자동차 사이버보안: UN-ECE WP.29 및 ISO 21434에서 정적 코드 분석의 역할

유용출
Agenda

- Cybersecurity - News, Regulations and Standards
- Automotive Cybersecurity & Static Application Security Testing
- Catching Up with Cybersecurity in Three Steps
Cybersecurity –
News, Regulations and Standards
Vehicle Connectivity

- Multimedia Distribution Rear-seat Entertainment
- Internet Access
- LTE Offroad Media Services OTA Update
- Car2Car Car2Infrastructure
- Hands Free Voice Music Streaming
- Display Sharing Apple CarPlay, Android Auto, MirrorLink
- TELEMATICS
- BODY
- WiFi
- Bluetooth
- Keyless Entry Phone as Key Automated Parking Tire Pressure Monitoring Remote Sensors Control
- EV Wireless Charging
Automotive Cybersecurity in the News

Hackers can take control of your ✅ and ✅ cars - Traction Control turned off!

April 2020

Security flaws have been uncovered in two best-selling cars that could allow computer hackers to gain access and put safety and privacy at risk.

Vehicle remote control
Privacy breach
Vehicle theft

https://www.wired.com/tag/car-hacking/

New Regulations and Guidance

UN Regulations on Cybersecurity and Software Updates to pave the way for mass roll out of connected vehicles

24 June 2020

The automotive sector is undergoing a profound transformation with the implementation of systems that are vehicle automation, traditional mobility. Today, cars, electronic control units, and software code – four times more than an aircraft, is expected to rise to 300 million lines of code by 2030.

This comes with significant cybersecurity risks, as hackers seek to access electronic system data, threatening vehicle safety and consumer privacy.

Two new UN Regulations on Cybersecurity and Software Updates will help tackle these issues, establishing clear performance and audit requirements for car manufacturers. These are the first ever internationally harmonized and binding norms in this area.

The two new UN Regulations, adopted yesterday by UNECE’s World Forum for Harmonization of Vehicle Regulations, require that measures be implemented across 4 distinct disciplines:

- Managing vehicle cyber risks;
- Securing vehicles by design to mitigate risks along the value chain;
- Detecting and responding to security incidents across vehicle fleets;
- Providing and securing software updates and ensuring vehicle safety is not compromised, including a legal basis for so-called “Over-the-Air” (O.T.A) updates to onboard vehicle software.

The regulations will apply to passenger cars, vans, trucks, and buses. They will enter into force in January 2021.

Japan has indicated that it plans to apply these regulations upon entry into force.

The Republic of Korea has adopted a stepwise approach, introducing the provisions of the regulation on Cybersecurity in a national guideline in the second half of 2020, and proceeding with the implementation of the regulation in a second step.

In the European Union, the new Cybersecurity Act will take effect on December 20th, 2022, with an implementation period until December 2024.


Cybersecurity Best Practices for Modern Vehicles

New Standards
ISO/SAE 21434 - Road vehicles — Cybersecurity engineering

- Standard for Auto industry – ISO 26262 cybersecurity counterpart
- Can be used as reference standard WP.29 and NHTSA
UN Vehicle Regulations Enter into Force

The following standards may be applicable:

(a) **ISO/SAE 21434**

can be used as the basis for evidencing and evaluating …

6. Link with ISO/SAE DIS 21434

<table>
<thead>
<tr>
<th>Paragraph</th>
<th>Clauses from ISO/SAE DIS 21434</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.2.1. The vehicle manufacturer shall demonstrate to an Approval Authority or Technical Service that their Cyber Security Management System applies to the following phases:</td>
<td></td>
</tr>
<tr>
<td>Development phase</td>
<td>Clauses 9, 10, 11, 15</td>
</tr>
<tr>
<td>Production phase</td>
<td>Clause 12</td>
</tr>
<tr>
<td>Post-production phase</td>
<td>Clauses 7, 13, 14, 15</td>
</tr>
<tr>
<td></td>
<td>.........</td>
</tr>
</tbody>
</table>
New Cybersecurity Requirements for Automotive in Korea
Secure Coding Guide for Automotive Embedded System

Polyspace provides High Coverage for C/C++
Automotive Cybersecurity
&
Static Application Security Testing

F.2.2 Analysis

Analysis is a systematic and methodical means to research one or more aspects of a work product or of an item or component. Analysis checks for inherent weaknesses, human errors, known and visible system flaws, observable artefacts under the scenario of operation, and overall consistency, correctness and completeness with respect to cybersecurity requirements specifications.

Techniques can include industry standardized or best practice leading tools for identifying known vulnerabilities and weaknesses.

EXAMPLE: Static software code analysis tools that check against MISRA-C and CERT-C.
Common Cyberattack Scenarios

Programming errors are one major source of vulnerabilities

Common Cyberattack Scenarios

Best approach?
Static Analysis Tools

Programming errors are one major source of vulnerabilities

Static Application Security “Testing” (SAST) with Polyspace

Analysis & proof instead of dynamic execution

1. Enforce Secure Coding Guidelines

2. Detect Security Flaws

3. Prove Absence of Critical Vulnerabilities
1. Enforce Secure Coding Guidelines

*CERT C(++) Secure Coding Standard in Polyspace*

- Coding standard to improve safety, reliability and security
- Cross-referenced by MISRA, CWE and others

Polyspace has 100% coverage of automatable rules

Other security-relevant coding standards in Polyspace: MISRA, ISO/IEC TS 17961
2. Detect Security Flaws

Common Weakness Enumeration (CWE) with Polyspace

- MITRE categorizes to stop/eliminate those known programming errors before production
- Polyspace provides CWE mappings & views for C and C++

CWE Output
Interactive Review
CWE Searchable & Extensive Documentation

CWE-compatible Polyspace
3. Prove Absence of Critical Vulnerabilities

Polyspace Code Prover

Considers all inputs & all program states
Static Code Analysis as Recommended Method in ISO 21434

Table E.4 - Methods for verification of integration ([RQ-10-12])

<table>
<thead>
<tr>
<th>Topic</th>
<th>CAL 1</th>
<th>CAL 2</th>
<th>CAL 3</th>
<th>CAL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement-based test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Interface test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Resource usage evaluation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Verification of the control flow and data flow</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Static code analysis</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table E.5 - Methods for deriving test cases ([RQ-10-14])

<table>
<thead>
<tr>
<th>Topic</th>
<th>CAL 1</th>
<th>CAL 2</th>
<th>CAL 3</th>
<th>CAL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of requirements</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Generation and analysis of equivalence classes</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary values analysis</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error guessing based on knowledge or experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table E.9 - Topic list ([RQ-10-20])

<table>
<thead>
<tr>
<th>Topic</th>
<th>CAL 1</th>
<th>CAL 2</th>
<th>CAL 3</th>
<th>CAL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of language subsets</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Enforcement of strong typing</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Use of defensive implementation techniques</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CERT
Polyspace Bug Finder
Polyspace Code Prover
Catching Up with Cybersecurity in Three Steps

Catching up with Cybersecurity in three steps:

1. Train developers…
   – Best practices & coding guidelines to avoid common errors
   – Distribute workload on the many, “shift left”

2. Miss “no” defects with static analysis…
   – *Sound* analysis is superior to Fuzz Testing
   – Considers all corner cases, guaranteed robustness

3. Automate, Collaborate & Monitor…
   – Rigorous “nightly security reviews” without experts
   – Supporting security code reviews
   – Quality gates to keep your software robust & clean
Catching up with Cybersecurity in three steps:

1. Train developers…
   - Best practices & coding guidelines to avoid common errors
   - Distribute workload on the many, “shift left”

2. Miss “no” defects with static analysis…
   - *Sound* analysis is superior to Fuzz Testing
   - Considers all corner cases, guaranteed robustness

3. Automate, Collaborate & Monitor…
   - Rigorous “nightly security reviews” without experts
   - Central result storage & review
   - Quality gates to keep your software robust & clean
Follow Secure Coding Guidelines and Practices As You Code

Polyspace has 99.4% coverage of secure coding guideline CERT-C(++) , identifies common programming errors (CWE) and computes complexity metrics.
Fixing Flaws Requires Understanding
Root cause analysis & attack path analysis made easy

- I don’t understand the tool warning…
- …suppress/ignore ➔ missed vulnerabilities

Event traces:
1. Ease comprehension
   - Control decisions to reach vulnerability
2. Support root cause & attack path analysis
   - Partial attack path for free
3. Shorten debugging time
   - No reconstruction in debugger needed

Interactive review interfaces reduce oversight
Beyond Guidelines: Dedicated Security Checkers

Examples: OpenSSL Heartbleed (lacking data dependency), Jeep Hack (weak RNG)

Wrong variable for length
Beyond Guidelines: Automated Taint Analysis

Defects related to data from an unsecure source

```c
#define SIZE 100
extern int tab[SIZE];

int taintedarrayindex(int num) {
    return tab[num];
}
```
Beyond Guidelines: Automated Taint Analysis

Defects related to data from an unsecure source

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100
extern int tab[SIZE];

int taintedarrayindex(int num) {
    return tab[num];
}
```

Array access with tainted index

Correction — Check Range Before Use
One possible correction is to check that num is in range before using it.

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100
extern int tab[SIZE];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"), NULL, 10);
}

int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE) {
        return tab[num];
    } else {
        return -9999;
    }
}
```
Catching up with Cybersecurity in three steps:

1. Train developers…
   - Best practices & coding guidelines to avoid common errors
   - Distribute workload on the many, “shift left”

2. Miss “no” defects with static analysis…
   - *Sound* analysis is superior to Fuzz Testing
   - Considers all corner cases, guaranteed robustness

3. Automate, Collaborate & Monitor…
   - Rigorous “nightly security reviews” without experts
   - Central result storage & review
   - Quality gates to keep your software robust & clean
Why coding guidelines are good, but not enough

Many SAST tools only check “patterns”

**Guideline passed != no vulnerabilities:**

- Invalid use of standard library routine
  - Warning: function ‘memmove’ is called with possibly invalid argument(s)
    - Checks on first argument (destination):
      - Not null.
    - May not be a memory area that is accessible within the boundary given by the third argument.
      - Actual value of first argument (pointer to void): points at offset 20 in buffer of [1..20] bytes.
      - Actual value of third argument (unsigned int 32): full-range [0..2^{32}-1]
    - Checks on second argument (source):
      - Not null.
      - Is a memory area that is accessible within the boundary given by the third argument.
      - Actual value of second argument (pointer to const void): points at offset multiple of 4 in [24..60]
      - Actual value of third argument (unsigned int 32): full-range [0..2^{32}-1]

- Source Code
  - Inconsistent arguments to `memmove` → DoS!
  - Not checked by CERT/MISRA/…

**Guideline violation != vulnerability:**

- Conversion from unsigned to signed
  - No loss of sign or value
  - Valid mixing of different data types → No harm done!
  - Safe to ignore/justify MISRA violation.
Robustness “Testing” with Guarantees

F.2.7 Fuzz Testing

Fuzz testing is a type of testing where large amounts of random data are provided (usually in an automated or semi-automated fashion) as the input to a system to look for weaknesses and vulnerabilities (e.g., failures and coding errors). If the system crashes or departs from the normal defined behavior, the output is reported as an error. Fuzz testing can be done at the system or interface level, or more exhaustively by listing every variable in the software under test and fuzzing random values for each software variable in the code. In the latter approach, the testing is typically highly automated. Fuzz testing can be used to discover, for example, overflows, segmentation and heap errors that have cybersecurity implications. Fuzz testing can be applied to hardware inputs. Fuzz testing can be used as a technique for penetration testing.

- Through Fuzz testing
  - Requires execution on target ⇒ slow
  - Requires test harness ⇒ effort
  - E.g., (anti-)random testing, coverage testing, genetic algorithms

Not exhaustive ⇒ may miss vulnerabilities

From: ISO/SAE DIS 21434
Robustness “Testing” with Guarantees

- Through Fuzz testing
  - Requires execution on target → slow
  - Requires test harness → effort
  - E.g., (anti-)random testing, coverage testing, genetic algorithms

- **Sound** static analysis with proof
  - Based on analysis, not execution
  - Requires no test harness
  - Considers all inputs & states
    - Boundary values, race conditions, sufficient checking of user inputs…?

Not exhaustive → may miss vulnerabilities

Miss no (checked) bugs → less vulnerabilities

Polyspace Code Prover
Sound Static Application Security Testing (SAST) with Polyspace

Proof of robustness by analysis instead of evidence from dynamic execution

Considers all inputs & all program states, reduces need for Fuzz Testing
Catching up with Cybersecurity in three steps:

1. Train developers…
   - Best practices & coding guidelines to avoid common errors
   - Distribute workload on the many, “shift left”

2. Miss “no” defects with static analysis…
   - Sound analysis is superior to Fuzz Testing
   - Considers all corner cases, guaranteed robustness

   Automate, Collaborate & Monitor…
   - Rigorous “nightly security reviews” without experts
   - Central result storage & review
   - Quality gates to keep your software robust & clean
Continuous Vulnerability Verification

Direct developer feedback in IDE to fix security coding standards (CERT)

Supporting unopinionated code reviews focusing on vulnerabilities

Automating quality gate into CI pipelines

Reporting and Certification artifacts
Cybersecurity Is Everyone Concern

Polyspace Access

Developer Branch

Focus on newly introduced vulnerabilities

Final VnV

Define quality threshold
Populate reports with justification

Collaborative vulnerabilities review
Integrate with Issue tracking tool (Jira, Redmine)
MathWorks Capabilities for Cybersecurity

- System Requirements
- System Design
- SW Design
- SW Test
- SW Implementation
- System Release
- System Integration and Test
- Continuous System Care

Code level security & robustness analysis
MathWorks Capabilities for Cybersecurity

System Requirements
- Allocate sec reqs, threat/risk analysis
- Secure design & modeling

System Design
- Secure code generation & deployment
  - MISRA C/C++
  - CERT C/C++
  - CWE
  - TS 17961

System Integration and Test
- System Testing
- Intrusion detection
- Attack Sim.

System Release
- Int. Tests & intrusion detection
- Req. testing, Fuzz testing, Attack Sim.

Continuous System Care
- SOC Data Analytics, Design Updates
- Intrusion
- Detection
- Reaction

SW Design
- Secure design & modeling

SW Test
- Secure code generation & deployment
- Code level security & robustness analysis

SW Implementation
- Allocate sec reqs, threat/risk analysis
- Intrusion detection
- Attack Sim.
Key Takeaways

▪ Achieve Higher Security Level with Polyspace Products

▪ Prove Absence of Critical Vulnerabilities to Reduce Testing Effort

▪ Raise Team Skills to Tackle Vulnerabilities
MATLAB EXPO
2021
Thank you