LDRA
Software Technology

Technical Briefing

Unveiling and preventing weaknesses: An exploration of taint analysis in security assurance

www.ldra.com

Introduction

Developers of embedded software have seen some dramatic events in recent years. The introduction of functional safety standards across the sectors has resulted in new, more structured established development practices for many. A near exponential rise in demand for embedded software content has added to the challenge, along with corresponding increases in software engineering complexity. But perhaps the advent of the Internet of Things (IoT) and its industrial counterpart (IIot) represents the biggest conceptual shift of all. Developers working on connected applications have seen a move away from static, fixed function, device specific applications to a "cloud" connected world, expected to be accessible to facilitate monitoring, upgrading, enhancement, and supplementation.

Each of these issues individually has the potential to turn established practices on their heads, and there is sufficient common ground between them for the combination to be especially telling. For example, ISO 26262 in the automotive sector is entitled "Road vehicles – functional safety" [1], and clearly software faults leading to safety issues fall within its remit. But less obviously it is also concerned with the cybersecurity issue too. After all, if a lack of security takes the control of a car away from its driver, then that security breach is also functional safety issue [2].

The good news is that despite all this connectivity being something of a Brave New World in the embedded sectors, there is no need for a completely fresh approach. The adaptation of the "designed-in" approach that has proven so successful in functional safety environments lends itself equally well to a "shift left" approach to the development of security-critical applications.

Cybersecurity is sufficiently complementary to functional safety that application developers tasked with the development of purely security-critical applications have much to learn from the world of functional safety. Similarly, developers of connected safety-critical applications will find that their existing development lifecycles can easily be adapted to serve the needs of cybersecurity. The addition of taint analysis to the verification and validation process is one such adaptation.

Defence-in-depth

Defending against cyberattacks is a daunting task. No single defence of a connected system can guarantee impenetrability. The best available approach is therefore to apply multiple levels of security to ensure that if one level fails, others stand guard. One popular analogy for this defence-in-depth approach [3] is that of a medieval castle equipped to defend its inhabitants from siege through the use of towers, curtain walls, moats, mounds, gates, drawbridges, and other defensive mechanisms [4] (Figure 1). And yet, aggressors did compromise castles despite these defences.

Figure 1: The city of Carcassonne in France is a concentric castle, featuring multiple layers of curtains walls.

Typical examples of the analogous defences available to developers of connected embedded systems include:

- **Data-at-rest protection:** For example, obfuscate the data via an encryption algorithm.
- **Secure boot:** Apply a well-designed secure boot sequence to ensure that the correct image loads.
- **Domain separation:** Prevent, as far as is reasonable, access to critical domains from more benign domains.
- **MILS design principles:** Apply a "least privilege" approach such that applications only have access to the minimum set of interfaces and services necessary to fulfil their intended function.
- **Attack surface reduction:** Minimize the attack surface by removing all code and interfaces that are not absolutely needed.
- **Secure coding best practices:** Apply a proactive "shift-left" approach to application code development.

Taint analysis has a significant contribution to make to secure coding best practices. But it will only be effective as part of a broader development policy that gives cybersecurity adequate priority across the system.

Weaknesses and vulnerabilities

In the field of cybersecurity, the terms "weakness" and "vulnerability" have specific and related meanings. Weaknesses are errors or issues that can lead to vulnerabilities. A software vulnerability, such as those enumerated on the Common Vulnerabilities and Exposures (CVE) List [5], is a mistake in software that can be directly used by a hacker to gain access to a system or network. The Common Vulnerability Scoring System (CVSS) [6] and Common Weakness Scoring System (CWSS) [7] can each help to categorize vulnerabilities and weaknesses in software whether at module, application, or source code level, and both are useful references for threat categorization.

Shifting left

Cybersecurity is primarily concerned with the protection of data. However, in developing a secure application, it is easy to forget that providing such protection involves more than just the data itself. It also demands that the system is designed to safeguard that data against aggressors, and to defend interfaces to the system.

Traditional practice for secure code verification is largely reactive, meaning that code is developed in accordance with relatively loose guidelines and then tested by means of performance, penetration, load, and functional testing to identify vulnerabilities prior to their being addressed (Figure 2)

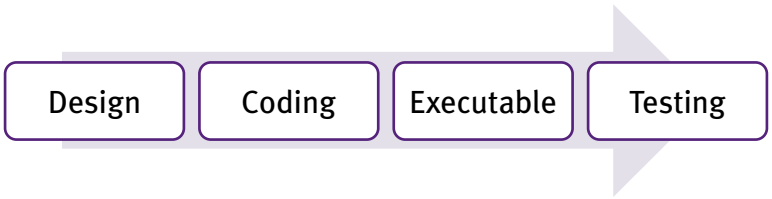


Figure 2: The traditional reactive approach to secure code development.

A better, proactive approach is to ensure that code is secure by design, as shown in Figure 3. That implies a systematic development process, where the code is written in accordance with secure coding standards, is traceable to security requirements, and is tested to demonstrate compliance with those requirements as development progresses.

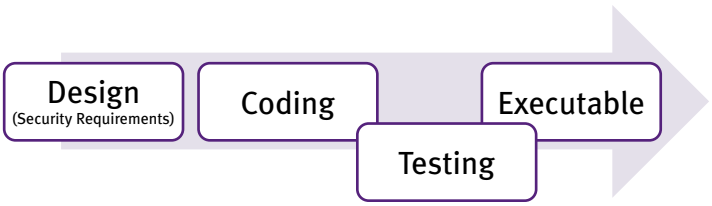


Figure 3: The proactive approach to secure code development.

This proactive approach integrates security related best practices into the traditional Software Development Life Cycle (SDLC). The resulting Secure Software Development Life Cycle (SSDLC) represents the state-of-the-art for application developers and is a practical approach to ensuring that weaknesses (and hence vulnerabilities) are designed out of the system or addressed in a timely and thorough manner.

Trust boundary identification

The term “trust boundary” carries multiple meanings. In the realm of cybersecurity, trust boundaries delineate the points at which there is a shift in the levels of “trust” concerning program execution or data protection. These boundaries can signify locations or surfaces susceptible to intervention by attackers. Therefore, employing systematic approaches that effectively aid in identifying trust boundaries is imperative.

Trust boundary identification (TBI) is a design-time technique, involving the systematic decomposition of the application, and to analyse the data and control entry and exit points.
The example shown in Figure 4 shows the trust-boundary layers of a use-case for an automotive Battery Management System (BMS), as cited by Georg Macher et al [8].

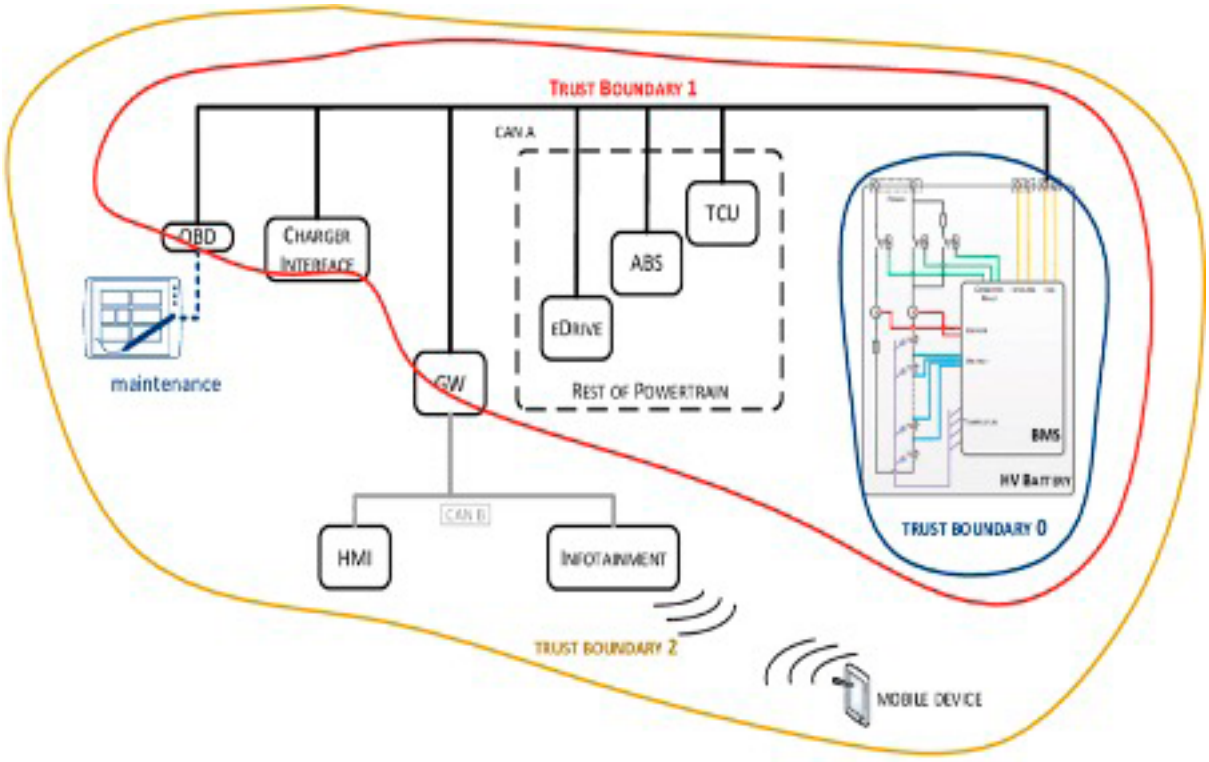


Figure 4: Trust-boundary layers of a use-case for an automotive Battery Management System (BMS).

The technique requires that appropriate controls are defined wherever data crosses the trust boundaries and documented in a "threat model" in the form of specialized data flow diagrams which show different paths through the overall system, highlighting privilege boundaries.

This analysis and its associated documentation help the design team determine where to place particular emphasis during cybersecurity validation – perhaps in the form of one the CERT top 10 secure coding practices [9], such as the sanitization of data send to peripherals. Later, during the implementation phase, the application of secure coding standards is an example of best practice is achieving that sanitization.

The second step involves using threat categorization to identify the severity of threats based on the break-down of the system. Threat categorization is the process of classifying and organizing potential threats or risks based on various attributes such as their nature, source, impact, and likelihood.

Understanding taint terminology

Taint analysis is concerned with the analysis of code that is handling data that has entered the system or subsystem by crossing a trust boundary. It is therefore related to Trust Boundary Analysis.

Taint sources are locations in the program where data is being read from a potentially risky source, and include things like environment variables, data, files, file metadata (such as a file's permissions or data stamps), the network or information bus, the system clock, communications from separated hypervisor ARINC 653 partitions, or network services (such as the results of a DNS query).

All data entering the (sub)system of interest is generally referred to as **tainted data** whether it has been compromised, or not. Given that tainted data is merely that data which has been read from a potentially risky source, then most of the time that data will be of no issue. The problem arises when the tainted data is somehow compromised. The opportunity to do so can typically be found in the application code where **weaknesses** are found that can lead to vulnerabilities.

Taint sinks are the end of data flow for tainted data. In the context of security static analysis, sinks are the operations that must have clean data. A taint sink could be a database call, a value written to a file, or a 4-20mA output - anything that could lead to complications if an attacker were to gain complete control over it. **Sanitizing** operations are used to filter tainted data. After sanitization, tainted data becomes clean and can safely enter a sink.

Taint checks are used to highlight specific security risks primarily associated with connected devices which are attacked using techniques such as SQL injection or buffer overflow attack approaches. Taint checks include techniques such as bounds limiting and value checking.

Static and dynamic attack vectors

Static attack vectors target vulnerabilities inherent in systems or software, independent of their operational state, often stemming from weaknesses associated with design or implementation flaws like buffer overflows or misconfigured permissions. Detection involves weakness scanning and code analysis – that is, static analysis techniques. Prevention involves focusing on patching, system hardening, and adherence to secure coding standards.

Dynamic attack vectors, on the other hand, exploit vulnerabilities arising during system operation. Examples include injection attacks (including SQL injection) or Denial of Service (DoS) attacks. The nature of dynamic attack vectors means that associated weaknesses are generally detected more effectively by means of dynamic analysis. For example:

- Injection attacks, such as SQL injection or command injection, involve the execution of malicious code or commands within the context of a running application.
- DoS attacks typically involve flooding a system or network with excessive traffic or resource requests to overwhelm its capacity and disrupt normal operation.

Dynamic analysis provides the means to mimic such actions, and to analyse the capacity of the code to prevent performance degradation or service interruptions during the attack.

Secure code needs to be robust in the face of both static and dynamic attack vectors – and is proven capable of it by means of a combination of static and dynamic analysis.

Taint analysis and security standards

While specific security standards may not explicitly mandate the use of taint analysis, many standards and best practices within the field of secure software development align with the use of such techniques.

For example, the ISA/IEC 62443 series of standards primarily focuses on industrial automation and control systems (IACS) security. These standards provide guidelines and best practices for securing industrial networks and systems against cyber threats. ISA/IEC 62443-4-1 [10] deals with the security of product development in the IACS environment.

The ISA/IEC 62443 standards do not prescribe specific technical methods or tools. Instead, they provide a framework and a set of security requirements that organizations can use to develop and implement their security measures. They therefore do not explicitly mention taint analysis, but they do emphasize risk assessment, security by design, and secure coding practices – and taint analysis aligns perfectly with those principles.

Other standards taking a similar position include OWASP ASVS [11] (application security), NIST SP 800-53 [12] (US federal systems), and ISO/SAE 21434 [13] (automotive).

Taint analysis involves four steps. Figure 5 shows the four taint analysis steps (left) and applicable techniques to be applied using the LDRA tool suite (right).

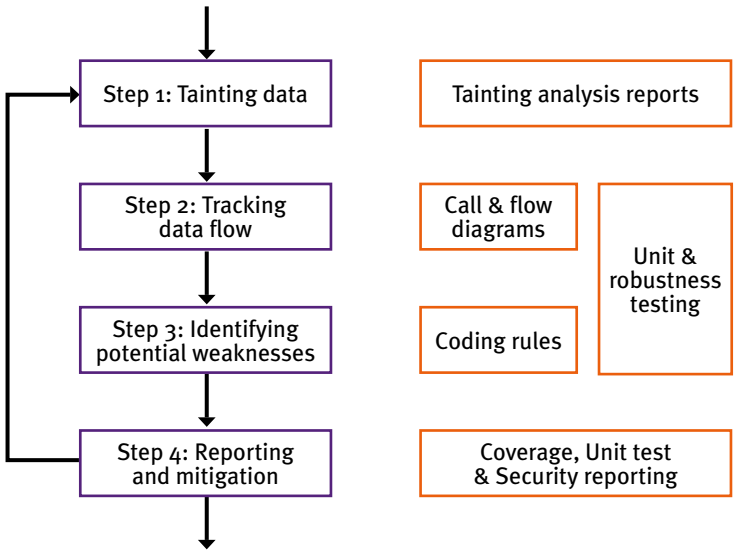


Figure 5: Diagram showing the four taint analysis steps (left). and applicable techniques (right).

Step 1: Tainting data

This step involves the marking of data as tainted when it originates from an untrusted or external source, such as user input, data received from a network, communication from another hypervisor partition – in fact, any data that could be influenced by an attacker.

The LDRA tool suite provides a facility to identify taint sinks and sources within the code base. The report shown in Figure 6 shows taint sources that are associated with stdio (standard input/output) but in many systems, there will be other keywords to consider.

For instance – these might consist of POSIX or other RTOS calls, access to shared memory between hypervisor partitions, inter-partition communication between ARINC 653 partitions, or TCP/IP communications. The LDRA tool suite is configurable to identify the keywords that are associated with such sources.

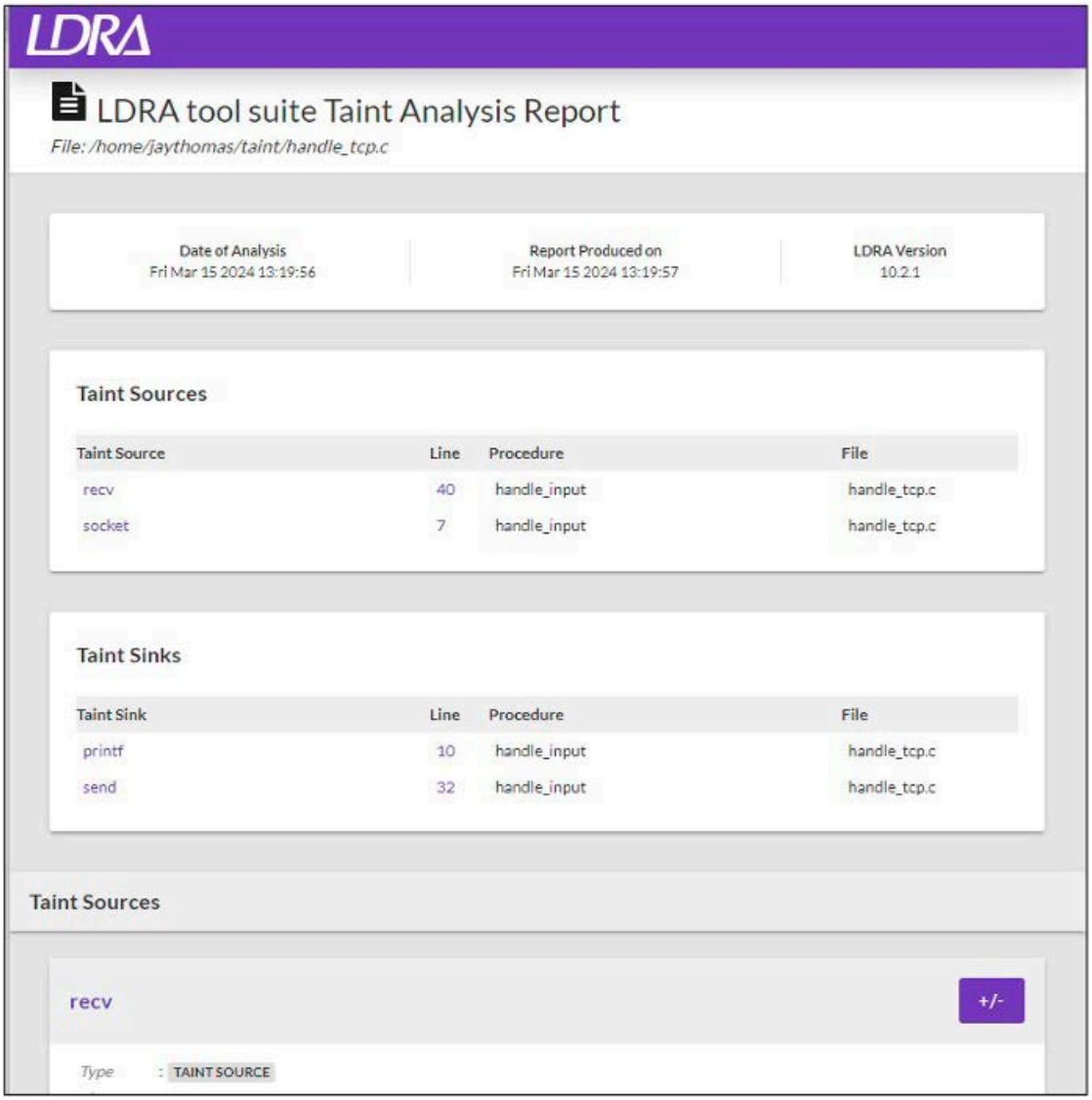


Figure 6: Taint analysis overview report as generated using the LDRA tool suite.

If trust boundary identification has been completed at design time, cross referencing the boundary crossings identified to sources and sinks identified in the report provides a means to confirm that the design has been correctly and fully implemented.

Step 2: Tracking data flow

The second step involves tracing the flow of tainted data as it moves through the (sub)system from source to the sink.

The LDRA tool suite [14] includes the capability to trace the flow statically, with reference to call and flow diagrams. However, although static analysis can provide useful information, it cannot provide the whole story.

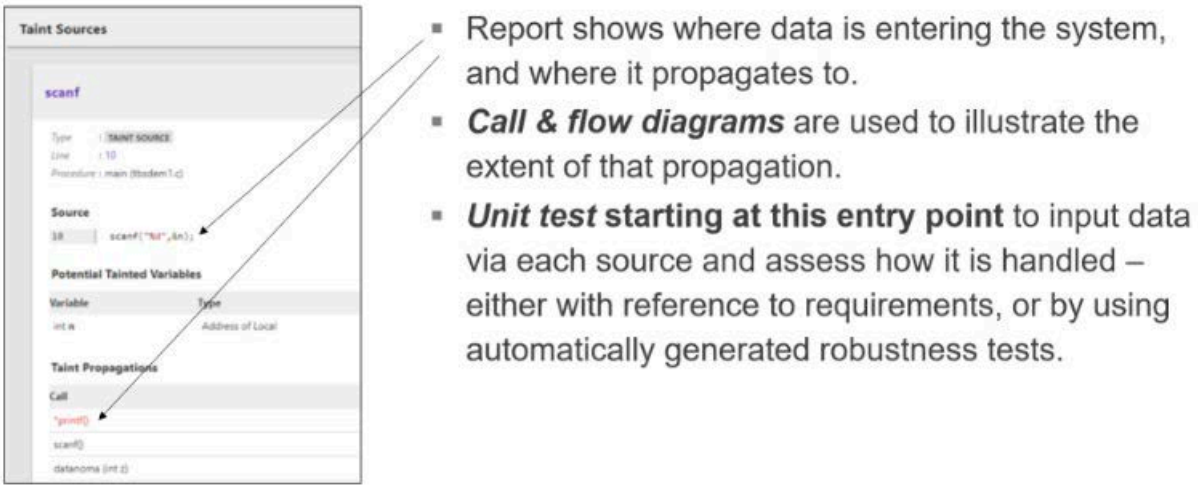


Figure 7: Tracking data flow with the LDRA tool suite using static and dynamic analysis.

LDRA tools do not analyse only what code might do based on static analysis. They can show dynamically what it will do when it is compiled and run on the environment in which it will operate in the field (Figure 7). Just as importantly, they can demonstrate HOW rogue tainted data is handled and confirm that the code sanitization mechanisms surrounding it are functional and effective (Figure 8), particularly with regards to dynamic attack vectors. No static analysis tool alone can do that.

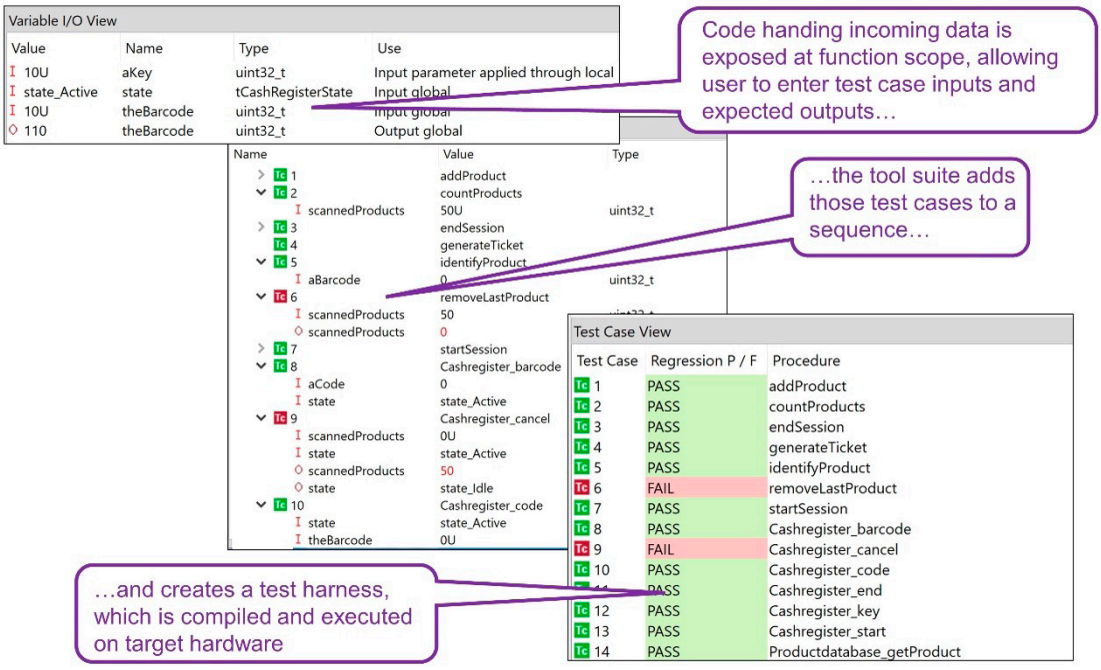
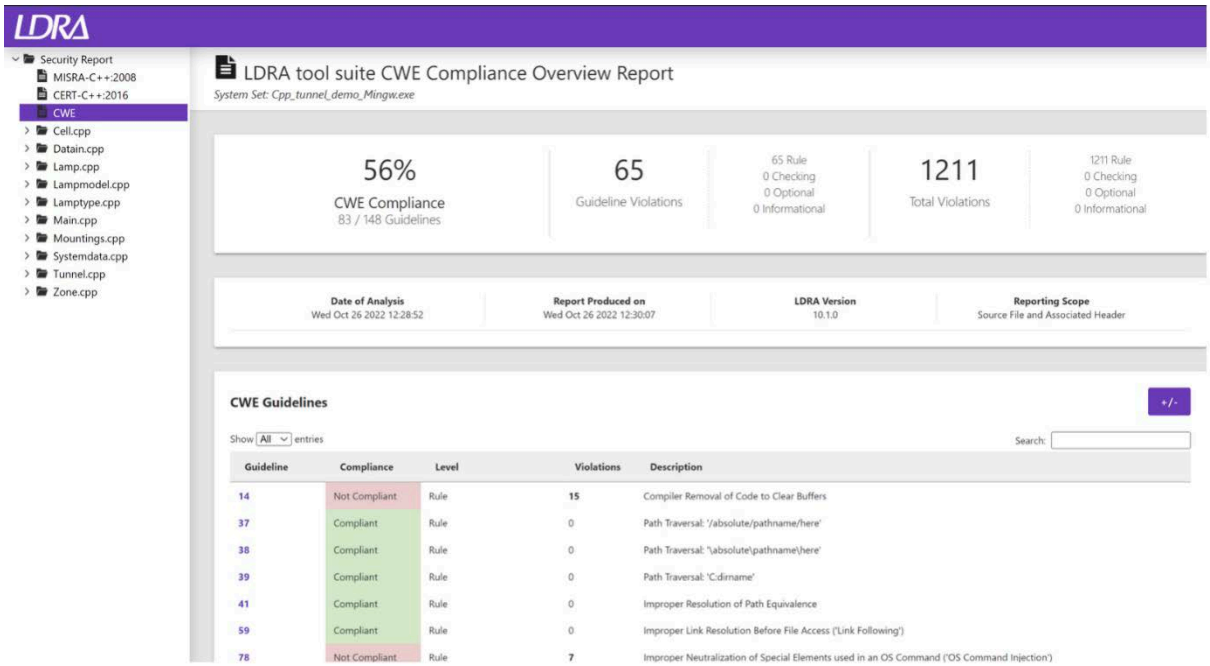


Figure 8: Leveraging unit test techniques to exercise code handling tainted data.

Step 3: Identifying potential security weaknesses

Step 3 involves the identification of potential weaknesses within the code handling tainted data. Clearly the dynamic analysis of step 2 will contribute significantly to that process but again, as illustrated in Figure 5, a combination of dynamic and static techniques provides an optimal solution.

In this case, the unit test techniques are underpinned by static analysis in the form of checking the application of coding standards and guidelines (such as those detailed by MISRA [15]), and weakness lists (typified by CWE [16]). Generally, such documents detail rules are that are applied throughout a code base, but developers may vary the level of vigour with which they are applied to reflect threat categorisation. Code that has been identified as handling tainted data will likely benefit from a more thorough approach.



Step 4: Reporting and mitigation

This final step involves the identification of weaknesses to developers so that they can take appropriate measures to mitigate the risks. This may involve fixing the code, adding input validation, or improving data sanitization procedures.

Summary and conclusions

In a world that is increasingly dominated by connectivity, cybersecurity is increasingly important – both to protect potentially compromising data, and to prevent interference with safety-critical systems. Proactive, "shift left", designed-in security provides a better solution than the traditional, reactive approach to secure software development.

There is no single "silver bullet" answer to maintaining cybersecurity for embedded systems, but secure coding has an important contribution to make to any defence-in-depth strategy. In turn, taint analysis is a key technique in the validation and verification of secure code. Only a combination of static and dynamic analysis as provided by the LDRA tool suite can tackle static and dynamic attack vectors satisfactorily.

Works cited

[1] International Organization for Standardization, ISO 26262-6:2018 "Road vehicles - Functional Safety - Product development at the software level", International Organization for Standardization, 2018.

[2] Chris Valasek & Charlie Miller, "Remote Exploitation of an Unaltered Passenger Vehicle," 2015. [Online]. Available: https://ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf. [Accessed 11th February 2023].

[3] M. Pitchford, "Defense in depth: What does it mean, and what can it achieve?," 12th October 2021. [Online]. Available: https://www.microcontrollertips.com/defense-in-depth-what-does-it-mean-and-what-can-it-achieve/. [Accessed 12th February 2023].

[4] Historic European Castles, "Medieval Castle Defence – Defending a Castle from Siege," 19 November 2020. [Online]. Available: https://historiceuropeancastles.com/medieval-castle-defence-defending-a-castle-from-siege/. [Accessed 15 September 2021].

[5] The Mitre Organization, "CVE Program Mission," 1999-2021. [Online]. Available: https://www.cve.org/. [Accessed 23 March 2022].

[6] Forum of Incident Response and Security Teams, Inc, "Common Vulnerability Scoring System," 2015-2021. [Online]. Available: https://www.first.org/cvss/. [Accessed 23 March 2022].

[7] The Mitre Corporation , “Common Weakness Scoring System (CWSS),” 5 September 2014. [Online]. Available: https://cwe.mitre.org/cwss/cwss_v1.0.1.html. [Accessed 23 March 2022].
[8] Macher, Georg & Sporer, Harald & Brenner, Eugen & Kreiner, Christian., “An Automotive Signal-Layer Security and Trust-Boundary Identification Approach.,” in The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017), Madeira, Portugal, 2017.
[9] R. Seacord, “Top 10 Secure Coding Practices,” Carnegie Mellon University Software Engineering Institute, 2 May 2018. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices. [Accessed 23 March 2022].
[10] I. E. C. (IEC), IEC 64333-4-1 “Security for industrial automation and control systems -Secure product development lifecycle requirements”, IEC, 2018.
[11] OWASP Foundation, Inc., “OWASP Application Security Verification Standard,” 2024. [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/. [Accessed 13th March 2024].
[12] National Institute of Standards and Tecchnology (NIST), “NIST SP 800-53 Rev. 5 Security and Privacy Controls for Information Systems and Organizations,” September 2020. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/53/r5/upd1/final. [Accessed 13th March 2024].
[13] International Organization for Standardization / SAE International, ISO/SAE 21434:2021 “Road vehicles — Cybersecurity engineering”, International Organization for Standardization / SAE International, 2021.
[14] LDRA, “LDRA tool suite,” LDRA, [Online]. Available: https://ldra.com/products/ldra-tool-suite/. [Accessed 23 March 2022].
[15] The MISRA Consortium Limited, “The MISRA website,” The MISRA Consortium Limited, 2021. [Online]. Available: https://www.misra.org.uk/. [Accessed 13 April 2022].
[16] Homeland Security Systems Engineering and Development Institute, “CWE Common Weakness Enumeration,” The MITRE corporation, August 2021. [Online]. Available: https://cwe.mitre.org/. [Accessed 24 September 2021].

www.ldra.com

LDRA UK & Worldwide
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: *+44 (0)151 649 9300*
e-mail: info@ldra.com

LDRA Technology Inc.
2540 King Arthur Blvd, Suite *228*,
Lewisville, Texas *75056*
United States
Tel: *+1 (855) 855 5372*
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit No B-*3*, *3*rd Floor Tower B, Golden Enclave,
HAL Airport Road, Bengaluru *560017*
India
Tel: *+91 80 4080 8707*
e-mail: india@ldra.com