Software Trustworthiness Best Practices

An Industrial Internet Consortium White Paper

Version 1.0 – 2020-03-23

Marcellus Buchheit (Wibu-Systems), Mark Hermeling (GrammaTech), Frederick Hirsch (Fujitsu), Bob Martin (MITRE), Simon Rix (Irdeto)

CONTENTS

FIGURES

TABLES

Untrustworthy software has significant, even life-threatening effects in an industrial context, where trustworthy implementations are required for safe, secure, private, reliable, resilient and functional systems.

This paper provides a high-level overview of software trustworthiness for developers, owner-operators and decision makers in industrial internet of things (IIoT) systems. We address various aspects of creating, acquiring and protecting software. We provide practical and actionable best practices for recognizing, addressing managing and mitigating risks and their sources, whether developed inhouse or acquired.

This paper addresses the complete software lifecycle process, including software development itself. This wider view, in our opinion, is critical in achieving the ultimate goal of trustworthy software that produces predictable and desirable business and operational outcomes.

We address practical issues such as software update and end-of-life strategy needed to enable aspects of trustworthiness and we discuss software trustworthiness concepts at a high level. Details of possible systems decompositions in multiple layers of software and detailed attack surface analysis, required for a real system, are not discussed.

In addition to the application software, vulnerabilities in the system environment where applications are deployed must be considered along with a careful examination of its specifics, its environment and the consequences of failures. Software trustworthiness depends on the existence of specific properties of the operational environment in which the software executes.

Trustworthiness in software can only be achieved with deliberate quality processes for the creation of the product in every stage. Objective, clearly articulated, measurable criteria and their prioritization allows an organization to reconcile competing factors and ensure the required quality throughout the lifecycle from inception to retirement. This defines the required quality and how it will be managed and reviewed throughout software design, development and assurance testing.

Often, one must consider who created and maintained the software, and how this occurred. Again, objective, clearly articulated, measurable criteria and their prioritization allow an organization to make choices about how software is purchased.

Operational software can be compromised. Software protection techniques reduce the operational risks from someone stealing software with the concomitant intellectual property, or from modifying it. They increase protection against threats ranging from buffer overflow attacks to reverse engineering and tampering thereby leading to a more trustworthy solution.

Refrigerated Truck Example

The following example is used throughout.

Figure I-1: Truck with refrigerated cargo container (source: Wikimedia Commons[1]) and Klinge Corporation Redundant Refrigeration System (source: Wikimedia[2])

A refrigerated truck that has a smart air conditioning system for its cargo, controlled by sophisticated software that reduces energy consumption by predicting the outdoor temperature along the route,[3] can be susceptible to attacks and hazards. The software receives the temperature of the cargo area and outdoor temperature readings from different sensors on the truck. UV sensors on all sides of the semi-trailer provide the intensity of the sun.

The software is responsible for the safety of the load. It ensures that the cargo remains within a specified temperature range and warns the driver if it is not, in order to prevent health issues with food. The goal is to reduce the energy for cooling or heating optimized by the weather forecast along the route. The software receives the weather forecast for the planned route from the navigation system and the internet. All components use the truck's Controller Area Network (CAN) bus.

We focus only on design goals to ensure trustworthiness.

[1] See [WikiM01]

[2] See [WikiM02]

[3] Note that this example does not refer to the specific containers shown in the diagram.

1 THE SOFTWARE TRUSTWORTHINESS CHALLENGE

Software is an essential component of many systems. With the trend to digital transformation involving control systems, system optimization capabilities, analytics, human interfaces, design and implementation of components, networking and connectivity, software is now the basis of much more business functionality and flexibility.

Trust and confidence that a system will behave as expected is stress-tested every day, whether from operational issues (expected or unexpected), environmental impacts and system hazards or malicious unauthorized parties. Confidence and trust require organized attention to the key characteristics of a system and the collection of appropriate evidence that the desired characteristics are being achieved.

Trustworthy software is needed to achieve stable and successful solutions in the industrial space. Choices in architecture, design, implementation, testing and operation of the software require analysis and pro-active steps to deliver a system that is reliable, resilient, safe, privacy-preserving and secure. The steps must address analysis, design and code quality for possible weaknesses that could lead to susceptibility to safety hazards, security attacks and theft of the software.

Software and Internet of Things (IoT) trustworthiness: Software trustworthiness is a key enabler of IoT trustworthiness, which is the degree of confidence that a system will perform as expected. Trustworthiness is based on five characteristics—safety, security, privacy, reliability and resilience, which directly and in combination provide protection against hazards and threats related to environmental disturbances, human errors, system faults and attacks.

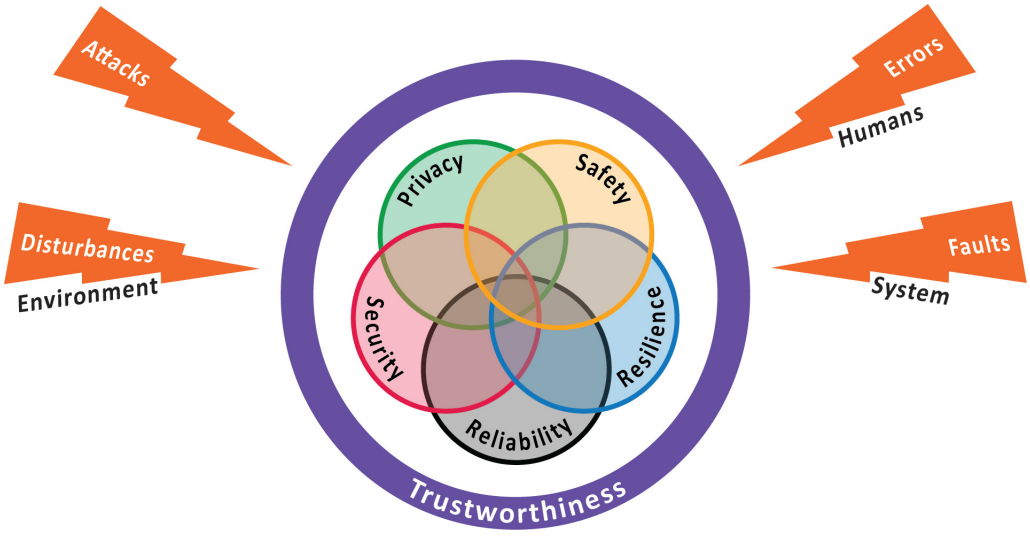

Figure 1-1: Definition of IIoT Trustworthiness

Trustworthiness concerns must be addressed through architecture, design and proper software implementation procedures.

Security concerns that come from active attacks include:

- injection of code, data, or modification of software configurations to bypass policy controls, escalate privileges or inappropriately change behavior,
- reverse engineering to allow an adversary to change or replace the software with operational consequences unanticipated in the original design, and
- insertion of malware, which could lead to a ransomware attack, data theft or system crashes.

Security, reliability and resilience can affect the safety of systems that rely on software for safe operation. The design and implementation also affect safety if the needs of safe operations are not considered as part of the process.

For some types of systems, the software must also be protected from reverse engineering and theft of IP.[1] Preventing reverse engineering, copying of algorithms or extracting other information concern both businesses and some areas of government, for example, protecting military-sensitive logic, performance or guiding parameters of their weapon and defense systems. Other types of system benefit from techniques that improve the 'anti-hackability' characteristics to make it harder to modify binaries successfully for malicious purposes.

Confidence and trust in software also depends on the level of understanding of its design and operation. Although modern software seems complex to most end-users, its development, architecture, assembly, delivery, installation, data processing and overall operation must be put forward and judged for appropriateness given how hazardous and fraught with threats the environment is and the seriousness of consequences when something goes wrong.

Trust and confidence in software is enhanced by knowing that process-oriented activities have been well executed throughout the software lifecycle. It must have well-defined phases and change-management methodologies. Each of these phases needs to be transparent, evidential and auditable. Methods for proving and controlling the provenance of various software components, their configurations and their pedigree further engender trust in the software.

Confidence in software is also reflected in the resulting reliability and resilience of the system. Reliability is affected by poorly written code, such as improper memory management. Resilience[2] is the ability of a system or component to maintain an acceptable level of service in the face of a disruption, such as how long it takes to create and install a software update to address a problem caused by an attack against a latent weakness in the system.

[1] Example, business-confidential algorithms, business-specific *secret* settings or other proprietary choices or knowledge that is contained within the software.

[2] For example, resilient software must be able to handle unexpected inputs, such as unanticipated parameters being entered by operators, *impossible* data values from malfunctioning sensors or an unexpected lack of memory gracefully.

Some software practices and organizational cultures may result in delivering software lacking quality and trustworthiness due to time-to-market pressures. Clearly, there is a need to reduce the costs of earlier phases in the research and product development lifecycle validating the underlying science and engineering principles of the product with a “minimal viable product”, but every release, from the beginning, should address trustworthiness and quality. This is discussed in more detail in the next section.

2 INSTITUTIONAL TRUST AND CONFIDENCE IN SOFTWARE

An organization's culture for quality and its ability to address basic security and safety hygiene enable trustworthiness. They require support at the highest level of the organization, including improving the security maturity of an organization.[1] Software lifecycle processes for acquiring or developing software and their use of tooling and technology to support them, improve quality and address weaknesses before they become vulnerabilities.

An organization must value software trustworthiness and the need to gain and give assurance. This must be communicated both for internal development and evaluations of prospective software purchases. Software quality and trustworthiness cannot be pasted on afterwards by applying automated tools, no matter how good they may be. Using such tools can improve matters, but not as much as a systematic approach to trustworthiness by design and default.

Time-to-market concerns may cause teams to cut corners and give lower priority to software quality and security. Ironically, this often increases the overall cost and time-to-market of subsequent versions through patching and redirecting developers to fix old versions, instead of working on new features. There is also the potential for damage to customers' operational systems and consequential financial and reputational risk.

User trust of software also depends on the transparency and communications about the software lifecycle process. If the software development procedures and the policies applied while creating the software are not easily understood, the user's trust in the resulting technology will be undermined.

These are management and organizational culture issues that need to be addressed before practices to improve the software development process can be applied. If the organization does not communicate the importance of software assurance or follow through with training and corresponding management decisions and priorities, then the software quality will suffer. Once an organization has a culture striving towards trustworthiness, then specific steps can be taken to improve it and an accompanying attention on collecting evidence to demonstrate their ability to deliver appropriately secure, safe and resilient software.

User trust of software is achieved through both its inherent quality and on the customer's willingness to trust the software creation activities, including consideration of the region where the software was produced. Where this trust does exist, implementing the technical best practices will be more effective than when it does not.

[1] See IoT Security Maturity Model: Description and Intended Use [IIC-SMMD2019]

To address these factors as well as enhance accountability and transparency, the following best practices should be considered:

Description and analysis of the surrounding business environment where the software is produced, including legal and political factors, governing laws and norms of behavior.

Transparent communication over the management-related and organizational processes for software development confirmed through an audit or certification, either by an independent third party or self-attestation, depending on your organization's need for independent validation.

Transparent reporting of the company's practices relating to data management practices of their software products, updates and operational connections and interactions back to the developing organization or to a service provider.

Ability to assess the developing organization's internal development and maintenance processes. Software must be managed carefully with change control mechanisms that only allow authorized parties to modify the code and which enables changes to be tracked.

There are several examples[1,2,3] of organizations already implementing these practices.

Trust in software is a direct function of the level of confidence in the management and organization that produces the software. If the organization is not trusted, technical assurance measures to prove software trustworthiness are only partially effective. One must therefore measure organizational, institutional and broader environment-related issues as well as manage the software lifecycle actively as discussed below.

[1] See Microsoft Transparency Centers [Msft-Trans]

[2] See Kaspersky Transparency Centers [Kasp-Trans]

[3] See Huawei Transparency Centers [Huaw-Trans]

3 MANAGING THE SOFTWARE LIFECYCLE

All software lifecycle stages, from inception through design, implementation, testing, deployment and into operation, maintenance, updates and end of life management require an understanding of how software quality and trustworthiness can be achieved and maintained through all the stages. It is also important to know how to assess software quality and trustworthiness, including that from third parties, as well as assessing the organizations and processes used to create it.

The complexity of larger systems hinders the ability of development teams to communicate and manage changes, and so develop trustworthy software of the required quality. The following table outlines some key terms related to the software lifecycle. Inception, architecture and design are included with software-as-written. Updates are included with software-in-delivery.

Term	Definition
Software lifecycle	The process of managing software throughout its entire lifecycle, from stakeholder requirements, system-level requirements, architecture, design, implementation, testing and assurance, deployment, storage, operational use, modifications and updates, through to retirement. Software lifecycle management can be made more trustworthy by using suitable cryptographic techniques to ensure the authenticity of any component of the software or associated information during the initial creation as well as updates. This gives system integrators the ability to track, trace and validate the source, processes and testing used at each step, especially when updating software already in operation.
Software-as-written	Source code is created by software developers during the software development process to guide the behavior of the system, so confidence in the software quality and correct operation is essential. Source code needs to be managed and is typically stored in a source control system at the organization that has authored the software, in an open source repository, when part of an open source project or as a combination of both.
Software-in-delivery	includes the transmission of software to the system on which it will be deployed, including over the air (OTA) updates, as well as installation, configuration and any other actions associated with provisioning the software onto the device on which it will execute. The analysis of trustworthiness of software delivery needs to consider plugins, end-user extensions and configuration by the end user as well as transmission and basic provisioning.
Software-at-rest	This is the software that is ready to execute, either loaded on the device or ready to be loaded on the device. This can be a binary format or source code for interpreted languages (e.g., Python), languages that are compiled into byte code (e.g., Java) or just-in-time compiled. Various cryptographic techniques have been designed to ensure that the software-at-rest has not been modified.

Term	Definition
Software-in-operation	This is the software as it is executing on the device. It consumes memory, has an execution state and is configured into its environment.
Software-end-of-support	Software that is no longer actively maintained by the software development or maintenance teams.
Software-end-of-life	When the software is no longer used operationally by the organization.

Table 3-1: Software Lifecycle Terms

3.1 Risks and Weaknesses in Software-as-Written

Each phase in the software lifecycle can introduce weaknesses that will manifest as risks in the software-in-operation. Typically, software is designed for a specific deployment context.[1] Although source code is the basis of any software system, the trustworthiness of the system also depends on the hardware on which the software is deployed, the communication mechanisms used and the actions of humans through any human-machine interfaces.

Term	Definition
Weakness	is a type of flaw or oversight in the system, software or hardware that, in certain conditions, may contribute to the introduction of vulnerabilities within that system. This term applies to flaws regardless of which phase of the lifecycle they occur.
Vulnerability	is a weakness in software that can be exploited. These weaknesses may be related to the software architecture and design, the detailed implementation or the way the software is deployed into the operating environment. A weakness may be apparent in the early phases of the software development lifecycle in the software-as-written (e.g., memory allocation flaws) as well as be observed later in software-in-operation (e.g., a race condition). Many issues may be detected with appropriate tools and detection methods applied against the software-as-written including those that may occur in software-in-operation.

Table 3-2: Software Risk Terms

The unfortunate reality is that there are always defects in software, such as:

- quality issues resulting from time pressure, lack of adequate training and management approaches incentivizing speed over reliability, security and safety,
- inherent aspects of some (older) languages (e.g., the use of memory allocation and pointers in C) that enable defects,[2]
- ambiguity in language definitions resulting in differences in various production systems depending on the tools used and

[1] We do not address the context in which the software runs (e.g., the hardware, communication mechanism etc.). The IIC Security Framework [IIC-IISF2016] and IIC Connectivity Framework [IIC-IICF2018] documents may be of use for this analysis.

[2] Newer languages such as Swift mitigate many such concerns, as do interpreted languages such as Python.

- issues with the management of the overall software development process, such as not considering software evolution or re-use.

Software quality defects give rise to a number of risks that can be exposed during normal operation, can result from weaknesses chaining to a larger fault or due to attacks. The Common Weakness Enumeration (CWE)[1] is an extensive collection of weaknesses and serves as a common language, a measuring stick for software security tools and as a baseline for weakness identification, mitigation and prevention efforts.

These weaknesses can be injected accidentally or intentionally. The effects are[2]:

- leakage of data,
- changed data or code, including adding malicious code,
- identity spoofing,
- escalated privileges,
- execution of arbitrary commands,
- covering up of activities,
- unpredictable resource usage and
- exhaustion of resources.

Intentional injection of weaknesses into software-as-written is a serious attack vector and it can be accomplished through:

- the source repository,
- third party code (open source or commercial) and
- tooling and subsequently into operational code (binary or interpreted).[3]

Any of these may allow an attacker to change the behavior of a system.

Well-designed software addresses the concerns and risks of the domain as well as the functional goals. The resulting system reflects the decisions based on the understanding of the domain. Real-world systems often comprise *systems-of-systems* where the underlying systems were created at different times, with different goals and under different administrative control. This can hinder creating an overall system that meets trustworthiness goals since a detailed understanding of the subsystems is required, for example, when an existing control system is subsequently enhanced to integrate with additional information systems. This is found in manufacturing in IT and operational technology (OT) convergence. Safety is a primary driver in OT systems, but may not be in IT systems. For this reason, software that was created within the IT domain is not necessarily fit for a purpose in an industrial or safety-critical domain.

[1] See [CWE]

[2] See Enumeration of Technical Impacts [CWE-ENUM]

[3] The classic example of modifying a compiler, see Ken Thompson: Reflections on Trusting Trust [Thomp984]

The figure shows the differing concerns of IT and OT and how they must be considered together in an overall system.

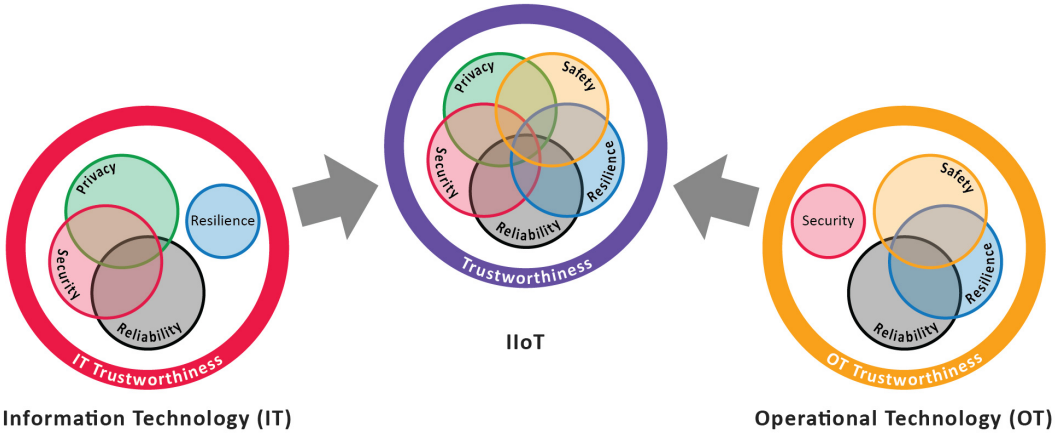

Figure 3-1: Trustworthiness in the Digital Transformation of Industrial Systems

Convergence of IT and OT bears on software since software control loops need to have higher quality and reliability than other software because human life or the environment is at stake.

Example: Early versions of a radiation cancer therapy system (Therac-25[1]) relied on hardware controls to prevent excessive radiation dosages. A later version removed these hardware controls and introduced software controls, which were not sufficiently examined. An error in the software allowed excessive radiation dosages to be delivered that killed patients.

3.2 SOFTWARE LIFECYCLE

The software lifecycle can be organized in different ways. One way, called waterfall, has the phases linearly progressing, one after the other, but with all activities in each phase being addressed before proceeding to the next. Today, organizations follow some form of an agile process, where slices of functionality are delivered rapidly and repeatedly.[2] Flaws and errors in the implementation of one slice of functionality are also cycled as a backlog of items to be addressed in a future cycle.

The figure below illustrates the lifecycle phases and the activities, focusing on iterative cycles of whatever duration. Particular attention is given to show how operation needs initiate an iterative cycle of requirements, design, product and solution validation as well as the underlying iterative cycles with the software production itself showing the flexibility of addressing them using either the waterfall or agile methods or a hybrid approach. The items in this figure are described throughout the remainder of this paper.

[1] See [Therac25]

[2] See ISO/IEC 12207:2008 [ISO/IEC 12207] and ISO/IEC 15288:2015 [ISO/IEC 15288] standards: system and software engineering – system life cycle processes

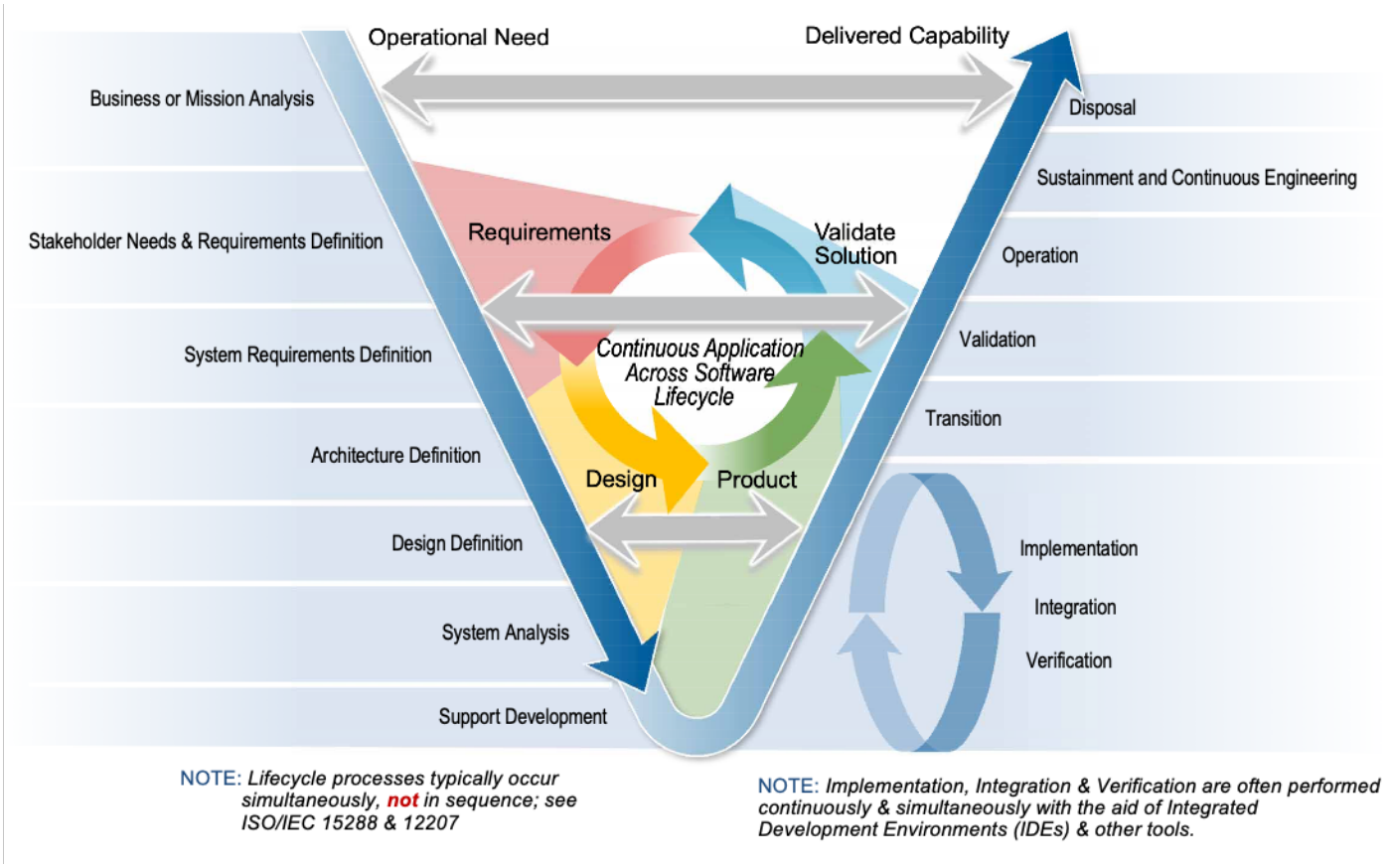

Figure 3-2: Software Development and Assurance Lifecycle Phases

The *inception phase* is where the idea for the software is conceived and the business operational motivation for the software is elaborated. This is followed by a requirements phase, an architecture and design phase, a product implementation phase and a testing/validation phase.

Deployment phase: Software is then delivered into operation, maintained and updated during deployment until it is retired. These phases are not necessarily discrete and separated in time. Requirements, design and implementation are often worked on together. Updates to software or any other elements, such as configuration files, require testing.

For software to remain trustworthy, the lifecycle must be managed carefully, and an audit trail maintained. The audit allows the software user to understand what changes have been implemented between versions, who made these changes and why.

Deployment involves transmitting software from creator to user and the receiver must be able assess whether he has received the correct software package and what is contained therein. The former can be achieved by signing the software deliverable and the latter by providing some form of bill of materials for the software, which should be linkable to the requirements, traceability, testing and assessment efforts, and audit trail information.

Software end-of-life and disposal: At the end of life, the producer no longer supports the software, but the operation often continues to use the software, but this is at their own risk. we do not recommend continuing to operate end of life software as this may cause functional and security risks to the operator as well as security risks to the ecosystem. The migration of software to open source may solve some of the problems that the operator may have when they continue to use end of life software.

Weaknesses in software can be sought and found in all phases of the lifecycle, but the detection methods and examined artifacts differ. If the early phases are supported by tools and models, then those can be examined for architecture and design weaknesses automatically, while if that phase is captured in documents and presentations, then people must identify those weaknesses in design reviews. As the software is written, tools that examine source code and later as executable software are available. Tools that can test running code or examine binaries can also be brought to bear.

While going through its lifecycle, software constantly changes as new features are added, bugs fixed, weaknesses addressed. These changes lead to a new version of the software that needs to be deployed to a device or service. Traceability is necessary to understand how software has changed and is linked to requirements. This is useful for auditing software as part of determining that it meets requirements and that additional unnecessary capabilities have been added.

Secure software updates are needed by producers to be able to update functionality, security risks and other issues. There needs to be a way for the software users to report vulnerabilities back to the creator. This could be an automated system or a manual vulnerability and defect reporting system.

3.3 Software Requirements and Validation Criteria

The operational needs are often non-technical and must be clearly articulated and approved prior to setting out the corresponding technical requirements. Many problems are caused by a lack of clarity in communication between business stakeholders and software developers.

Validation criteria should be articulated early to ensure that end customer and software architects have successfully communicated and captured all requirements. They are the starting point for the validation tests that are required to approve the final software product. The earlier in the process one begins to think of testing, the better.

Changes in requirements almost always cause re-architecting and re-designing software, which is hard to do without introducing weaknesses. It is therefore imperative not to short-cut this phase or leave validation planning for later.

3.4 Software Architecture and Design Process

Half of the CWE collection relate to software design.[1] Automated verification may ease detection and correction of these design flaws, but unfortunately, it is difficult to automate and relies on using a model-based approach. Education about how these weaknesses affect a system and peer-reviews remain the most common but effective ways of avoiding design weaknesses.

The goal of the software design and specification activities is to formalize software properties and behavior with respect to their support for the trustworthiness characteristics of system so they can be verified through some type of assessment or test activity thereby providing evidence that the requirement has been met.

Software protection needs to be planned at the design stage as the design and structure of the software affects the levels of protection that can be achieved.


Example

Refrigerated Truck Example (continued)

The software architecture must be easy to verify and test that the safety of the cargo has a higher priority than energy reduction and that measures are taken to keep the temperature within permissible maxima and minima constraints.

There must be clear mechanisms to raise timely early warning alarms to enable the truck operator to take remedial actions as well as mechanisms and methods to log and record measurement, decisions and remedial actions.

The design needs to articulate clearly the rationale and reasons why certain design decisions were made, such as the number of temperature sensors and their location.

To enable independent testing of software modules the design needs to allow different modules to accept input from the same temperature sensors.

Software protection techniques need to be applied from an early point in the cycle to ensure that the design is protectable, to allow early validation of performance metrics and to enable for early validation of the achieved level of protection.

3.5 Software Implementation (Coding) Process

For larger projects, multiple teams must work across the organization, requiring coordination and communication. With agile development, requirements refinement is broken up across the iterations, with just enough definition and refinement to support the current iteration.

[1] See Weaknesses Introduced During Design [CWE-701]

Software development often requires a multi-stage build process in which numerous source files, binary executables and libraries, configurations and related files are combined to create a system image.[1] Each of these steps in a build (compilation, linking, configuration, etc.) provides an opportunity to check for weakness and validate that the quality of the software design, code and executable are meeting their trustworthiness goals. The information associated with these steps can also be recorded in a software-bill-of-materials (SBoM) creating a record of how software was created and configured. This information can be used to enhance the confidence in the software.

Once ready to ship this image, the software-as-written is transferred to the targets to become software-in-use. The image comprises various components, including source language files (either for compilation or interpretation), binaries produced from the build (with different results depending on build options), configuration and auxiliary files. All components must be managed so that the correct versions are used, changes are managed and to ensure they work together.

Change is usually managed with a version control system (e.g., git, RCS, SVN or CVS). The design work may produce electronic artifacts (e.g., if UML is used), but more likely depends on written documentation and this, in fact, may be one of the issues with software quality if this documentation is incomplete or not maintained as software is revised.

Periodically the software-at-rest is completely re-built using all the source material (source files, configuration and other meta data) as part of an update, a later part of the lifecycle. This software is then executed and becomes software-in-operation. This means that management of software implementation is relevant throughout the software lifecycle. When severe operational, safety or security relevant errors are found in the software-in-operation, a revision may need to be developed and deployed quickly, but in a trustworthy manner. This is one of the most expensive places to find errors as patches are unexpected and require unanticipated work and reallocation of skilled staff.

The reliability and correct operation of software is essential to safety and security and is best addressed early on in the software lifecycle whenever possible but must also be quickly addressed whenever it is found in software that is in operation.

Finally, there are many software compilation and creation options in today's compilers and build tools that can add protections against some types of attacks, for example, *address space layout randomization* and *data execution prevention.* Solutions that monitor for stack overflows or invalidate writes to adjacent stack memory locations can be used to mitigate some risks.

There are multiple ways to transfer software-at-rest from the build environment to the execution environment: in conjunction with hardware creation, configuration and delivery, in the field as a

[1] The same principles apply to interpreted code, when considering imported modules and possible linkages to compiled libraries.

direct copy via a USB stick or by an over-the-air transfer. Every mechanism must ensure the integrity of the delivered software and its authenticity.

Automation will improve software quality and therefore the trustworthiness of what is deployed. Many different software development tools may be used during the development process, including compilers, static testing tools, integrated development environments, and automated dynamic testing tools.

There must be a means to trace the software-in-operation to the included software components and configuration used to construct it so that a defect, such as a software vulnerability, can be fixed. This should include the metadata describing attributes of the components and the tools used.[1] If the defect was introduced through a change then, as a way to improve the process, previous versions of all the source, configuration and other build files should be examined to determine how the vulnerability causing change was introduced. This understanding can be used to improve the process.

This type of information about the processes and tools used in building the software is part of the pedigree information in the software-bill-of-materials of the software-in-operation.

3.6 Software Assurance

To enhance or achieve superior levels of trustworthiness the software development process and the quality of the software itself must be evaluated using software assurance practices.

Detection methods for locating unintentional weaknesses in the software architecture, design and code need to be applied from the beginning of the lifecycle and continuously throughout.

Term	Definition
Software Assurance	is defined by the Committee on National Security Systems as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner". Some alter the end of this to "only in the intended manner", which aligns with safety and reliability concerns.

Table 3-3: Definition of Software Assurance

Many different techniques are used to assess software assurance, including those described in the table below.

These assurance assessment techniques are often supported by tools that can automate all or part of the process. Each has a different scope (though some overlap) and can contribute to an understanding of software quality. They are used in three testing categories: static testing, dynamic functional testing and security testing.

1 In older environments this knowledge may reside in the expertise of the software developers and not be maintained as explicit metadata (e.g., knowledge of the build system and Makefiles for example).

Software assurance does not end when the system is deployed. As defects are reported, either for functional, safety or security issues, the software will need to be modified, the assurance process will need to run its course and the software-at-rest needs to be updated. Monitoring during operation can be used to detect anomalous situations enabling timely response as well as further software revision to address the underlying flaws.

Technique	Description
Risk Analysis	An analysis of all the vulnerabilities of the system under examination, commonly a reflection of the complexity of the system, including the number of interfaces.
Design Review	A review of a design against requirements to identify completeness and correctness.
Code Review	A review of software code against a design and requirements to identify completeness and correctness, proper style and documentation and to identify software flaws and vulnerabilities.
Static Analysis	Analysis of source code to identify flaws and vulnerabilities, often an automated process.
Fuzz Testing	Varying input parameters to software functions, often outside of expected bounds, to validate software error handling and identify potential vulnerabilities.
Unit and Functional Testing	Analysis of executable code at runtime to determine whether it meets the functional requirements. This includes unit testing, integration testing and system testing reflecting increasing levels of completeness of the system. Testing can be integrated into the implementation process as in *test-driven development* where test cases are created as part of the development process and used throughout to verify correct implementation and avoid regression.
Security Testing	Security testing includes: *Pen Testing* by friendly teams to determine vulnerabilities that can be exploited. Also known as *penetration testing*. *Blue Teaming* by friendly teams who have access to all relevant documentation. *Red Teaming* by adversarial teams who have no prior knowledge of the design.

Table 3-4: Software Assurance Assessment Techniques

3.6.1 Risk Analysis

Attack surface analysis should occur throughout the software development lifecycle. As soon as the system concept is established, it should be analyzed to determine if adjustments to the conceptualization could influence the trustworthiness of the system.

The scope of the attack surface analysis needs to broaden as we move from a security focus to a trustworthiness focus. For example, a privacy analysis ensures the use of personal data collected and used is minimized; a hazard analysis could highlight safety concerns; and a reliability analysis of the hardware and software could predict the need for maintenance and downtime.

As the system and software architecture and design are laid out, they should be reviewed for security, privacy and safety. The confidence in the analysis depends on how explicit the

architectural and design decisions are. When the architecture and design are captured formally, formal model tools can perform these tasks and capture the results.

3.6.2 Code Review and Static Analysis

Software developers may accidentally introduce security, reliability and safety weaknesses into their code. Computer programming education has traditionally focused on getting software to work without teaching aspiring programmers about dangerous constructs that can allow attackers or unexpected situations to affect their code.

The CWE list is a community effort to categorize and document weaknesses that can occur in architecture, design and implementation.

The first defense against these weaknesses is typically a peer code review, where the source code is reviewed with other programmers and approved before submission to the project code base. Code reviews help avoid simple mistakes and multiple people reviewing the code reduces the chance that malicious code is added to the repository.

Static analysis is an automated technique to examine the structure and weaknesses of source code without executing it. Static analysis finds deep problems that are easily missed by a code review. Both code review and static analysis discover problems before the software is deployed as software-in-operation, reducing the risks and costs of remediation. Both have their uses and should be part of any software assurance process.

The integration of basic (quick) static analyzer tools in the *integrated development environments* (IDEs) used to create and edit software is very effective in preventing a significant number of issues as this gives immediate feedback to the developers whenever they add insecure, unreliable or unsafe constructs into their code.

More advanced static analysis can also be integrated into the build and integration process, enabling more extensive analysis of the entire deliverable, where data flow analysis and other more resource-intensive analysis techniques can be applied to each build cycle. Thus, a build can detect quality issues through a variety of means, such as build failures (e.g., unable to compile source), static analysis issues and unit test failures. For DevSecOps, where security, operations and development teams work together, this ends up as a continuous analysis within the *continuous integration/continuous build tool chain*.

The problems found by code review and static analysis tools include violations of coding standards, variations of known dangerous constructs in specific software languages or suspicious code. Some violations will affect software quality while others affect trustworthiness.

Violations of coding standards: Software teams use coding standards to deliver high quality code that is easy to read and understand. Coding standards could be defined by the team itself or they

could use one of the well-known standards such as JPL's Power of Ten,[1] MISRA[2] or the Barr Group's Embedded C Coding Standard.[3] A violation of a coding standard does not immediately indicate a problem, but it does indicate that the process is not being followed meticulously and the specific code could be misinterpreted by others, perhaps during modification.

Use of dangerous constructs: Most languages have dangerous constructs. Examples of these vary from reading or writing outside of a previously defined buffer, using a variable with an uninitialized value, the use of sensitive *cookies* and poor use of encryption libraries. The CWE list is a great resource to understand the risks. The 2013 standard ISO/IEC TS 17961 "C Secure Coding Rules"[4] describes many C language issues and constructs to avoid. Using these dangerous constructs can lead to runtime errors, such as data leakage, command injection and program crashes, affecting the confidentiality, integrity and availability of a system. Code reviews cannot detect many dangerous constructs, nor can they be found by dynamic testing. Thorough static analysis is highly recommended to detect these dangerous constructs.

Suspicious code: Suspicious code looks *off*. For example, code that can never execute (also known as dead code) or a switch statement that does not cover all possible values of the parameter. While not immediately dangerous, it does indicate code that should be reviewed to ensure that the code reflects a correct design and implementation.

3.6.3 UNIT TESTING

Unit testing quickly provides confidence that smaller components can be integrated. Unit testing typically focusses on validating that both the happy path and the error paths are correctly implemented. For example, successful login and login attempts with incorrect passwords should be tested. Security focused testing is typically excluded from functional testing.

3.6.4 FUNCTIONAL TESTING

Before the software is used in production environments it needs to be tested to ensure that the software indeed behaves as the requirements described. Software development teams typically create automated test cases to validate the functionality of the software. These test cases are part of the software-as-written as well, even though they are likely not included in the deployed software-at-rest. For organizations concerned with the resilience, safety and security of their software-in-operation, this is also the portion of the lifecycle where misuse and abuse testing is conducted to find aberrant behaviors that need to be eliminated.

Functional testing ensures that the functional behavior of the application satisfies the requirements. It is typically separated into unit testing, integration testing and system testing,

[1] See [Holzman2006]
[2] See [MISRA]
[3] See [Barr]
[4] See [ISO-TS17961]

each providing additional levels of system completeness. Functional testing should be automated, test failures should be inspected and resolved as quickly as possible.

Dynamic testing should be run continuously as the system evolves, including deployment and maintenance.


Example

Refrigerated Truck Example (continued)

Efficient testing requires the ability to input sensors and to monitor control output values or any commands sent to other pieces of equipment.

A design and test methodology that allows for the development of test sets and software algorithms in a virtual (or modeled) environment allows for the rapid debugging and validation of both. System level tests can be reused to validate the final hardware product.

The analysis of test results should be automated as the quality of repetitive human evaluation degrades over time.

3.6.4.1 Security Testing (AKA, Misuse and Abuse Testing)

While functional testing is focused on the correct operation on the system, security testing is focused on trying to subvert the system by misusing it and abusing the functionality in an attempt to get the software to do something it was not intended to do.

Fuzz-testing, or fuzzing, is a technique in which the system is fed unexpected inputs, typically automatically. This works well for systems with process streams of information such as network systems. Fuzzing applies different types of inputs to the system to try and drive the system through new code paths in an attempt to trigger a problem.

Blue and red teaming are manual efforts to test the security of a system through teams of experts. The blue team works on the system from the inside to identify security flaws, verify security measures and defend the system against malicious use. The red team is typically an external team that attempts to infiltrate the system from the outside. The blue team works inside the attack surface while the red team works from outside the attack surface.


Example

Refrigerated Truck Example (continued)

System modeling greatly aids security testing as one can now disturb a model more invasively than a real device on a system. Plus, any additional test functionality inserted into the design is never used in a production build.

Predictive tests need to be developed that can accurately predict the expected resultant actions of the systems allowed with a set of acceptable tolerances. For the truck, we should generate various combinations of temperature measurements and alter incoming weather data and door position sensors. Where these input devices are external to the main electronic control unit, the communication links between them need to be stress tested.

4 TRUSTWORTHY SOFTWARE OPERATION

4.1 TRUSTWORTHY TRANSFER OF SOFTWARE

Once software has been created and tested, it needs to be transferred to the hardware where it is to be run and appropriately configured.

Plan how you wish to support in field updates from the beginning.

Trustworthy software updates: We need to fix bugs and allow for possible feature updates over time. Software should be updated securely and authenticated. Updates also reduce support costs by ensuring consistent versions of software in the field. They need to be performed securely.

Typically, software updates are implemented using hardware-based root-of-trust and a secure bootloader, to protect the device from accepting software updates from any unauthorized source.

Software needs to be protected from alteration by malicious parties and this needs modern cryptographic algorithms and protocols to protect the transmission and on device storage of the executable. Often, different techniques are chosen for transmission and on-device storage (a type of the data-at-rest problem domain).[1]

There are many issues that affect software updates in addition to the choice of the cryptographic algorithms and protocols. You should:

Allow the device to download the updated version of software while it is still functioning, using a prior version of software, so you can minimize device down time due to re-boot, and allow the device to revert to a prior build in the event of operation problems. This must be carefully controlled as there can be a requirement to prevent or restrict the ability to roll back to earlier versions of software to allow adversaries to take advantage of weaknesses already remedied.

Within reason, *allow the end user to configure when the device updates*. Clearly some updates need to be forced and users may need to be given a more restricted time window from which to select.

Automated software updates are ideal, but this is not always possible. In this event endpoint users need to:

- easily find out that an update is available,
- be able to find and download the update easily and
- be able to schedule when to update easily within the parameters described above.

[1] The choice between various asymmetric and symmetric techniques to ensure authenticity, provenance and possible secrecy is complex and beyond scope. Similarly, key provisioning for use in secure bootloaders.

Where feasible, *allow the user to select only security updates without new features*. This complicates support as many software versions are in use, but this may be easier than helping users through an upgrade process.

4.2 TRUSTWORTHY CONFIGURATION OF SOFTWARE

Configuration risks and weaknesses: Software is usually configurable, especially software that is broadly usable, and can be adapted through configuration settings for the specific functionality needed for the system. Unfortunately, many involved in deployment pay inadequate attention to selecting, implementing and monitoring the configuration of the software they use. This leaves many software systems attackable in ways that the development team had provided protections against, but the deployed software is not using. The CWE collection contains many examples of inappropriate configuration.

An active log of all configuration activity, when each configuration was applied to specific machines or devices and the active software version, should be maintained as a useful source of analytical information in the event of any operational incidents.

4.3 TRUSTWORTHY SOFTWARE-AT-REST

Software-at-rest is software that is configured, installed and ready to operate but not yet operational.

While most systems run on well-tested commercial platforms, some run on dedicated lightweight real-time platforms that may be untrusted. If not properly secured, any software running on these systems may be reverse-engineered using readily available tools for those environments, making the code an open book to anyone caring to read it. The person creating the software will probably not be the one making updates and corrections, which is why developers are taught throughout their education to write clean, streamlined, well-documented code that is easily testable and maintainable by others.

This is an attacker's paradise. While there are obfuscation tools available, modern software reverse engineering tools have advanced significantly to the point where most obfuscation methods are ineffective against a focused adversary. The price of these tools has dropped significantly. Today a hacker can purchase a single seat of professional reverse engineering tools for an affordable price or make use of the open source Ghidra toolset[1] made available by the National Security Agency in April 2019.

These tools provide an affordable and accessible set of tools for attackers to use to gather a detailed and accurate understanding of your valuable algorithms, program flow, variables, data and even cryptographic keys.

[1] See [Ghidra]

Depending on the type of software, there may be huge commercial risk to the IP of the software product, which requires more advanced software protection techniques discussed later.

4.4 TRUSTWORTHY SOFTWARE-IN-OPERATION

There are several issues that need to be considered with software-in-operation.

First, has the software-at-rest been modified? The validity of the software-at-rest should be checked each time prior to execution, including that the latest version of the software is being executed or at least that the version is acceptable. Typically, this is done using public key cryptography where the version of software and relevant metadata are signed by an offline entity to ensure that the latest software version is used in operation. Where symmetric cryptographic methods, such as device identifier composition engine architectures[1] are used, keys require *trusted platform module* functionality to keep the symmetric keys secure.

The version of software executing in the computer's memory must be untampered. This can be checked by internal integrity verification techniques and cross-checking that the software-in-operation is the same as the software-at-rest. This rapidly evolves into the realm of software protection whereby the act of comparing the two versions needs to be hardened or defended. The same is true for any data or metadata associated with the executable software.

Operational risks and weaknesses: The business impact of poor-quality software, whether from careless inclusion of weaknesses or from malicious insertion, is in the operational use of that software. All of the weaknesses in the CWE[2] collection resolve down to eight technical impacts in the section entitled "Risks and Weaknesses in Software-as-Written" above, that come from when the weakness manifests in the operational software and is exploited.

Depending on what the software is doing in the operational context, the relevance of these eight technical impacts ranges from critical to unimportant for the software and the operational capability it is supporting. The scale of impact allows a software development team to focus on those weaknesses that can cause the most dangerous operational risks.

Trustworthy software retirement: When software is to be retired, care must be taken to delete data that may be misused securely. Data should therefore be bound in some way to the software and rendered indecipherable in the absence of it.

[1] See Trusted Computing Group, DICE Architectures [TCG-DICE]

[2] See [CWE-701]

5 SOFTWARE PROTECTION

Intellectual property (IP) sets a company apart from the competition It comprises business-confidential algorithms (or "secret sauce" if you wish) and a wealth of practical know how and effort required to create complete products that produce economic value. All of this IP is incorporated into an organization's software.

Any software of value is vulnerable to theft enabling attackers to learn business secrets as well as special algorithms and secrets that make up the *wow* factor. Software can also contain specific proprietary information such as the specific production and mixing factors in the control software that runs the production line used to create a product.

The ease of reverse engineering executable software comes as a shock to many and needs to be addressed to protect the economic investments made to make businesses successful.

Term	Definition
Intellectual Property	Something of value that is incorporated into or represented in the software. For example, this may include algorithms, business logic, proprietary parameters, software design or models.
Software Hardening	This encompasses all types of efforts taken to ensure that software is not susceptible to attack. Software hardening is therefore the combination of software assurance and software protection.
Software Protection	Efforts taken to ensure that software is not the target of an attack. Software protection consists of a broad collection of principles, approaches and techniques intended to increase protection against threats ranging from buffer overflow attacks to reverse engineering and tampering. Software protection techniques aim to make much of the functionality too complex to reverse engineer and modify as well as protecting algorithms and IP. Software may be protected both for software-at-rest (static software protection) and for software-in-operation (dynamic software protection). Software confidentiality may also be used to protect software-in-delivery as well as software-at-rest.
Software Confidentiality	Consists of protecting software from unauthorized examination. Such confidentiality can prevent examination of the software for vulnerabilities that can be used for attacks and also to prevent the understanding needed to make modifications. Software confidentiality can also contribute to protecting IP. One approach toward software confidentiality is encryption of the software.

Table 5-1: Terms of Software Protection

Software protection seeks to ensure software is not the target of an attack. Software protection is implemented to add specific well-defined defensive properties and should be applied after all the 'software lifecycle management' processes and techniques have been executed.

Software protection has the following goals:

- to make software difficult to understand to conserve the investment in software,
- to prevent tampering with the executable to ensure functionality is unchanged,

- to provide active validation of the environment within which the software executes,
- to protect the data in use as software executes[1] and to bind the data and metadata to the executable software so that the data is indecipherable except to the software.
- to detect tampering or substitution of the data, the software needs to be able to detect any alteration of the data or metadata in a well-defended manner. This is helpful in General Data Protection Regulation (GDPR) compliance,
- to hide cryptographic keys and calculations in an insecure environment using whitebox cryptography[2] and
- to improve the operational security surrounding the deployment of bug fixes.

Software protection techniques improve the level of hack resistance.

Software protection solutions entangle code and data by hardening techniques such as control-flow flattening, function merging, function in-lining and data transformation. Entanglement can be applied algorithmically at the source level such that nothing can be modified without affecting the control flow or data flow of the program by software transformation[3] while retaining functional equivalence.

In the physical world a brittle object breaks into many pieces when one attempts to modify it or use the modified object for its intended purpose. Software protection techniques result in brittle executables that are hard to modify without the modified software crashing.

Brittleness and tamper resistance are enhanced by injecting active defences to perform specific defense in-depth functions such as integrity verification anti-debug, rooting/jail-break detection and anti-hooking as part of the software transformation process, thereby entwining the active defenses as an integral part of the software transformation process.

Attacks are varied and morph rapidly, so defenses need to be highly adaptive. Given that software protection uses a variety of techniques that are blended together and can be applied in different ratios, the result can be tailored or customized to achieve the required defensive characteristics.

When bug fixes are distributed, attackers perform differential analysis on the released executables to learn about the now-repaired vulnerabilities. They then use this knowledge to attack not-yet-upgraded systems. Software protection obscures what has been repaired to give more time to upgrade devices prior to differential attacks based on the newly discovered bugs. Source-based software protection solutions can create diverse builds each time. This is beneficial as it forces the attacker to analyze every release.

Software protection obscures faults for both the development and maintenance teams if binary debugging is required. Automated system-level tests should be used to test executables that

[1] See A Survey of Anti-Tamper Technologies [Bryant2004]

[2] See IIC: Data Protection Best Practices Whitepaper [IIC-DPBP2019]

[3] See More than meets the Eye... Software Transformation vs Obfuscation [Goodes2018]

have been built using an unprotected source as well as the protected executable version. This allows debugging to occur prior to applying software protection and for automated verification after software protection has been applied. Such an automated development methodology enables DevSecOps environments.

Good software protection techniques force attackers to perform more complex static analysis and run-time attacks. A multi-dimensional blended set of defensive techniques, including methods to resist static and run-time analysis, greatly increase the costs and timescale of an attack and reduce the size of the skill pool able to undertake such advanced hacking.

Too frequently, the security gaps between the software implementations and the physical system are poorly understood—this is why full threat risk analysis is needed across the entire ecosystem at the start of a design. Hardware roots of trust may not be available for a desktop software environment. These gaps, if not addressed, can lead to threats to the system as well as the manufacturer's business model. Software protection may then become the only readily available defensive tool.

The economic perspective motivates the amount and type of software protection applied to:

- increase the cost of an attack to the point where the attack is no longer economically attractive for the attacker. This needs to be balanced with the cost of adding the defensive feature(s) the first time, and the ongoing cost of protecting repeated bug fixes and functional improvements,
- maximize investment return by hiding the secret sauce used to optimize your system to maximize the time it takes for competitors to catch up and
- encourage potential attackers to attack more vulnerable targets elsewhere

Software protection, as opposed to obfuscation techniques, may require some explanation, so let's unpack this a little. Using reverse engineering tools, an attacker first attempts to understand the static structure of the executable being analyzed. These tools create call graphs[1] which help enable an attacker to understand the control flow of the executable being analyzed. Call graphs are a graphical representation of each node (we use circles or ovals below) to show the relationship(s) between nodes.

In the call graphs below, the colors illustrate where the node (or procedure) occurs in the protected version and maps it back to where the procedure existed in the unprotected build. The control flow graphs below illustrate the effect of control flow flattening. The control flow of a simple function is shown on the left—it is a simple structure and easy to understand. The right-hand graph shows the same function after control flow flattening. When faced with a flattened structure, the attacker gains no clue from the structure itself and they must make much more effort to understand the executable.

[1] a good introduction is found at Wikipedia: Call Graph [WIKI-CG]

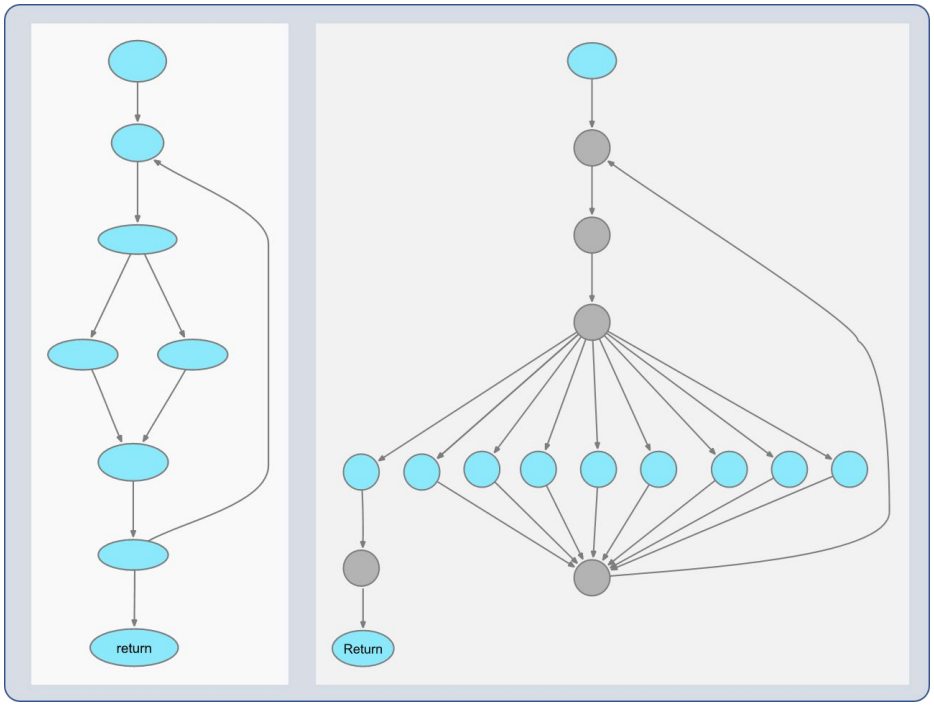

Figure 5-1: Control Flow Graphs Before and After Control Flow Flattening

As other techniques are combined to increase the complexity faced by the attacker, the attack difficulty escalates. An example of this is where function merging (the intertwining two functions of unrelated functionality) is combined with control flow flattening. Such software transformations significantly increase the complexity an adversary must overcome and is shown in the call graphs below:

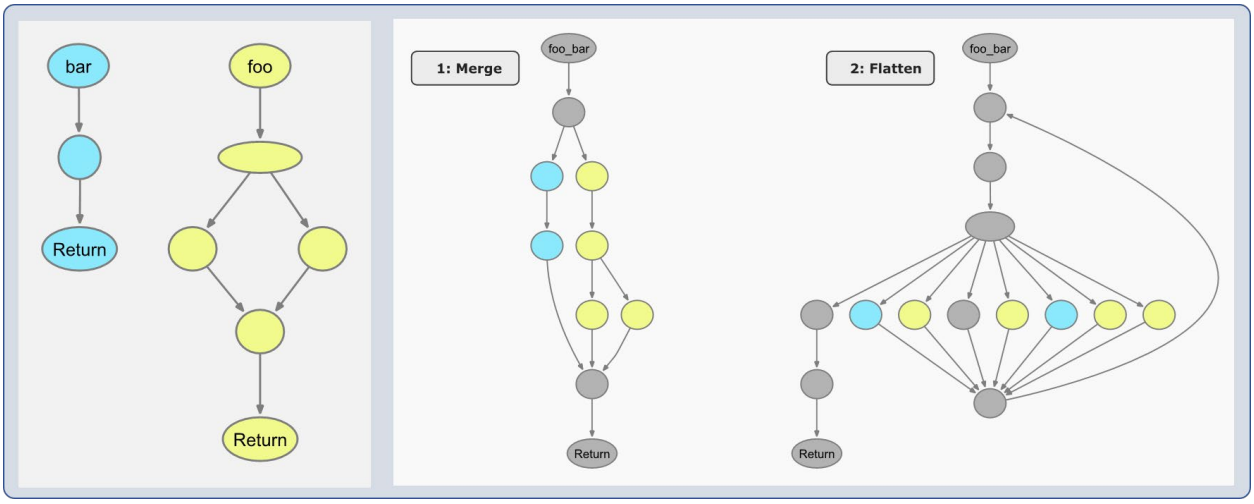

Figure 5-2: Function Merging Followed by Control Flow Flattening

Nice, clean and well-partitioned code can be further transformed by hiding well-structured and efficient function calls by replacing calls with the functions themselves and then applying control

flow flattening. This hides external function calls. Thereby helping to destroy the structure of the now protected program.

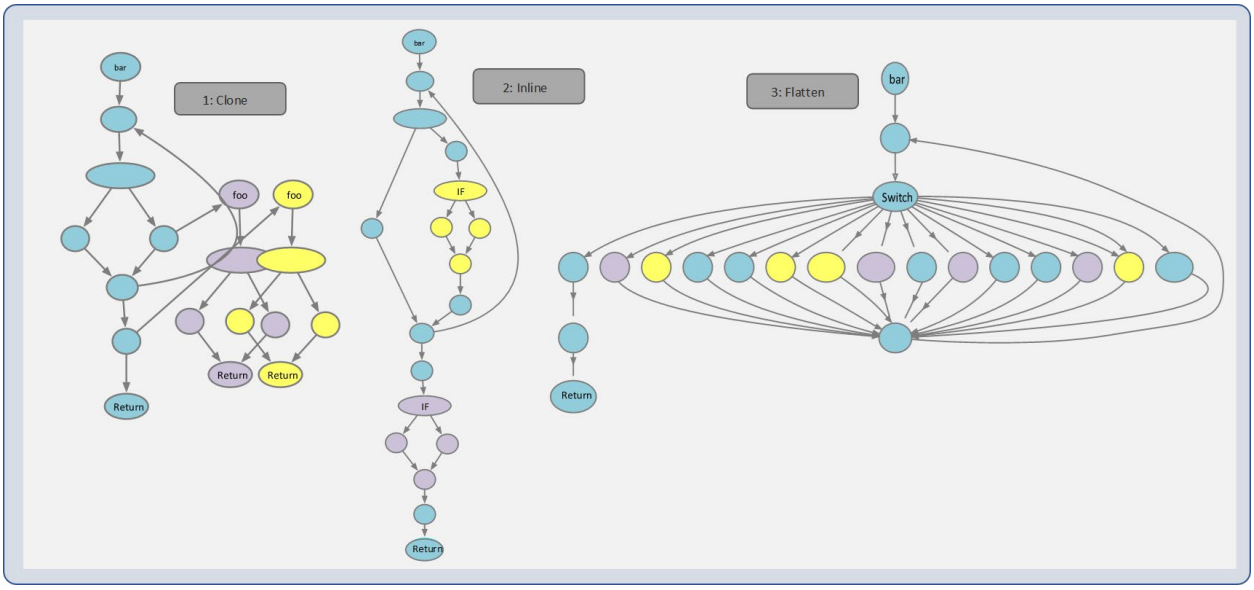

Figure 5-3: Function In-lining Prior to Control Flow Flattening

Using a well-known reverse engineering tool, we can show the effect of using control flow flattening combined with data transforms.

```
int strcmpish(const char* buf1, int len1,
              const char* buf2, int
len2){
    int i;
    if (len1 != 16) {
        return 0;
    }
    for (i=0; i<len1; i++) {
        if (buf1[i] == '\0') {
            return 0;
        }
    }
    if (len2 != 16) {
        return 0;
    }
    for (i=0; i<len2; i++) {
        if (buf2[i] == '\0') {
            return 0;
        }
    }

    if (len1 != len2) {
        return 0;
    }
    for (i=0; i<len1 && i<len2; i++) {
        if (buf1[i] != buf2[i]) {
            return 0;
        }
    }

    return 1;
 }
```

```
while ( 1 )
{
  switch ( *(_DWORD *)r_65c + 835940861 )
  {
    case 0:
      av_538[6] = -1228287702LL;
      *(_DWORD *)r_65c = -835940856;
      break;
    case 1:
      av_538[6] = -1228287703LL;
      *(_DWORD *)r_65c = -835940856;
      break;
    case 2:
      return;
    case 3:
      if ( *((_DWORD *)av_538 + 6) == 16 )
        v1 = -835940854;
      else
        v1 = 0;
      *(_DWORD *)r_65c = v1 - 835940860 * (*((_DWORD *)av_538 + 6) != 16);
      break;
    case 4:
      r_5s3 = r_5rt + 1228287688;
      r_5sT = r_5rt + 1;
      if ( (int)(r_5rt
               + 1228287688
               + ((unsigned __int8)(39 - r_5rt) ^ 0x10 | 0xFFFFFFEF)
               - ((r_5rt + 1228287688) | (-1228287705 - r_5rt) ^ 0x10)) >= 0 )
        v2 = 0;
      else
        v2 = -835940853;
      *(_DWORD *)r_65c = v2 - 835940855 * ((int)(r_5rt + 1228287688
                + ((unsigned __int8)(39 - r_5rt) ^ 0x10| 0xFFFFFFEF)
                - ((r_5rt + 1228287688) | (-1228287705 -r_5rt) ^ 0x10)) >= 0);
      break;
    case 5:
```

Figure 5-4: Control Flow Flattening Combined with Data Transforms


Example

Refrigerated Truck Example (continued)

During the design process, designers need to identify data, calculations, decisions and configurations that require protection and determine how to harden the code. For example, using software protection to harden the algorithm that ensures the safety of the cargo has a higher priority than energy reduction hardens a vulnerable and critical decision-making function. Using data transforms to protect variables in use and then entangling that with the decision-making sequence makes it harder to alter.

Binaries must be pen tested to validate defensive techniques that have been implemented. It is not economic to pen test every build. As weaknesses will be remedied subsequent to pen testing, any software protection techniques need to be automated and repeatable to ensure incremental and repeatable levels of achieved security.

6 CONCLUSION – PULLING IT ALL TOGETHER

Trustworthiness in software depends on software quality and protection. Like all system characteristics, the specific aspects of trustworthiness in a system depend on the kind of system, the norms of the industry in which the system operates, and the consequences of failures. The trustworthiness requirements can be met with attention to software quality and trustworthiness throughout the entire software lifecycle from architecture, design and development through operations, maintenance and decommissioning. Confidence in the software depends on the quality of the software itself, and confidence in the organization and the processes it uses to produce the software. Thus, the transparency and evidence of good practices is necessary to convince others that the software should be trusted.

Change management, including secure and authenticated software updates, is essential to maintaining trust in software as it changes to enhance the capabilities and address weaknesses, and for changes in the context in which the software operates. Traceability is important to ensure that the software, as designed, tested and modified, meets the intended use cases and requirements and can be audited to ensure this is the case. A variety of techniques may be used to provide software assurance, confidence that the software does what it is supposed to and is also free from vulnerabilities that can impact trustworthiness. Care must be taken to include correct configuration and operation as part of the analysis of software quality and assurance.

Software protection should be planned at the design stage and integrated into the lifecycle process, both to protect intellectual property and reduce the risk of improper use or modification of the software.

Achieving the economic balance between risks, costs and time to market is difficult. A disciplined approach to trustworthiness gives the organization a structured and well-reasoned capability to decide what measures need to be taken and which do not need to be taken based on that system's trustworthiness requirements.

Attention to the quality and protection of software throughout the lifecycle can improve confidence in the software, it's trustworthiness and suitability for deployment.

Annex A SOFTWARE LIFECYCLE CHECK LIST

The table sets out a list of items that are required for large scale projects and is not restricted to the Industrial context. This table elaborates the topics and concepts found in the "Software Development and Assurance Lifecycle Phases" as shown in Figure 3-2.

Phase Name	Software Development and Assurance Lifecycle Phases	Description of Phases
Operational need	Business or mission analysis	Identify threat environment and opportunities for attack
Requirements	Stakeholder needs and requirements definition	Define functional requirements for operation in cyber contested environment
	System requirements definition	Derive non-functional software assurance (SwA) requirements
Design	Architecture definition	Develop secure architecture Obtain data rights
	Design definition	Design system with considerations for SwA
	System analysis	Criticality analysis SwA evaluation of COTS Analyze suppliers Selection of cybersecurity controls
Product	Support development	Risk management with assurance cases Configuration management with version control Access control and code signing Project status with measures and metrics
	Implementation	Warning flags and coding standards Hardening measures Software protection Code reviews
	Integration	Full system regression testing Automated reproducible build
	Verification	Static source code weakness analysis Binary analysis Origin analysis Web app scanners and fuzzers Negative testing Automated test suite with coverage Penetration testing

Phase Name	Software Development and Assurance Lifecycle Phases	Description of Phases
Validate solution	Transition	Transition data rights Ensure acquirer can rebuild and retest
	Validation	Conduct third-party SwA testing Validate security requirements/assumptions
Delivered Capabilities	Update	Monitor for 3rd party vulnerabilities Continued Assessment & Timely Patching
	Unsupported	When software support and updates are no longer available from the developer[1].
	End of Life	When software is removed from operational use by the organization.

Table A-1: Software Lifecycle Check List

[1] Due to the developer no longer supporting the software or the operator choosing to not subscribe to the available support.

Annex B EXAMPLES OF SOFTWARE LIFECYCLE FAILURES

Inappropriate management of software design, development, and deployment impact trustworthiness, in particular safety and reliability. All stages of the software lifecycle require consideration and assurance. In some cases, disaster has been narrowly averted due to good fortune, in others, deaths have resulted. Here are some examples.

B.1 LA FLIGHT CONTROL LOST COMMUNICATION WITH AIRPLANES: SOFTWARE IMPLEMENTATION ERROR WITH TIMER RESET

In 2004, both an air traffic control system and its backup system shut down, due to the same flaw. This meant that the air traffic controls could not guide the traffic as planes "started to head toward one another". The cause was a controller timer that reached the maximum value and then shut down. Typically, the timer would be reset upon a system restart so this problem would not occur, but not this time.[1]

B.2 THERAC-25 DEATHS DUE TO PROJECT MANAGEMENT, SYSTEM DESIGN, IMPLEMENTATION AND TESTING ERRORS

The Therac-25[2] computer-controlled radiation therapy machine was released in 1982, but the organizational and software issues remain relevant today. The Therac-25 caused massive radiation doses in a number of patients in the period of 1985-1987 causing several deaths.[3] There were numerous causes, including a lack of documented software design, inadequate testing, and the difficulty of reproducing the faults due to race conditions. The device was based on an earlier version that had hardware safety interlocks that were removed in the new version, which relied on software instead. Latent flaws were not evident until the software was reused. Therac-25 relied on the correct positioning of a hardware "turntable" to pass strong x-ray beams through a "flattener" to create a uniform field of application on the patient. Other positions allowed a weaker therapy or machine alignment, neither of which required the "flattener". If the turntable were positioned in the incorrect position, then the patient would be subject to a huge and potentially deadly radiation dose. This happened in multiple instances, and not always from the same software flaw. In some cases, a flag indicating data entry was complete could incorrectly signal that treatment could begin, even though updates from data entry had not been correctly recorded. In another case this could happen if treatment were started at the time a byte flag value wrapped. Subsequent to the accidents, FDA investigation, lawsuits and user group feedback, hardware safety features were reintroduced, as well as updates to the software and software development process.

[1] See Lost Radio Contact Leaves Pilots on Their Own [Geppert2004]

[2] See [Therac25]

[3] See Medical Devices: The Therac-25 [Leveson1995]

B.3 TEMPE ARIZONA POWER OUTAGE: SUPPLY CHAIN VENDOR COMMUNICATION & CONFIGURATION MANAGEMENT ERROR

Tempe Arizona experienced a power outage in June 2007 when a vendor engineer changed a SCADA system configuration to automatic from the utility configuration of manual, affecting the Energy Management System load-shedding program operation when a test went awry.[1] This highlights the need for configuration management and vendor communication and controls. No significant harm resulted in this instance, apart from the 45-minute outage affecting the reliability of electric service to 98,700 customers, but it could have had subsequent safety impacts by affecting systems dependent on power.

B.4 NISSAN AIR BAG RECALL: TESTING AND CALIBRATION FAILURE

In 1994 Nissan[2] was forced to recall almost a million cars for a software fix. The air bag system did not detect a person in the seat, which led to the air bag not deploying in an accident and placing the passenger at risk.

The failure was cited as a 'calibration error'. It transpired that typical sequences of events were not tested to exercise the software in enough real-world situations. For example, if the seat was initially empty, the engine vibration masked the fact that a person had subsequently sat in the seat. Unusual seating postures taken by the passenger also led to failures to detect the passenger.

B.5 POINT OF SALE CARD SKIMMING: DATA PRIVACY AND INADEQUATE SOFTWARE PROTECTION PLANNING

Large retail vendors, such as Target and Home Depot,[3] were subject to credit-card skimming attacks in 2014. A common feature of the attacks was the failure to protect privacy of the credit card information as it passes through the point-of-sale device. Various mechanisms to inject malware that scanned for credit card numbers in the memory of the terminal (even after the completion of the transaction). The captured information was held on the terminals for subsequent period transmission to the perpetrators of the hack for sale on the dark web.

Such data protection risks depend on the ability of malicious software to take advantage of architecture and software mechanisms to obtain data inappropriately.[4] Inadequate data partitioning and software protection led to devices that were vulnerable to the injected malware.

[1] See Protecting Industrial Control Systems from Electronic Threats [Weiss2010]
[2] See Nissan recalls nearly 1 million cars for air bag software fix [Charette2014]
[3] See Home Depot to pay $27.25M in latest data breach settlement [Seals2017]
[4] See Make Yourself Less of a Target – A multi-layered Approach to Application Shielding [Yuan2018]

B.6 SOFTWARE PATCHING FAILURES

Software patching raises risks in ongoing software maintenance. For example, an unnamed water utility discovered that they could not operate their system after installing a software patch, since it erased their license keys. The system was down until the keys could be replaced. In another instance a Windows NT system was updated with a new service pack that affected an ethernet card. This allowed a water utility to start pumps at a water sewage plant, but not to stop them. A software update typical in an IT environment had unexpected consequences when an IT system component was used in an industrial control context.

These few examples demonstrate the need to consider software trustworthiness throughout the entire software lifecycle.

Annex C REFERENCES

[Barr] Embedded C Coding Standard, *https://barrgroup.com/Embedded-Systems/Books/Embedded-C-Coding-Standard*

[Bryant2004] Eric D. Bryant, Mikhail J. Atallah, Martin R. Stytz: A Survey of Anti-Tamper Technologies, CERIAS Tech Report 2004-55, Center for Education and Research in Information Assurance and Security, Purdue University, Nov 2004, *https://pdfs.semanticscholar.org/7834/4269b7c378b5a0c955865cf8286896e4bcda.pdf?_ga=2.35193634.2003186129.1565099942-2098620458.1557753915*

[Charette2014] Robert N. Charette: Nissan recalls nearly 1 million cars for air bag software fix, March 2014, *https://spectrum.ieee.org/riskfactor/transportation/safety/nissan-recalls-nearly-1-million-cars-for-airbag-software-fix*

[CWE] Common Weakness Enumeration – a community developed list of software weakness types (citing a web page with an unknown author), *https://cwe.mitre.org*

[CWE-ENUM] Enumeration of Technical Impacts - each weakness, if successfully exploited, can lead to one or more potential Technical Impacts (citing a web page with an unknown author), *https://cwe.mitre.org/cwraf/enum_of_ti.html*

[CWE-701] Weaknesses Introduced During Design (citing a web page with an unknown author), *https://cwe.mitre.org/data/definitions/701.html*

[Geppert2004] Linda Geppert: Lost Radio Contact Leaves Pilots on Their Own, IEEE Spectrum 2004, *https://spectrum.ieee.org/aerospace/aviation/lost-radio-contact-leaves-pilots-on-their-own*

[Ghidra] Ghidra resources as posted online by the NSA (citing a web page with an unknown author), *https://www.nsa.gov/resources/everyone/ghidra/*

[Goodes2018] Grant Goodes: More than meets the Eye… Software Transformation vs Obfuscation, *https://cloakable.irdeto.com/2018/05/07/more-than-meets-the-eye-software-transformation-vs-obfuscation/*

[Holzman2006] Gerard J. Holzman: The Power of 10: Rules for Developing Safety-Critical Code, NASA/JPL Laboratory for Reliable Software, June 2006, *http://web.eecs.umich.edu/~imarkov/10rules.pdf*

[Huaw-Trans] Huawei Cyber Security Transparency Centre (citing a web page with an unknown author), *https://www.huawei.com/en/about-huawei/trust-center/transparency/huawei-cyber-security-transparency-centre-brochure*

[IIC-DPBP2019] IIC: Data Protection Best Practices Whitepaper, IIC Publication May 2019, *https://www.iiconsortium.org/pdf/Data_Protection_Best_Practices_Whitepaper_2019-07-22.pdf*

[IIC-IICF2018] The Industrial Internet of Things Volume G5: Connectivity Framework, version 1.01, 2018-February-28, *https://www.iiconsortium.org/pdf/IIC_PUB_G5_V1.01_PB_20180228.pdf*

[IIC-IISF2016] Industrial Internet of Things Volume G4: Security Framework, version 1.0, 2016-September-26, *https://www.iiconsortium.org/IISF.htm*

[IIC-IIV2015] IIC: The Industrial Internet, Volume G8: Vocabulary Technical Report, version 1.0, 2015-May-07, *http://www.iiconsortium.org/vocab/index.htm*

[IIC-SMMD2019] IoT Security Maturity Model: Description and Intended Use, Version 1.1, 2019-February-15 *https://www.iiconsortium.org/pdf/SMM_Description_and_Intended_Use_FINAL_Updated_V1.1.pdf*

[IIC-SMM2019] IoT Security Maturity Model: Practitioner's Guide, Version 1.0, 2019-February-25 *https://www.iiconsortium.org/pdf/IoT_SMM_Practitioner_Guide_2019-02-25.pdf*

[IS0/IEC 12207] ISO/IEC standard 12207: Systems and software engineering — Software life cycle processes, 2008 *https://www.iso.org/standard/63711.html* Introduction and overview: *https://en.wikipedia.org/wiki/ISO/IEC_12207*

[IS0/IEC 15288] ISO/IEC standard 15288: Systems and software engineering — System life cycle processes, 2015
https://www.iso.org/standard/63711.html
Introduction and overview: *https://en.wikipedia.org/wiki/ISO/IEC 15288*

[ISO-TS17961] ISO/IEC TS 17961, Information Technology — Programming languages, their environments and system software interfaces — C Secure Coding Rules, 26th June 2012. *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1624.pdf*

[Kasp-Trans] Kaspersky Global Transparency Initiative and Transparency Centers (citing a web page with an unknown author)
https://www.kaspersky.com/about/transparency

[Leveson1995] Nancy Leveson: Medical Devices: The Therac-25, from Safeware: System Safety and Computers, Addison-Wesley, 1995.
http://sunnyday.mit.edu/papers/therac.pdf

[Melt-Spect] Meltdown and Spectre - vulnerabilities in modern computers leak passwords and sensitive data (citing a web page with an unknown author)
https://meltdownattack.com

[Msft-Trans] Microsoft Transparency Centers (citing a web page with an unknown author)
https://docs.microsoft.com/en-us/security/gsp/contenttransparencycenters

[MISRA] MISRA-C:2012, Guidelines for the use of the C language in critical systems,
https://www.misra.org.uk/Activities/MISRAC/tabid/160/Default.aspx

[Seals2017] Tara Seals: Home Depot to pay $27.25M in latest data breach settlement, Info Security magazine, March 2017,
https://www.infosecurity-magazine.com/news/home-depot-to-pay-2725m/

[TCG-DICE] Trusted Computing Group, DICE Architectures (citing a web page with an unknown author),
https://trustedcomputinggroup.org/work-groups/dice-architectures/

[Therac25] Therac-25: A description of the problems with a computer controlled radiation machine starting in 1982 (citing a web page with an unknown author),
https://en.wikipedia.org/wiki/Therac-25

[Thomp984] Ken Thompson: Reflections on Trusting Trust, Communication of the ACM, Vol. 27, No. 8, August 1984,
https://www.win.tue.nl/~aeb/linux/hh/thompson/trust.html

[Weiss2010] Joseph Weiss: Protecting Industrial Control Systems from Electronic Threats, Momentum Press, 2010

[WIKI-CG] Wikipedia: Call Graph (citing a web page with an unknown author), *https://en.wikipedia.org/wiki/Call_graph*

[WikiM01] Wikimedia Commons, Volvo FH12-Inger (F)-2003.jpg, *https://commons.wikimedia.org/w/index.php?curid=649934*

[WikiM02] Wikimedia, Klinge Corporation Redundant Refrigeration System, *https://upload.wikimedia.org/wikipedia/commons/6/60/Redundantreefer.JPG*

[Yuan2018] Brian Yuan: Make Yourself Less of a Target – A multi-layered Approach to Application Shielding, 9th January 2018, *https://cloakable.irdeto.com/2018/01/09/make-yourself-less-of-a-target-a-multi-layered-approach-to-application-shielding/*

Annex D REVISION HISTORY

Revisions	Date	Editors	Changes Made
1.0	2020-03-23	Marcellus Buchheit, Mark Hermeling, Frederick Hirsch, Bob Martin, Simon Rix	Initial Release

Table D-2: Revision History

AUTHORS AND LEGAL NOTICE

This document is a work product of the Industrial Internet Consortium Trustworthiness Task Group, co-chaired by Frederick Hirsch (Fujitsu), Marcellus Buchheit (Wibu-Systems) and Bob Martin (MITRE), which is a subgroup of the Security Working Group co-chaired by Sven Schrecker (Trust Driven Solutions), Jesus Molina (Waterfall Security Solutions) and John Zao (NTCU PET Lab).

AUTHORS

The following persons contributed substantial written content to this document:

Marcellus Buchheit (Wibu-Systems), Mark Hermeling (GrammaTech), Frederick Hirsch (Fujitsu), Bob Martin (MITRE), Simon Rix (Irdeto).

EDITORS

Marcellus Buchheit (Wibu-Systems), Frederick Hirsch (Fujitsu), Simon Rix (Irdeto).

CONTRIBUTORS

The following persons contributed valuable ideas and feedback that significantly improved the content and quality of this document:

Anastasiya Kazakova (Kaspersky), Igor Kumagin (Kaspersky), Tetsushi Matsuda (Mitsubishi Electric), David Noller (IBM), Paul Peters (IBM), Bassam Zarkout (IGnPower).

TECHNICAL EDITOR

Stephen Mellor (IIC staff) oversaw the process of organizing the contributions of the above Editors, Authors and Contributors into an integrated document.

IIC ISSUE REPORTING

All IIC documents are subject to continuous review and improvement. As part of this process, we encourage readers to report any ambiguities, inconsistencies or inaccuracies they may find in this Document or other IIC materials by sending an email to *admin@iiconsortium.org*.