MATLAB EXPO 2019
Making Software Safe and Secure with Team Collaboration

Vaishnavi H R
Application Engineer, Mathworks
Agenda

1. Making Software Safe and Secure
2. Polyspace Static Analysis
3. Team Collaboration with Polyspace
1. Making Software Safe and Secure
“Program testing can be used to show the presence of bugs, but never to show their absence”

Edsger Dijkstra, Computer Science Pioneer

“Given that we cannot really show there are no more errors in the program, when do we stop testing?”

Brent Hailpern, Head of Computer Science

Dijkstra, “Notes on Structured Programming” (1972)
Using Static Analysis to Make Software Safe and Secure

- Find bugs without code execution
  - Code analyzed without running tests
  - Identify bugs and coding rule violations for MISRA, AUTOSAR, CERT

- Prove absence of critical run-time errors
  - Identify code that will never experience errors regardless of run-time conditions

- Complements dynamic testing
  - Used together, you can find more bugs for higher quality code
When Software Safety and Security Matter

- Industries where safety and security matter
  - Automotive, Aerospace, Medical Device, Industrial Machinery

- Governed by functional safety and other standards
  - ISO 26262, DO-178, IEC 62304, IEC 61508
  - MISRA, CERT, AUTOSAR

- Static analysis provides certification credits
  - For standards such as ISO 26262 and DO-178
2. Polyspace Static Analysis

_For software written in C, C++, and Ada_
Proving Absence of Critical Run-Time Errors

- How many run-time errors are possible?
  1. Divide by zero
  2. Overflow
  3. Uninitialized variables

- How to test all floating point variable combinations?

- How do you prove that this code will not fail?

```c
float x, y;
...
x = x / (x - y);
```
Proving Absence of Critical Run-Time Errors

Proven by Polyspace that run-time error will _not_ occur

```c
float where_are_errors_float(float input)
{
    float x, y, k, l, limit = 1000.0f;
    if (input <= -limit || input > limit) return (-9999.0f);
    k = input / 100.0f;
    x = 2.0f;
    y = k + 5.0f;
    while (x < 10.0f)
    {
        x++;
        y = y + 3.141592f;
    }
    if ((3.0f * k + 100.0f) > 71.0f)
    {
        y++;
        x = x / (x - y);
    }
    return x;
}
```
Proving Absence of Critical Run-Time Errors with Polyspace
**Polyspace Tools**

**Bug Finder**
- Produce code metrics
- Check coding standards
- Find defects and vulnerabilities

**Code Prover**
- Proves code Safe and Secure
- 33 most critical run-time checks
- Supports DO-178 and ISO 26262
Polyspace Customer References

KOSTAL Asia R&D Center Receives ISO 26262 ASIL D Certification for Automotive Software

Alenia Aermacchi Develops Autopilot Software for DO-178B Level A Certification

Miracor Eliminates Run-Time Errors and Reduces Testing Time for Class III Medical Device Software
3. Team Collaboration with Polyspace
Workflow with New Polyspace Products in R2019a

1. Developers check-in code into repository, Build Engineer has configured Jenkins to run Polyspace analysis
2. Jenkins initiates Polyspace analysis run on the server (periodically or at program milestones)
3. Once Polyspace analysis run concludes, results are uploaded to Polyspace Access
4. Team Lead/Manager, QA, Developers use web browser to review results, open Jira defects, monitor quality metrics
Team Collaboration Story

Bob is the Build Engineer
He has configured Polyspace in a Jenkins CI workflow

Quinn is a Quality Engineer
She is responsible for triaging software defects

Dara is a software developer
She is responsible for writing code and fixing defects

Eric is a Simulink and Embedded Coder user
He is responsible for generating code from models

Martin is a project manager
He is responsible for software quality of the project
Bob is the Build Engineer
He has configured Polyspace in a Jenkins CI workflow
Quinn is a Quality Engineer
She is responsible for triaging software defects

- She received an email notification from last night’s Jenkins initiated Polyspace analysis
- The email indicates several findings were found in her project
- She click on the link in the email to view the findings in Polyspace Access
Quinn is a Quality Engineer
She is responsible for triaging software defects
Dara is a software developer
She is responsible for writing code and fixing defects

- Dara has been assigned 2 defect tickets in Jira
- She opens the first JIRA ticket and clicks the Polyspace Access link
Dara is a software developer. She is responsible for writing code and fixing defects.
Eric is a Simulink and Embedded Coder user. He is responsible for generating code from models.
Eric is a Simulink and Embedded Coder user. He is responsible for generating code from models.
Martin is a project manager
He is responsible for software quality of the project
Summary

▪ Use Polyspace to achieve high quality software with reduced testing effort
  – Prove that your code will not cause safety hazards or security issues

▪ Polyspace fits software development workflows
  – Jenkins for build automation and Jira for bug tracking

▪ Supports team based collaboration
  – Results published for web-browser based review by developers and quality engineers
  – Dashboards to show quality metrics for project and safety managers.
Polyspace Helps Makes C, C++, and Ada Safe and Secure

<table>
<thead>
<tr>
<th>Safety Standards:</th>
<th>Security Standards:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• DO-178 (aero)</td>
<td>• CERT-C</td>
</tr>
<tr>
<td>• ISO 26262 (auto)</td>
<td>• CWE</td>
</tr>
<tr>
<td>• IEC 61508 (industrial)</td>
<td>• ISO 17961</td>
</tr>
<tr>
<td>• IEC 62304 (medical)</td>
<td>• MISRA-C:2012 Appendix 1</td>
</tr>
<tr>
<td>• EN 50128 (rail)</td>
<td>• Tainted data tracking</td>
</tr>
</tbody>
</table>

**Reliability and Robustness**

**Code Proving**
- Prove absence of critical runtime errors (or find even the slightest vulnerability)
- Exhaustive: all possible inputs, control flows, data flows (no instrumentation, execution, test cases)
- Sound: no false negatives

**Quality**

- Coding Standards
- Find Probable Bugs, Defects
- Code Metrics
- Formal Method: Runtime Behavior, Debugger-like view
- Review Scopes / Software Quality Objectives
- Simulink Integration: trace issues in generated code back to model
Learn More

Visit MathWorks Code Verification Solution Page:
Polyspace for C/C++ Code Verification

This two-day course discusses the use of Polyspace Bug Finder™ and Polyspace Code Prover™ to prove code correctness, improve software quality metrics, and ensure product integrity.

**Topics include:**
- Creating a verification project
- Reviewing and understanding verification results
- Emulating target execution environments
- Handling missing functions and data
- Managing unproven code (color-coded in orange by Polyspace® products)
- Applying MISRA C® rules
- Reporting
Thank You