Addressing Future Challenges in the Development of Safe and Secure Software Components
Cybersecurity – Emerging Topic in the Auto Industry

Vehicle-to-Infrastructure

Wifi Hotspot

Remote Start

Internet Connection

Wireless Key

Tire Pressure Monitor

Vehicle-to-Vehicle

eCall

Bluetooth Connection
Growing security concerns – rise in number and complexity

- “In 2013, 2164 separate incidents, over 822 million records were exposed, nearly doubling the previous highest year on record (2011).”

- “Hacking accounted for almost 60% of incidents .. US has by far the biggest share of pain with 48.7% of the total ..”

- Continued to increase in 2014 and 2015

- Researchers demonstrated a proof-of-concept attack where they managed to take control of a vehicle remotely

Sources:
https://nakedsecurity.sophos.com/2014/02/19/2013-an-epic-year-for-data-breaches-with-over-800-million-records-lost/
http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
Conflicting Safety & Security Goals

- "When the full locking system is employed, the interior operations for unlocking the door locking mechanism are disengaged, which makes the locking system an effective anti-theft measure. However, if the doors cannot be opened by an occupant from inside the vehicle, they may be trapped inside."

- Resulted in recall of more than 6000 cars

Communication between Cybersecurity and Safety Process

Source: SAE 3061 Cyber Security Framework
Cybersecurity – Industry Activities & Standards

SAE – Vehicle Cybersecurity Systems Engineering Committee
- SAE J3061 - Cybersecurity Guidebook for Cyber-Physical Vehicle Systems
- SAE J3101 - Requirements for Hardware-Protected Security for Ground Vehicle Applications (WIP)
- SAE “Cybersecurity Assurance Testing Task Force” (TEVEES18A1)

Coding standards & practices that we observe at automotive customers
- CERT C
- ISO/IEC TS 17961 – C Secure Coding Rules
- CWE – Common Weakness Enumeration
- MISRA-C:2012 Amendment 1
Example Cybersecurity Lifecycle (SAE 3061)

Source: SAE 3061 Cyber Security Framework
Identifying flaws as early as possible in the CPS design is $5\times$ to $10\times$ less expensive than finding them during the detailed design stages.

Model-Based Safety/Security Analysis

Simulink Model of a function

Property proving (e.g., Simulink Design Verifier)

„Maintain safe distance by braking“

Satisfied

Violated

Counterexamples
Model-Based Safety/Security Analysis

Assets and Attack Potentials

1. distance_to_va
2. velocity_va
3. velocity
4. lane_change_cmd

Sensors

Control

d
w
v
prs
Model-Based Safety/Security Analysis

Simulink Model of a function
Property proving (e.g., Simulink Design Verifier)

"Maintain safe distance by braking"

Threat models
Threat and Risk Assessment

Property proving (e.g., Simulink Design Verifier)
Violated Counterexamples
Satisfied
Model-Based Safety/Security Analysis

- Interruption attack
- Man-in-the-middle attack
- Overflow attack
- Downsampling attack

Property proving (e.g., Simulink Design Verifier)

„Maintain safe distance by braking“

Violated Counterexamples

Satisfied
Model-Based Safety/Security Analysis

Simulink Design Verifier
(Property proving)

Violated Counterexamples

Satisfied
Fuzzing is a technique whereby an application is fed malformed input data with the aim to detect anomalies in the processing of unexpected data entry.

- Leverage **smart** fuzz testing to identify vulnerabilities
- Define intended cybersecurity violation
- Automated test/attack case generation
Automation to handle design Trade-offs

Check Modified Design

Code generation

C Code

Improvement Suggestions
# Coding Standards Overview

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE</td>
<td>None/all</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>MISRA C:2004</td>
<td>C89</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MISRA C:2012*</td>
<td>C99</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CERT C99</td>
<td>C99</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CERT C11</td>
<td>C11</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ISO/IEC TS 17961</td>
<td>C11</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*) 04/2016: Additional security guidelines for MISRA-C:2012 Amendment 1

Source: Table is based on the book:
**CWE** : Common Weakness Enumeration

- Not limited to the C programming language

- Not limited to potential issues caused by source code
  - E.g.: CWE-5: J2EE Misconfiguration: Data Transmission Without Encryption

- Includes CERT C
This coding standard consists of **rules** and **recommendations**, collectively referred to as **guidelines**.

**Rules** are meant to provide normative requirements for code, whereas
- Actual: 98 Rules (depending on applicable categories)

**Recommendations** are meant to provide guidance that, when followed, should improve the safety, reliability, and security of software systems.
- Actual: 178 Recommendations (depending on applicable categories)
L2 Example: Rule 05. Floating Point (FLP)

FLP30-C. Do not use floating-point variables as loop counters

Because floating-point numbers represent real numbers, it is often mistakenly assumed that they can represent any simple fraction exactly. Floating-point numbers are subject to representational limitations just as integers are, and binary floating-point numbers cannot represent all real numbers exactly, even if they can be represented in a small number of decimal digits.

In addition, because floating-point numbers can represent large values, it is often mistakenly assumed that they can represent all significant digits of those values. To gain a large dynamic range, floating-point numbers maintain a fixed number of precision bits (also called the significand) and an exponent, which limit the number of significant digits they can represent.

Different implementations have different precision limitations, and to keep code portable, floating-point variables should not be used as loop counters.

Noncompliant Code Example

In this noncompliant code example, a floating-point variable is used as a loop counter. The decimal number 0.1 is a repeating fraction in binary and cannot be exactly represented as a binary floating-point number. Depending on the implementation, the loop may iterate 9 or 10 times.

```c
void func(void)
    {for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
        /* loop may iterate 9 or 10 times */
    }
}
```

Compliant Solution

For example, when compiled with GCC or Microsoft Visual Studio.

Risk Assessment

The use of floating-point variables as loop counters can result in unexpected behavior.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLP30-C</td>
<td>Low</td>
<td>Probable</td>
<td>Low</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

Related Guidelines

- SEI CERT C++ Coding Standard
- CERT Oracle Secure Coding Standard for Java
- NIST 800-127-J. Do not use floating-point variables as loop counters
- ISO/IEC TR 24772:2013 Floating-Point Arithmetic (FLP)
- MISRA C:2012 Directive 1.1 (required)
- Rule 14.1 (required)
Mapping to MISRA C:2012

Rule 14.1 A loop counter shall not have essentially floating type

Category Required
Analysis Undecidable, System
Applies to C90, C99

Rationale
When using a floating-point loop counter, accumulation of rounding errors may result in a mismatch between the expected and actual number of iterations. This can happen when a loop step that is not a power of the floating-point radix is rounded to a value that can be represented.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it may give a different number of iterations on another implementation.

Example
In the following non-compliant example, the value of counter is unlikely to be 1000 at the end of the loop.

```c
uint32_t counter = 0u;
for ( float32_t f = 0.0f; f < 1.0f; f += 0.001f )
{
    ++counter;
}
```
Potential Cert C issues *

- FLP30-C 2
  - Do not use floating-point variables as loop counters
- FLP34-C 41
  - Ensure that floating-point conversions are within range of the new type
- INT30-C 72
  - Ensure that unsigned integer operations do not wrap
- INT31-C 343
  - Ensure that integer conversions do not result in lost or misinterpreted data

*) all user made; found in C code generated from 50 industry models
Potential Cert C issues *
Reason: models have not been designed to comply with Cert C (issues are specifically relevant if taint data is involved)

- **FLP30-C**
  - Do not use floating-point variables as loop counters
- **FLP34-C**
  - Ensure that floating-point conversions are within range of the new type
- **INT30-C**
  - Ensure that unsigned integer operations do not wrap
- **INT31-C**
  - Ensure that integer conversions do not result in lost or misinterpreted data

*) all user made; found in C code generated from 50 industry models
Automated analysis
Defects related to Code elements from an unsecure source (e.g. parameters, globals, etc.)

- Example: Array access with tainted index
/* Use Index to Return Buffer Value */
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
Automated analysis: e.g. Static Taint Analysis detects issues
Defects related to Code elements from an unsecure source

Example: Array access with tainted index

```c
/* Use Index to Return Buffer Value */
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```
Automated analysis: e.g. Static Taint Analysis satisfies standards

Defects related to Code elements from an unsecure source

- Example: Array access with tainted index

```c
/* Use Index to Return Buffer Value */
#define SIZE100 100
extern int tab[SIZE100];

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

- 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
Automated analysis: e.g. Static Taint Analysis on the fix...

Defects related to Code elements from an unsecure source

- **Example: Array access with tainted index**

```c
/* Use Index to Return Buffer Value */
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -9999;
    }
}
```
Automated analysis: e.g. Analysis with Formal Proof
Defects related to Code elements from an unsecure source

- Example: Array access with tainted index

```c
/* Use Index to Return Buffer Value */
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -9999;
    }
}
```

- ID 15: Out of bounds array index
  Array index is within bounds: [0..99]
  array size: 100
  array index value: [0 .. 99]

- Array access with tainted index
- Command executed from externally controlled path
- Execution of externally controlled command
- Host change using externally controlled elements
- Library loaded from externally controlled path
- Loop bounded with tainted value
- Memory allocation with tainted size
- Pointer dereference with tainted offset
- Tainted division operand
- Tainted modulo operand
- Tainted NULL or non-null-terminated string
- Tainted sign change conversion
- Tainted size of variable length array
- Tainted string format
- Use of externally controlled environment variable
- Use of tainted pointer
A static and semantic code analysis tool helps you to...

- Identify defects
- Enforce coding rules
- Produce and monitor quality metrics

Enforce coding rules:
- 1 A standard C environment
- 2 Compilation and build
- 3 Unused code
- 4 Code design
- 5 Identifiers
- 7 Literals and constants
- 8 Variables and definitions

Produce and monitor quality metrics:
- Event tracking
- Code coverage analysis
- Fault distribution by severity
- Memory allocation analysis
- Function call analysis
- Resource usage analysis
Key Takeaways and call to action

- Adressing cyber-security uses processes, methods and tools known from functional safety domain, with different scope

- Static and semantic code analysis provides a quick start to identify security vulnerabilities and assess coding standard compliance

- Model-based design streamlines efforts for meeting existing and emerging safety & security standard compliance
Questions!?! 

Thank You!