Sicher, performant oder schnell entwickelt: Was darf's sein?
Wie Sie modellbasiert fundierte Designentscheidungen treffen

Stefan David, MathWorks

Abstract

Im Zeitalter der Vernetzung und der Autonomie von Maschinen (Cyber-Physical Systems) sind einige Anstrengungen nötig um sicherzustellen, dass das Risiko von Cyber-Security Attacken nicht zu gefährlichen Situationen führt, weil Hacker sich Zugriff auch sicherheitsrelevante Funktionen verschaffen können. Bei System- und Komponenten Design und der Implementierung müssen dabei oft Kompromisse eingegangen und Entscheidungen getroffen werden, da die Anforderungen in Bezug auf Funktionalität, Performanz sowie Safety und Security teilweise kontradiktorisch zueinander sind, speziell wenn es um die Erfüllung von Standards geht. Wir stellen Beispiele und Methoden vor, wie mit Hilfe von Model-Based Design, -Verifikation und statischer Code Analyse, Sicherheitslücken identifiziert, Applikationen abgesichert, standard-konform entwickelt und trotzdem performant und schnell implementiert werden können.

1 Einführung

Das Thema Security und Cybersecurity gerät zunehmend in den Vordergrund der Software-Entwicklungsprozesse. Jüngste Schätzungen gehen von Millliarden vernetzten Geräten im Jahre 2019 [1][2].

Das U.S. Department of Homeland Security (DHS) hat durch das „Industrial Control Systems (ICS) Cyber Emergency Response Team“ (ICS-CERT) ermittelt, dass der höchste Prozentsatz von bekannten Schwachstellen bzw. Sicherheitslücken in ICS-Software durch fehlende oder mangelhafte Überprüfung von Eingaben verursacht werden (Abb.1).

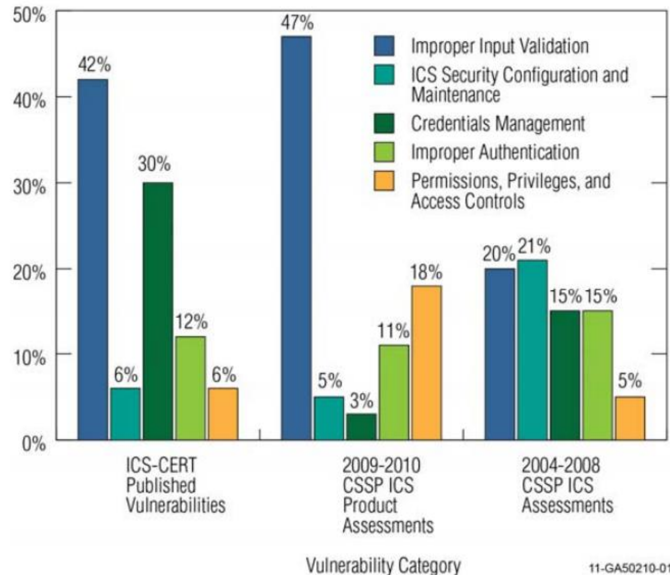

Abbildung 1: Vergleich der ICS-Software Sicherheitslücken [3]

Oftmals scheitert die Absicherung durch robuste Sicherheitsmaßnahmen an den begrenzten physikalischen Ressourcen der eingebetteten Systeme. Insbesondere bei der Verwendung von kleinen, kostengünstigen Komponenten. Ist so ein System erst einmal infiziert, ist es schwer dies zu erkennen, um die Software zu aktualisieren. Hinzu kommt, dass in der Praxis häufig erst an die Sicherheit gedacht wird, wenn die Geräte schon fertig entworfen und möglicherweise bereits im Einsatz sind.
Aufgrund der hohen Gefahr durch Angriffe entstehen zunehmend Regularien und Standards zum Thema Cybersecurity. Standards wie CERT C, ISO-TS 17961, der CWE und MISRA C:2012 Amendment 1 befassen sich mit dem Thema Security von Software. Beim System- und Komponenten Design und der Implementierung müssen dadurch oft Kompromisse werden, da die Anforderungen in Bezug auf Funktionalität, Performanz, sowie Safety und Security teilweise konkurrierend zueinander sind, speziell wenn es um die Erfüllung von Standards geht.

2 Modell-basierte Threat-/Risk Analyse

Model-based Design hat sich als eine effektive Methodik bewährt, Fehler und Schwachstellen während frühen Entwicklungsphasen kosteneffizienter zu entdecken und zu bereinigen, als in den späteren Phasen der Entwicklung [**Fehler! Verweisquelle konnte nicht gefunden werden.**]. Der Aufbau eines Modells entspricht im Allgemeinen der Applikation eines Embedded Software Systems, welches in vielen Fällen über Schnittstellen mit externen Komponenten verbunden ist (Abb. 2). Diese Schnittstellen können Unbefugten Zugriff auf sensible Bereiche Ihrer Applikation verschaffen.

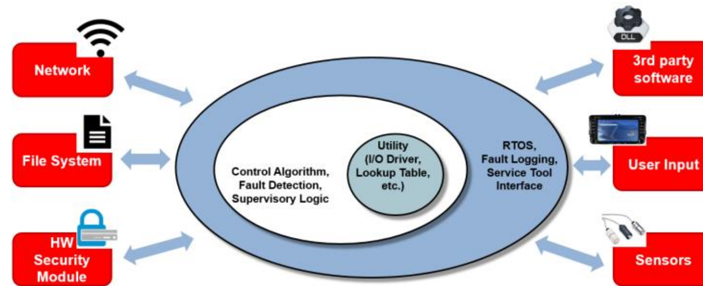

Abbildung 2: Embedded Software Applikation mit externer Interaktion

Ein strukturiertes Vorgehen, um die Ursache und Fortpflanzung von Angriffen zu erkennen, ist dabei der Schlüssel zum Erfolg. In Abb. 3 sehen wir Analyseverfahren, die über verschiedene Ebenen durchgeführt werden, z.B. **Assets and Attack Potentials** und **Threat and Risk Assessment**. Auf das Modell, bestehend aus den Blöcken Sensors, Control und Actuators, werden dabei gezielt Angriffsszenarien auf Eingänge induziert, um herauszufinden welche Kanäle anfällig sind.

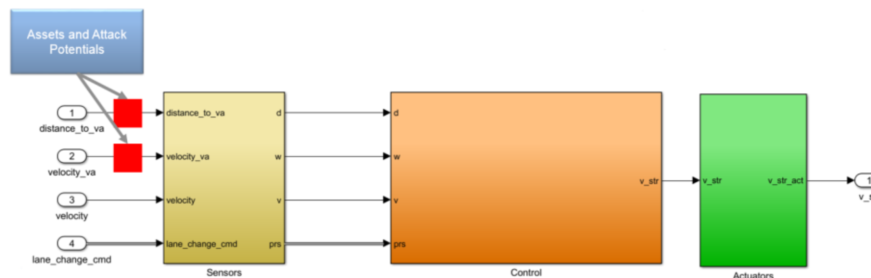

Abbildung 3: Simulink-Modell mit Angriffsszenarien in einem ICS

Im Modell angewendete Angriffs-Methoden sind:

- **Attacker centric**: Dieser Ansatz startet beim Angreifer selbst, um dessen Angriffsziele zu simulieren.
- **Design centric**: Dieser Ansatz beleuchtet das Design des Systems selbst und identifiziert mögliche Schwachstellen
- **Asset Centric**: Dieser Ansatz bezieht sich auf Daten, Informationen oder Geräte die es zu schützen gilt. Diese meist streng vertraulichen Informationen unterstehen einer höheren Priorität als die des Gesamtsystems und müssen somit gesondert betrachtet und geschützt werden.

In Kombination mit formalen Analyse-Methoden, erlaubt es **Threat Modeling** mögliche Angriffspfade als Szenarien darzustellen und Schwachstellen durch mögliche Attacken zu identifizieren, priorisieren und entsprechend zu schließen. Zum Beispiel verwendet der Simulink Design Verifier [5] formale Methoden, um Schwachstellen in Simulink

Modellen automatisiert, ohne umfangreiche Simulationsläufe, zu identifizieren. Durch Property Proving lässt sich beweisen, ob das Design, wie in den Anforderungen beschrieben, unter Berücksichtigung des Angriffsszenarios funktioniert. Sollte dieser Beweis nicht erbracht werden können, wird ein Gegenbeispiel ermittelt, das als Testfall auf das Modell ausgeführt werden kann, um das fehlerhafte Verhalten sichtbar zu machen bzw. um die Sicherheitsalgorithmen, die die Angriffe abwehren soll, zu validieren (Abb. 4).

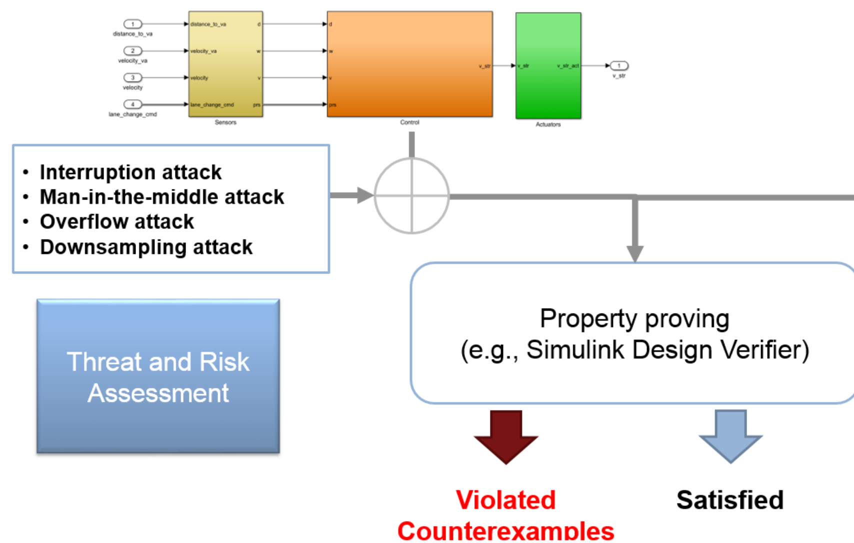

Abbildung 4: Modellierung von Angriffsszenarien mit formalen Methoden

Für das Threat Modelling können z.B. folgende Angriffs-Modelle angewendet werden

- **Interruption attack model** [6]: um den Informationsfluss zu unterbrechen
- **Overflow attack model** [7]: provozieren von Überläufen von Datentypen über Eingangs-Kanäle
- **Man-in-the-middle attack** [7] : ist ein Ansatz die Kommunikation zwischen zwei Systemen abzufangen.

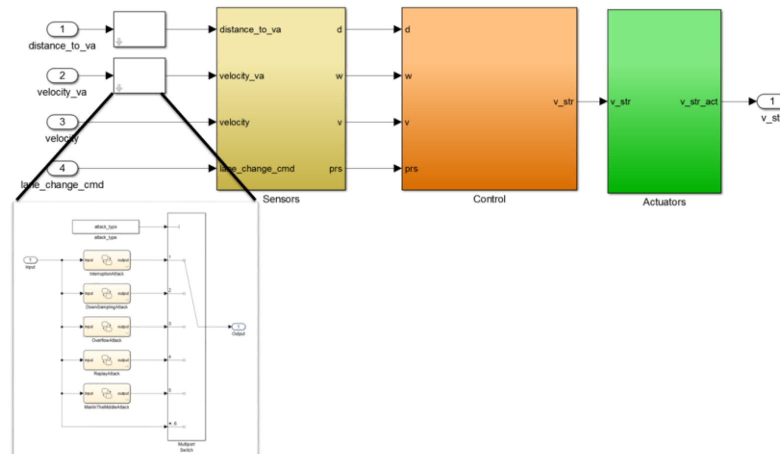

Abbildung 5: Fuzzing mit getriggerter Angriffs-Modellierung

Fuzzing (Abb.5) ist eine Test-Methodik mit der ein Applikations-Modell mit gültigen und ungültigen Eingaben bzw. „Fault Injections“ bedient wird. Diese sollen bestimmte Attacken an den Schnittstellen simulieren. Untersucht wird das System nun auf bestimmte Verletzungen, z.B. gegen funktionale oder Performance-Anforderungen.

3 Verifikation auf Code-Ebene

Durch die Integration einzelner Software-Komponenten zu einem Gesamtsystem auf Code-Ebene, das z.B. Multitasking-fähig und durch Interrupts unterbrechbar ist, können zusätzliche Schwachstellen entstehen, die Attacken zulassen und eine Analyse auf Code-Ebene erfordern.
Ein Ansatz, um den Stand der Technik einzuhalten, ist die Anwendung von Security Guidelines, um Schwachstellen zu erkennen und zu vermeiden. (Abb.7) zeigt einen Überblick verbreiteter Coding-Standards mit deren Klassifizierung ob der Standard Security oder Safety adressiert, basierend auf „The CERT C Coding Standard“ [8], wobei MISRA C:2012 durch das Amendement 1 mittlerweile ebenso Security addressiert.

Coding Standard	C Standard	Security Standard	Safety Standard	International Standard	Whole Language
CWE	None/all	Yes	No	No	N/A
MISRA C:2004	C89	No	Yes	No	No
MISRA C:2012	C99	No	Yes	No	No
CERT C99	C99	Yes	No	No	Yes
CERT C11	C11	Yes	Yes	No	Yes
ISO/IEC TS 17961	C11	Yes	No	Yes	Yes

Abbildung 7: Coding Standards for Safety and Security

Ein effektiver und kostengünster Ansatz zur Prüfung ist statische Code-Analyse. Diese hilft:

- Manuelle Code Reviews und Tests zu automatisieren

- Software auf Code-Richtlinien zu überprüfen und Verletzungen zu dokumentieren oder zu kommentieren

- Schwachstellen und Defekte automatisiert zu finden

Ein Beispiel für mögliche Schwachstellen sind Daten, die in einer Funktion verwendet, jedoch von einer äußeren Quelle an diese Funktion übergeben werden (**Tainted Data**), z.B. die Größe eines übergebenen Arrays. Durch gezielte Manipulation dieses Wertes kann damit ein Zugriff außerhalb der gültigen Arraygrenzen und somit auf beliebigen Speicherbereich stattfinden. Tainted Data sind ein beliebtes Ziel von Angriffen. Ein solcher Array-Zugriff kann sowohl ein Safety-, als auch ein Security Problem darstellen. Ein Beispiel für die Identifikation von Tainted Data Schwachstellen mittels statischer Code-Analyse, zeigt Abb. 8:

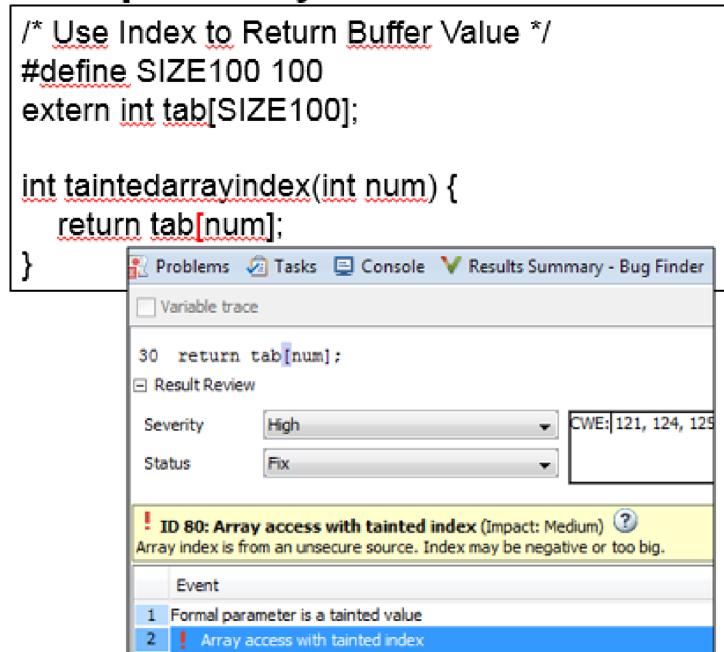

Abbildung 8: Arrayzugriff mit tainted Index identifiziert mit dem statischen Analyse-Werkzeug Polyspace Bug Finder [9]

- CWE-121: Stack-based Buffer Overflow
- CWE-124: Buffer Underwrite
- CWE-125: Out-of-bounds Read
- CWE-129: Improper Validation of Array Index
- CERT C Coding Standard — API02-C: Functions that read or write to or from an array should take an argument to specify the source or target size
- CERT C Coding Standard — ARR30-C: Do not form or use out-of-bounds pointers or array subscripts

Abbildung 9: Array Zugriff mit tainted Index und Relevanz zu Security-Standards mit Polyspace Bug-Finder [9]

Ist die Schwachstelle identifiziert, kann diese im Design oder im Code, z.B. durch eine gezielte Überprüfung der Schnittstelle auf Gültigkeit der Übergabeparameter zur Laufzeit, behoben und die Robustheit der Applikation erhöht werden.
Statische Analysetools, die zusätzlich über formale Kontroll- und Datenflussanalyse-Methoden verfügen, wie z.B. Polyspace Code Prover [9], sind darüber hinaus in der Lage die Abwesenheit bestimmter Fehler bzw. Schwachstellen zu beweisen, was den Aufwand für Tests, Reviews und den Nachweis der Compliance zu Standards (Abb.9), erheblich reduziert. Des Weiteren lässt sich dadurch auch die Code-Performance erhöhen, sowie der Speicherbedarf reduzieren, da Laufzeitchecks viel zeilgerichteter eingesetzt, bzw. vermieden werden können [10].

4 Zusammenfassung

Beim Design und der Implementierung von vernetzten Software-Systemen müssen oft Kompromisse eingegangen werden, speziell wenn es um die Erfüllung von Standards und Coding-Guidelines geht. Daraus ergibt sich, dass es enorm wichtig ist, Schwachstellen und Defekte frühzeitig, und am Besten während des Designs und der Implementierungsphase, zu erkennen und zu bereinigen. Wir haben beispielhaft Modell- und Code-basierte Methoden gezeigt, mit denen Angriffe frühzeitig simuliert und die Robustheit kostengünstig erhöht werden kann. Gerade formale Methoden lassen sich mittlerweile einfach einsetzen, um Code-Performance erhöhen, sowie Speicherbedarf reduzieren. Die Modell-basierte Entwicklung ermöglicht zudem eine deutlich schnellere Reaktion auf geänderte Standards und identifizierte Schwachstellen, als traditionell entwickelte Systeme.

Literatur

[1] J. Greenough, " 'The Internet of Things' will be the world's most massive device market and save companies billions of dollars" Feb 2015, http://www.businessinsider.de/the-internet-of-things-market-growth-and-trends-2015-2?r=US&IR=T

[2] U.G.C.S: Adviser, "The Internet of Things: making the most of the Second Digital Evolution,"

https://www.gov.uk/government/uploads/attachment_data/file/409774/14-1230-internet-of-things-review.pdf.

[3]
https://ics-cert.us-cert.gov/sites/default/files/recommended_practices/DHS_Common_Cybersecurity_Vulnerabilities_ICS_2010.pdf

[4] A. Wasicek, P. Derler, and E. A. Lee. Aspect-oriented modeling of attacks in automotive cyberphysical systems. In Design Automation Conference (DAC), 2014 51st pages 1-6. IEEE, 2014.

[5] http://de.mathworks.com/products/sldesignverifier/

[6] G. Tassey. The economic impacts of inadequate infrastructure for software testing. RTI Project Number 7007.011, NIST, 2002.

[7] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In USENIX Security Symposium. San Francisco, 2011.

[8] Robert C. Seacord, The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems. SEI series in software engineering Addison-Wesley, 2014, ISBN 0321984048, 9780321984043

[9] http://de.mathworks.com/products/polyspace/

[10] http://www.elektronikpraxis.vogel.de/embedded
computing/articles/342436/index2.html

Autor

Stefan David erhielt sein Dipl.-Ing. (BA) von der Universität in Mannheim und seinen B.Sc. von der Open University Open University UK in Informatik. Er ist ISTQB-zertifizierter Tester und sammelte mehrere Jahre Erfahrung in Entwicklung und Test, sowie als Consultant und Trainer im Bereich statische und formale Verifikation für Embedded Software. 2007 stieg er bei MathWorks als Application Engineer ein und war dort speziell für sicherheitsrelevante Applikationen zuständig. Momentan leitet er ein Team von Applikationsingenieuren, welches für Verifikationslösungen von MathWorks zuständig ist.