Web-Servers

```
int webui_checkLogin(const char* username, const char* password) {
    //[...]
    //build query
    char* secHashPw = secureHash(password);
    char query[BUFFER_QUERY_LENGTH] = {0};
    const char* queryStmt = "select * from Users where name=\"%s\"
        and pwHash=\"%s\";";
    snprintf(query, BUFFER_QUERY_LENGTH, queryStmt, username,
        secHashPw);
```

```
    //send query
    if (mysql_query(conn, query)) {
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(1);
    }
    //[...] process result
}
```

Von der Datenbank werden die Anweisungen im *char-Array* $query$ ausgeführt. Dabei enthält $query$ beispielsweise folgenden ausführbaren Inhalt:

*select * from Users where name="serviceuser8532871" and pwHash="0c83d51d9..";*

Gibt aber ein Angreifer beispielsweise als Benutzername `Bob" OR 1=1;--` ein, werden Zeilen der Tabelle $User$ zurückgegeben und er bekommt unter Umständen Zugang zu einem fremden Konto.

Dazu kommt es, weil die Eingaben des Angreifers als Anweisung verstanden werden und nicht als Daten. Dazu gibt der Angreifer einen Benutzernamen ein, schließt den Datenblock für den Benutzernamen mit `"` und fügt Anweisungen in die *where*-Klausel hinzu, wodurch diese immer zutrifft. Nachfolgende Anweisungen (z.B. die Passwortprüfung) werden mit `--` auskommentiert. Der ausführbare Inhalt von $query$ ergibt sich so wie folgt:

*select * from Users where name="Bob"OR 1=1;*

## 5.2 Ein Modell gegen SQL-Injection

Da eine Eingabe-Validierung fehleranfällig ist, sollte man Programmcode (z.B. SQL-Anweisungen) von Daten (z.B. Benutzereingaben) trennen – bei einigen Datenbank-APIs kann man so beispielsweise eine SQL-Abfrage vorbereiten (den ausführbaren Programmcode hinzufügen) und später Benutzereingaben oder Ähnliches (nicht ausführbare Daten) an die Abfrage binden (engl.: *prepared statement*). So ist beispielsweise sichergestellt, dass egal was der Nutzer eingibt, damit die *where*-Klausel nicht geändert werden kann, geschweige denn Anweisungen auskommentiert werden können. Der Workflow eines *Prepared Statements* ist in Abbildung 5 zu sehen.

## 5.3 TESLA

Die *Temporal Assertion* in Listing 9 beschreibt den Ablauf des *Prepared Statements* für einen erfolgreichen Login auf Seiten des Webservers. Dabei ist die Funktion $webui\_loginPrepStmt$ im Webserver für den gesamten Login zuständig. $TESLA\_NOW$ beschreibt den Zeitpunkt, zu dem die *Temporal Assertion* geprüft wird, die Prüfung erfolgt also bevor der Login erlaubt wird.

**Lst. 9:** *Temporal Assertion* für den Login

```
TESLA_WITHIN(webui_loginPrepStmt,
    TSEQUENCE(
        prepareStatement(queryStmt) == OK,
        provideLogin(loginCallback) == OK,
        loginCallback(loginData) == OK,
```

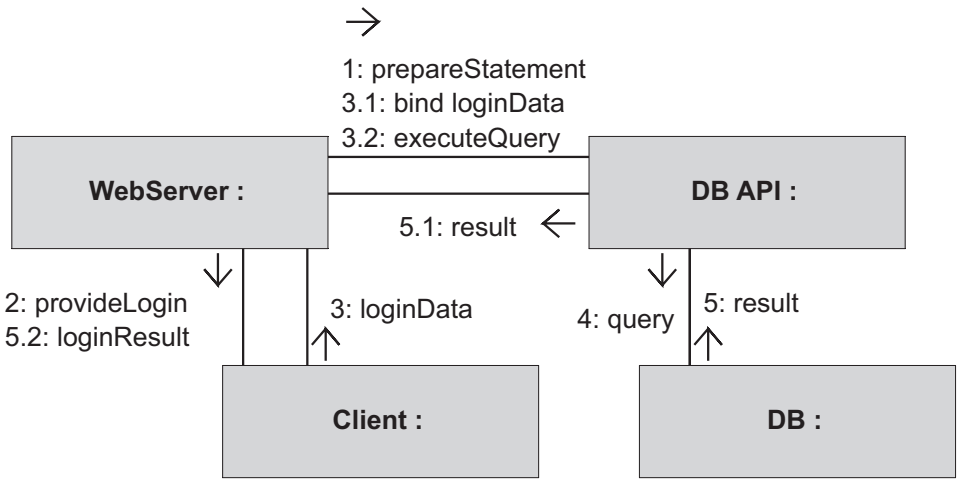


**Abb. 5:** Trennung von Programmcode (1) und Daten (3.1)

```
            bind(loginData) == OK,
            executeQuery(queryCallback) == OK,
            queryCallback() == OK,
            TESLA_NOW, allowLogin()
        )
    );
```

# 6 Fazit und Ausblick

Es wurde gezeigt, wie mit Hilfe von formalen Methoden und (Verifikations-)Werkzeugen, wie dem hier vorgestellten TESLA, Software-Schwachstellen entgegengewirkt werden kann. Dafür werden sicherheitsspezifische Anforderungen formal spezifiziert, wofür sich *Temporal Assertions* sehr gut eignen. Der stärkere Fokus auf IT-Sicherheit würde unter anderem Risiken (Rückruf, Imageverlust, Datenklau, ...) verringern und proaktiv auf Probleme eingehen, statt zurzeit oft nur reaktiv den Schaden zu begrenzen. Gerade Software für kritische Systeme und eingebettete Systeme, ist dafür prädestiniert, neue, fortschrittliche und praktikable Methoden für Software Security zu nutzen. Im Allgemeinen ist eine direkte Verankerung entsprechender Tools im Entwicklungsprozess anzustreben, da sich so auf die Entwicklung des Gesamtproduktes fokussiert werden kann, dabei aber gleichzeitig Methoden zur Steigerung der Software Security zur Anwendung kommen, ohne sich selbst mit deren Komplexität näher beschäftigen zu müssen.

Statische Analysen können grundlegende Hinweise auf potentielle Fehler geben, ob tatsächlich ein Fehler vorliegt ist von Fall zu Fall zu prüfen, auch weil falsche Alarme den Nutzen trüben können. Für konkrete Fehlerfälle ist die dynamische Analyse vorzuziehen, welche ungetrübt Fehler im Ist-Zustand einer konkreten Programmausführung erkennen kann.

Selbst wenn alles wie spezifiziert implementiert ist, liegt noch keine sichere Software vor. Sind im Software-Design bestimmte Sicherheitsmaßnahmen nicht vorhanden oder wurden (bestimmte) Angriffe nicht berücksichtigt, so helfen auch formale Methoden nicht weiter, um die Software sicher zu machen – Stichwort *Security by Design*. Da häufig die UML genutzt wird, um Software zu spezifizieren bzw. das Software-Design zu modellieren, wäre es

wünschenswert, wenn mit der UML bereits die zu prüfenden Eigenschaften formalisiert wären. In der Tat ist das möglich, den es gibt Formalismen und Werkzeuge, um die UML Darstellungen automatisch in eine Darstellungsform zu bringen, welche Verifikationswerkzeuge verstehen und auf Grundlage derer sie ihrer Arbeit nachgehen können [LiPa99, LTMD+09], um durch Falsifikation entsprechende Gegenbeispiele zu finden.

Hinsichtlich TESLA ist festzuhalten, dass in künftigen Versionen auch statische Analysen [Schw15] durchgeführt werden, deren Ergebnisse in die dynamische Analyse einfließen und diese effizienter gestalten. Anzustreben ist auch eine automatische Generierung von *Temporal Assertions* aus UML Diagrammen, sowie ein Plugin für Entwicklungsumgebungen. Dadurch ließen sich Sicherheitsaspekte leichter direkt im Softwareentwicklungsprozess verankern, was der Software Security des Produktes zugutekommen würde.

## Literatur


[Appl14] Apple: Von Apple veröffentlichter Programmcode des Goto–Bugs (2014) (2014), online (abgerufen 01.04.2017): http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c.

[AWCG+14] J. Anderson, R. N. M. Watson, D. Chisnall, K. Gudka, I. Marinos, D. Brooks: TESLA: Temporally Enhanced System Logic Assertions. *In: Proceedings of The 2014 European Conference on Computer Systems (EuroSys)* (2014).

[BBFL+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, P. McKenzie: Systems and Software Verification. Springer (2001).

[Bosc17] Bosch: Was ist das Internet der Dinge? (2017), online (abgerufen 31.05.2017): https://www.bosch-si.com/de/unternehmen/ueber-uns/internet-der-dinge/iot.html.

[Böck16] H. Böck: Juniper vergisst Signaturprüfung (2016), online (abgerufen 01.04.2017): https://www.golem.de/news/sicherheitsluecke-juniper-vergisst-signaturpruefung-1607-122169.html.

[CPRO17] CPROVER: CPROVER Eclipse Plugin (2017), online (abgerufen 01.06.2017): http://www.cprover.org/eclipse-plugin/.

[Delo17] Deloitte: Cloud computing - A collection of working papers (2017), online (abgerufen 02.04.2017): http://www.johnhagel.com/cloudperspectives.pdf.

[Drak15] J. "jduck" Drake: Stagefright: Scary Code in the Heart of Android (2015), online (abgerufen 01.04.2017): https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf.

[Duck14] P. Ducklin: Anatomy of a "goto fail" - Apple's SSL bug explained, plus an unofficial patch for OS X! (2014), online (abgerufen 01.04.2017): https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch.

[KhSK09] U. P. Khedker, A. Sanyal, B. Karkare: Data Flow Analysis - Theory and Practice. CRC Press (2009).

[Kleu09] S. Kleuker: Formale Modelle der Softwareentwicklung. Vieweg+Teubner (2009).

[KrMe08] F. Kröger, S. Merz: Temporal Logic and State Systems. Springer (2008).

[LiPa99] J. Lilius, I. Paltor: Formalising UML State Machines for Model Checking. In: *Electronic Notes in Theoretical Computer Science 254* (1999).

[LTMD+09] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang: Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. In: *International Conference on the Unified Modeling Language* (2009).

[Lüne14] Lünendonk: Software–Modernisierung – Im Spannungsfeld zwischen Zwangsläufigkeit und Aufwand (2014), online (abgerufen 02.04.2017): http://luenendonk-shop.de/out/pictures/0/lue_wp_softwaremodernisierung_-f211114_fl.pdf.

[Mann03] Z. Manna: Mathematical Theory of Computation. Dover (2003).

[Matt00] F. Mattern: State of the Art and Future Trends in Distributed Systems and Ubiquitous Computing. Tech. Rep., ETH Zürich (2000).

[Micr17] Microsoft: Internet der Dinge (IoT) (2017), online (abgerufen 31.05.2017): https://www.microsoft.com/de-de/internet-of-things/.

[NiNH05] F. Nielson, H. R. Nielson, C. Hankin: Principles of Program Analysis. Springer (2005).

[NiNi07] H. R. Nielson, F. Nielson: Semantics with Applications. Springer (2007).

[OWAS17] OWASP: OWASP Top 10 - 2017 rc1 (2017), online (abgerufen 31.05.2017): https://github.com/OWASP/Top10/raw/master/2017/OWASP

[Reiß14] O. Reißmann: Apples furchtbarer Fehler (2014), online (abgerufen 01.04.2017): http://www.spiegel.de/netzwelt/web/goto-fail-apples-furchtbarer-fehler-a-955154.html.

[Schn04] K. Schneider: Verification of Reactive Systems. Springer (2004).

[Schw15] P. X. Schwemmer: Static analysis for Temporally Enhanced Security Logic Assertions (TESLA). In: *Bachelor's thesis for the B.Eng. in Defence Engineering* (2015).

[Somm12] I. Sommerville: Software Engineering (9., aktualisierte Auflage). Pearson (2012).

[Weis99] M. Weiser: The Computer for the 21st Century. In: *SIGMOBILE Mob. Comput. Commun. Rev. Volume 3 Issue 3, July 1999* (1999).