KDAB

# Cybersecurity and HMI

*What should you do?*

## Background

Stories of cybersecurity issues appear in the news almost constantly. Sophisticated and daring cyber-attacks have grabbed the headlines and the attention of the public. Both Miller and Valasek's famous attack on a Chrysler Jeep as referenced in their 2015 paper "Remote Exploitation of an Unaltered Passenger Vehicle" [1] and a separate, sophisticated attack on the Ukraine power grid [2] that same year involved highly motivated, organized, qualified and professional individuals with considerable resources at their disposal. And as recently as January 2023, T-Mobile discovered that a malicious actor had gained access to their systems and stolen personal information, like names, emails, and birthdays, from over 37 million customers [3].

Images of subterfuge, highly qualified individuals in laboratories, and suspected international espionage are more reminiscent of a James Bond movie than the average HMI interface to a specialist niche of limited general interest. And surely, security challenges associated with IT systems like the T-Mobile example are interesting, but irrelevant?

Because the HMI isn't under attack, is it? The back-end stuff is where the problems are, and that's someone else's problem, isn't it? It doesn't really matter to you what happens downstream. Right?

Well, yes. And no.

## Using this document

This document assumes no prior knowledge of cybersecurity. If you are looking to learn how cybersecurity impacts HMI design and implementation and have the time to read it all – that's great!

If not – stick to the black section, p. 3-7, for an overview, or dip into the detailed color-coded sections as you please. Each offers a complete and rounded introduction to its topic.

## Cybersecurity buzzwords and phrases

Like most other specialist fields, the world of cybersecurity didn't take long to adopt a whole set of terminology that can be an obstacle for the newcomer.

Useful terms to know in the context of this document include:

**Attacker:** a person or process attempting unauthorised access to restricted areas of a system in order to perform a malicious act.

**Attack surface:** the sum total of vulnerabilities in a system or subsystem, or the end goal of one or more attack vectors. For example, network drivers, user applications and file systems will each have an attack surface.

**Attack vector:** an access route taken by an attacker to compromise a system. Embedded systems attack vectors include external disk drives, LANs, the Internet, and Wi-Fi.

**Bad actor (aka threat actor):** an individual performing a malicious act.

**Edge computing:** a computing paradigm which refers to networks and devices at or near the user.

**Endpoint:** A physical device that connects to and exchanges information with a computer network.

**Hacker:** an individual who uses technical skills to exploit cybersecurity defences.

- **Black-hat hacker:** criminal who breaks into computer systems or networks with malicious intent – perhaps for financial gain, or just to create mischief.
- **White-hat hacker:** ethical cybersecurity technician who openly exploits computer systems or networks to isolate their security flaws and to identify potential improvement.
- **Grey-hat hacker:** cybersecurity expert who may well look for vulnerabilities in a system without the owner's permission or knowledge, but who will then report them to the owner. As the name suggests, from an ethical perspective the activities of a grey-hat hacker sit somewhere between white- and black-hat hackers.

**Vulnerability:** a weakness that a bad actor can exploit to perform unauthorized actions within a system.

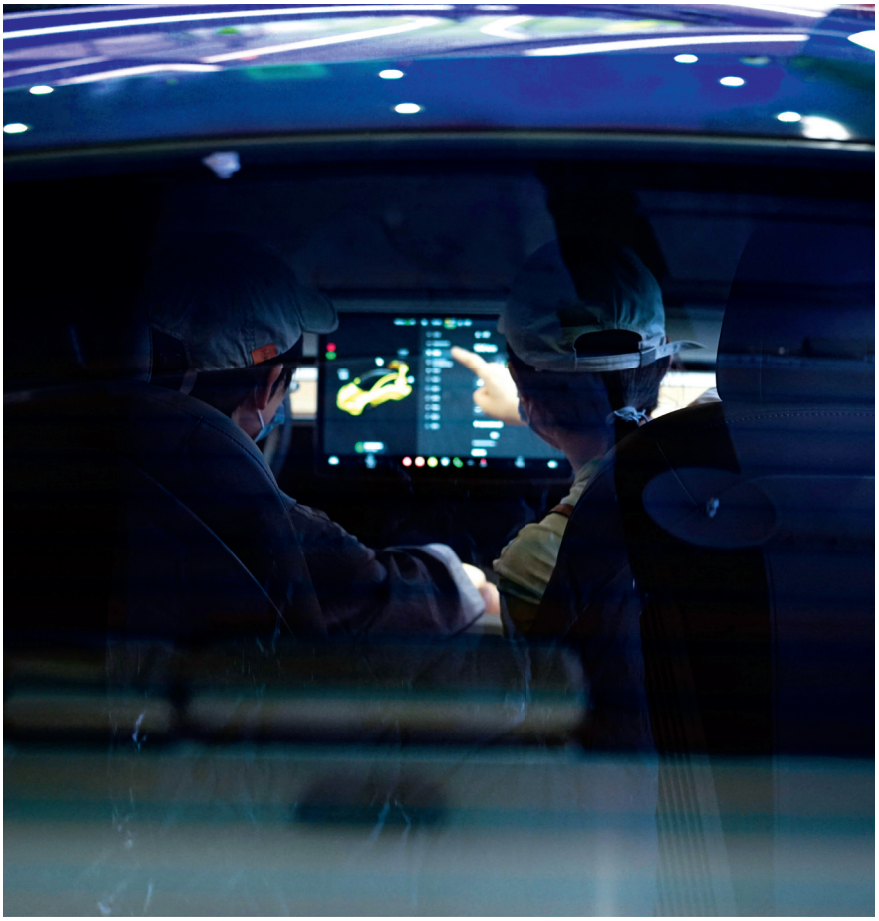


*Image 1: Functionality is being integrated into the touchscreen HMI*
***Photo by: ün LIU***

## Cybersecurity and the HMI

Cybersecurity can be important for all sorts of reasons. It is important to protect a patient's medical records in a medical device, for example. It is also important to protect account details in a banking application, and PIN numbers in an ATM. In safety-critical devices – cars, trains, aeroplanes, industrial plant - there can be safety implications too.

Automotive systems present an obvious example of where capacitive HMI systems are becoming an integral part of critical functionality such that a malfunction could have implications for driver, passengers, or other road users. Most cars released in recent years have seen the functionality traditionally associated with many physical buttons integrated into the touchscreen HMI, lending a minimalistic and clean look to the interior as a whole.

The evolution of car control systems from a collection of disparate, stand-alone control units to an integrated, connected whole has brought many benefits. But white-hat hackers Miller and Valasek provided a pertinent example of why it needs to be approached with care. In their paper "*Remote Exploitation of*

*“Recent years have seen a shift to a ‘cloud’ connected world, expected to be accessible to facilitate monitoring, upgrading, enhancement, and supplementation”*

*an Unaltered Passenger Vehicle*”, they observed of a Jeep vehicle that “*…there are no CAN bus architectural restrictions, such as the steering being on a physically separate bus. If we can send messages from the head unit, we should be able to send them to every ECU on the CAN bus.*”

The implication here is that it is important to restrict access to safety critical domains from those that are more benign. The paper references a “*physically separate bus*”, which in theory would protect the safety critical element from outside interference. However, it is not possible to both isolate the domain completely and still leverage the full potential of the connected car because some communication between domains is always going to be necessary.

For example, a touchscreen HMI may be innocuous in isolation, but if it allows the driver to manipulate settings related to functions such as lane change assist, intelligent cruise control, and ABS, then clearly there is a need to communicate changes to those settings.

In practice there are several ways to provide some degree of separation between different domains and yet still provide a level of communication between them. These include hardware features such as ARM TrustZone [4], and software solutions including hypervisors and RTOS.

## Cybersecurity and “shifting left”

Most embedded applications have traditionally been isolated, static, fixed-function, device-specific implementations. Established practices and processes have relied on that status. But recent years have seen a shift to a “cloud” connected world, expected to be accessible to facilitate monitoring, upgrading, enhancement, and supplementation. That has cybersecurity implications for the whole system – including the HMI.

Traditionally, the practice for secure code verification has been largely reactive. Code is developed by following somewhat loose guidelines and then subjected to performance, penetration, load, and functional testing to find vulnerabilities, which are fixed later.

A better, more proactive approach ensures code is secure by design—a “shift left” along the timeline. 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.

The resulting Secure Software Development Life Cycle (SSDLC) involves integrating security testing and other activities into an existing development process – for example, by writing security requirements alongside functional requirements or by performing an architecture risk analysis as part of an established design phase.

This proactive approach ensures that vulnerabilities are designed out of the system or addressed in a timely and thorough manner without disrupting existing development practice. For example, Figure 1 shows a V-model SSDLC as described in the automotive cybersecurity standard, ISO/SAE 21434. This model

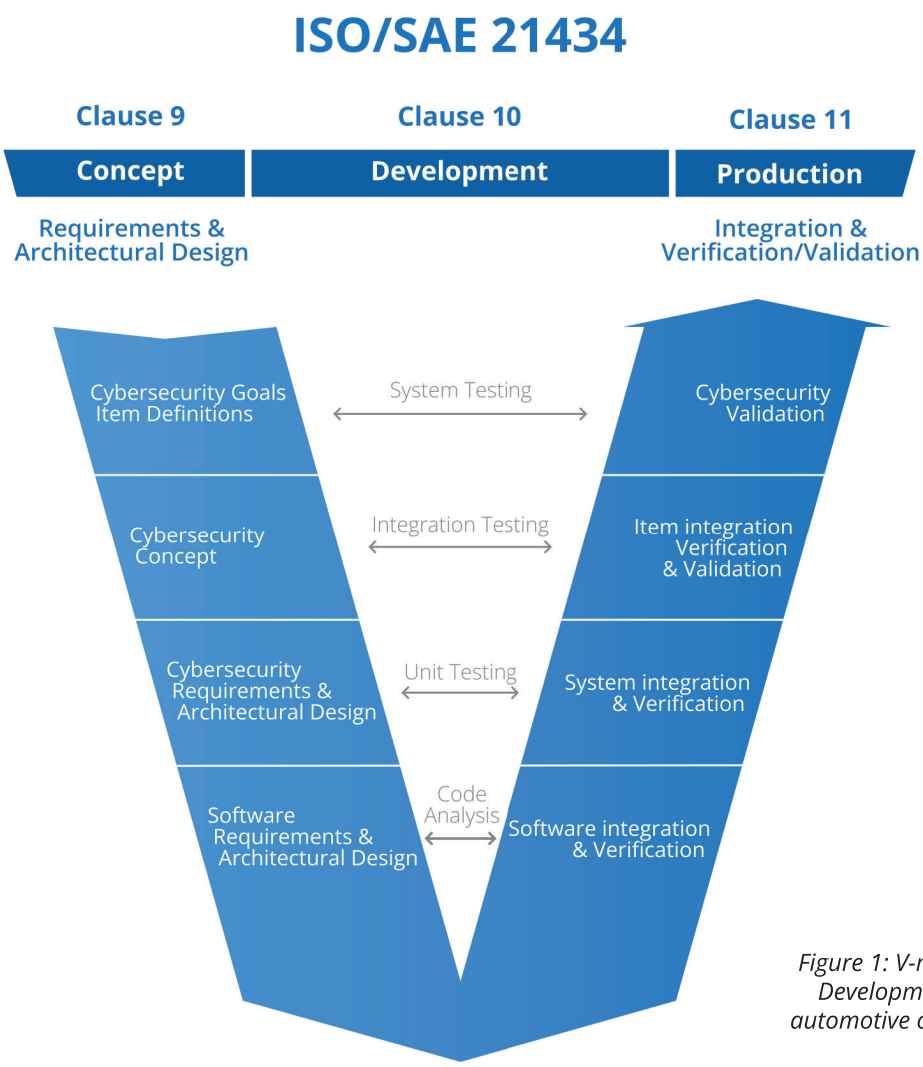



*Figure 1: V-model Secure Software Development Life Cycle from the automotive cybersecurity standard ISO/SAE 21434*

can be applied concurrently with its functional safety partner document, ISO 26262.

The same principles can be applied to the DevOps lifecycle, resulting in what has become known as DevSecOps. Although the context differs between DevSecOps and the SSDLC, shift left therefore implies the same thing for both—that is, an early and ongoing consideration of security.

In the DevSecOps model, the DevOps life cycle is superimposed with security-related activities throughout the continuous development process (Figure 2).

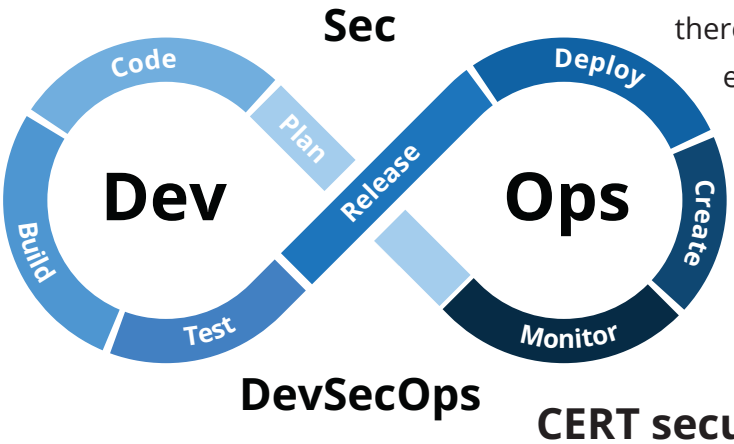



*Figure 2: In the DevSecOps model, the DevOps life cycle is superimposed with security-related activities throughout the continuous development process*

## CERT secure coding practices

There are many sources of good advice on secure software development, but perhaps the most definitive comes from the CERT (Computer Emergency Readiness Team) Division of the Software Engineering Institute (SEI) [5]. Created in 1988 as the CERT Coordination Centre in response to the Morris worm incident, the CERT division now has more than 150 cyber-security professionals working on projects that take a proactive approach to securing systems.

The CERT division is a trusted, authoritative organisation dedicated to improving the security and resilience of computer systems and networks. It partners with government, industry, law enforcement, and academia to develop advanced methods and technologies to counter large-scale, sophisticated cyber threats.

CERT have nominated a total of 12 key secure coding practices – a “top ten”, more recently supplemented by two “bonus practices”. Of course, like the “shift left” principle, these recommendations are for software development in general and are not specific to HMI development. Here we will interpret those practices as they relate to the development of secure HMI related code.

## Grouping the CERT practices

As the previous two diagrams illustrate, whatever the preferred development lifecycle there is always a need to specify requirements and architect the system, design it, and implement that

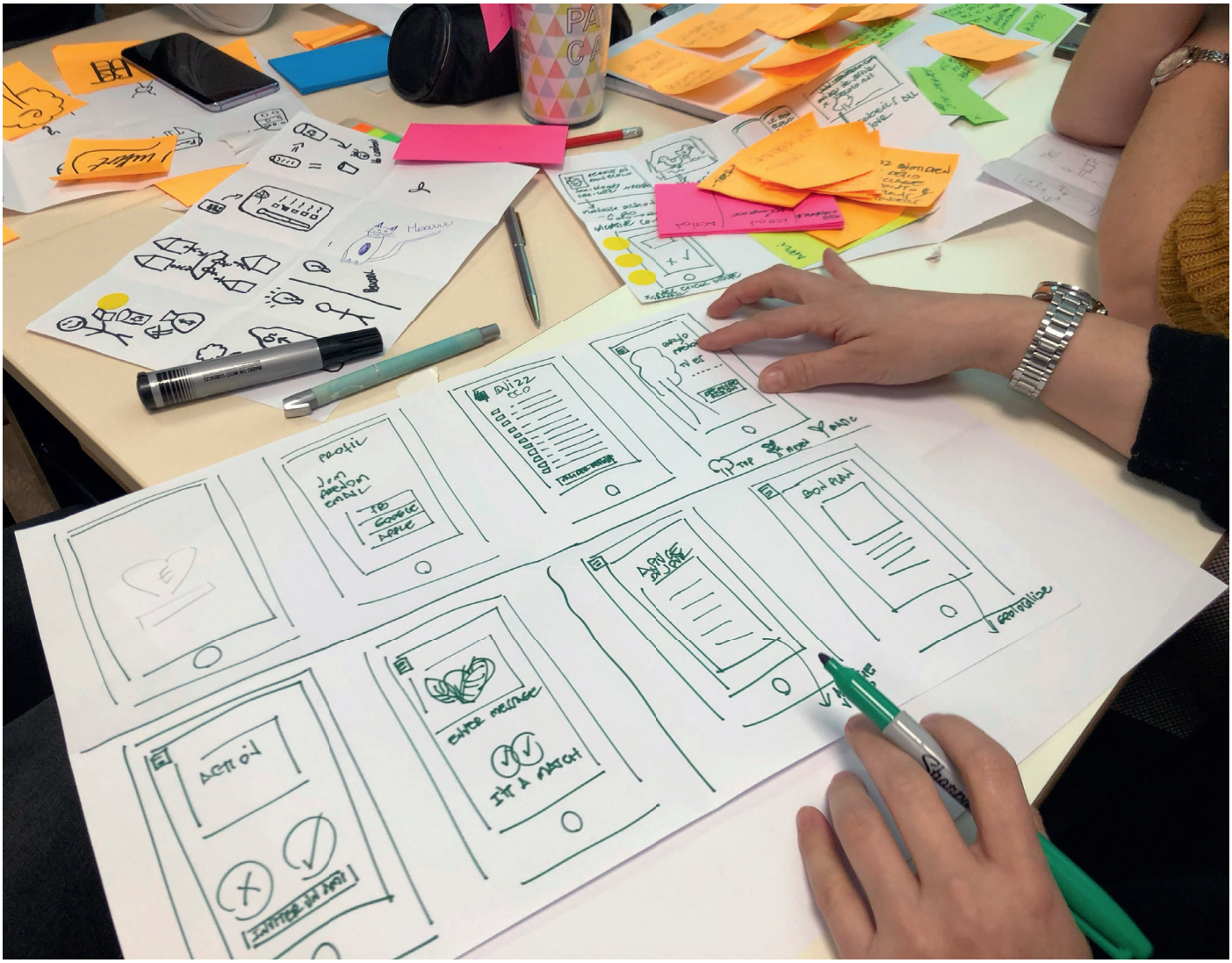


*Image 2: Shifting left implies designing in security from the very beginning* *Photo by: Amélie Mourichon*

design. The “shift left” principle underlines the need to check traceability at each step of that path to ensure correct implementation, and to test as early as possible.

Arguably, that traceability implies that all of the CERT practices are relevant throughout the development lifecycle. Grouping them in this way is necessarily subjective. However, the aim is to detail each of the twelve best practices at the stage where it first makes an impact on the lifecycle of HMI development.

## Requirements and architecture

A best-practice secure software system architecture will:

- Implement defined security requirements and policies which will include the adherence to the principle of least privilege as part of a defence-in-depth strategy, where:
  - The principle of least privilege is that every process within the system should execute with the least set of

privileges necessary to complete its job.

- A defence-in-depth strategy involves the adoption of multiple defensive strategies, so that if one layer of defence turns out to be inadequate, another layer of defence can minimize the extent to which a security flaw becomes an exploitable vulnerability.

If you would like to learn more about best-practice requirements definition and architecture of a secure HMI application, read the blue section, p. 10-15.

## System design

A best-practice secure software system design will:

- Demonstrate traceability to show that requirements and architecture have all been implemented – and nothing more.
- Be simple. Unnecessary complexity implies software that is difficult to understand, difficult to maintain, difficult to test, and prone to error – and hence liable to contain vulnerabilities.
- Nominate or define a secure coding standard. The right coding standard will limit the number of vulnerabilities that are inadvertently included in the code base.
- Include for the validation and sanitization of data interfaces to other systems. Outgoing and incoming data needs to be "sanity checked" to ensure that it is always within expected bounds, both in terms of values and in terms of use.
- Base access decisions on permission rather than exclusion ("default deny"). Such an approach is more challenging to implement but implicitly more secure.

If you would like to learn more about best-practice design of a secure HMI application, read the green section, p. 16-20.

## System development

Secure software system development best-practices include

- Demonstrate traceability to show that the requirements, design, and architecture have all been implemented – and nothing more.
- Apply effective quality assurance techniques. Select the right approaches to demonstrate that security has been "designed in", as well as demonstrating that the completed system is secure.

- Understand the potential security threats, and model them to demonstrate that they are accounted for.
- Heed and attend to compiler warnings, which imply that the code is not ideal. Code that is demonstrably less than perfect at the outset will certainly present a larger attack surface.

If you would like to learn more about best-practice development of a secure HMI application, read the yellow section, p. 21-23.

## Conclusions

There will almost certainly be more critical aspects of any connected system than the HMI. However, that does not make cybersecurity less important for the HMI developer. The HMI is the interface to the outside world – an “endpoint” – and it therefore has a key role to play in ensuring that more critical domains are protected from bad actors.

Cybersecurity is a very broad subject indeed, and secure coding is one small part of that. Within that context, CERT’s recommended practices are also quite broad, language-independent recommendations, some of which require interpretation in the programming language of choice before they can be implemented. Coding standards such as CERT C++ provide the most obvious example of that.

The core principle underlying all of these practices is a recognition that the cost to remove defects, including security flaws, can be hundreds of times higher after deployment. Solutions for identifying and preventing security flaws during development are much more cost effective than in the test phase or post-deployment.

Perhaps one issue they fail to highlight, however, is that arguably the development lifecycle for a secure application can never be closed. Any newly exposed vulnerability implies a new requirement to address it, even on software that has been deployed for years. For that reason, the ongoing maintenance of the documentation and support infrastructure implied by many of these suggested practices is an important consideration.

# Secure HMI requirements and architecture

## Define security requirements

*"Identify and document security requirements early in the development life cycle and make sure that subsequent development artefacts are evaluated for compliance with those requirements. When security requirements are not defined, the security of the resulting system cannot be effectively evaluated."*

*- CERT Bonus Secure Coding Practices*

In order to architect and design for security policies, it is first necessary to establish what those policies are – and, by implication, the requirement to adhere to those policies. Security requirements, like safety and functional requirements, need to be specified at the outset and need to be shown to have been implemented.

## Architect and design for security policies

*"Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set."*

*- CERT Top Ten Practices*

Security should not be regarded as a "bolt-on" feature. For an HMI, that means establishing a set of security requirements at the outset and ensuring that they are fully implemented at the architecture and design stage. Ensure that "Least Privilege" principles are followed within the structure of the HMI and the system as a whole, dividing each into subsystems such that each is only allowed the privileges necessary to perform the task it is designed for.

It also applies within the HMI itself. The division of business logic and user interface logic is considered good practice from the perspective of language selection (perhaps C++ and QML, respectively) [6] and the implied separation between those domains reinforces that best advice.

## Adhere to the principle of least privilege (POLP)

*"Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges"*

*- CERT Top Ten Practices*

The desirability of "Least Privilege" principles are emphasised more explicitly in this CERT recommendation. It is one of their recommended security policies. If processes are to be separated in this way, then the systems they belong to must be modularised sufficiently for that separation to be effective and communications between the resulting modularised domains must be tightly controlled.

These principles have long been promoted in academic circles. As long ago as the early 1970s, Saltzer and Schroeder [7] established a set of principles based on the idea of modularization, noting that *"Every program and every user of the ... system should operate using the least set of privileges necessary to complete the job"* – the Principle Of Least Privilege, or POLP.

*"As long ago as the early 1970s, Saltzer and Schroeder established a set of principles based on the idea of modularization"*

A few years later, John Rushby [8] developed a similar line of thinking in the shape of the MILS (Multiple Independent Levels of Security/Safety) initiative. MILS is a high-assurance security architecture, which by design provides:

- separation and controlled information flow
- separation mechanisms that support both untrusted and trustworthy components
- security mechanisms that are non-bypassable, evaluable, always invoked, and tamperproof.

A MILS compliant connected system will therefore consists of high-assurance components and applications which can be independently developed, modularly combined, evaluated, and certified to adhere to these principles [9].

Of course, developers responsible for the HMI will not always have control or even input into the architecture of the system as a whole. But adherence to these principles in so far as they apply to the relationship between the UI and business

logic, and between the HMI and the remainder of the system, represents best practice.

## Communication paths between segregated software items

There is usually a need for communication between software components that make up a system, including those of different criticality. The interface between business logic and UI logic is a good example of that. Communications paths from less critical components can represent attack vectors with the potential to compromise highly critical code.

Clearly the elements of the system considered most at risk by this, or any other measure will depend on the system itself. However, as a general rule, if domain separation is to deliver its promise of safe secure software, then it is essential that code handling any communication between domains must be scrutinised to at least the level of the most critical software component involved – and arguably to a higher standard than the remainder of the code base.

This implies a responsibility to write HMI code in accordance with best practice, particularly if the HMI is associated with a connected, critical system.

## Practice defence in depth

*"Manage risk with multiple defensive strategies, so that if one layer of defence turns out to be inadequate, another layer of defence can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit.*

*For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment".*

*- CERT Top Ten Practices*

***"Defence in depth is another of CERTs recommended policies and its principles should be implicit in the requirements and architecture of the system"***

Defence in depth is another of CERTs recommended policies and its principles should be implicit in the requirements and architecture of the system.

Perhaps the most challenging aspect of developing software with cybersecurity in mind is that the job can never be consid-

*"A securely coded HMI in isolation is unlikely to be a complete defence from cyberattack"*

ered complete. A securely coded HMI in isolation is unlikely to be a complete defence from cyberattack. But it can make a sound contribution to the security of a system as a whole when considered alongside other physical and software security measures.

These might include some or all of the following:

- Tamper-resistant memory. [10]
- Immutable memory technology. [11]
- Protected key stores. [12]
- Trusted execution environments (e.g., ARM TrustZone)
- Secure boot to make sure that the "correct" image is loaded.
- Domain separation to defend critical parts of the system.
- MILS (Least Privilege) design principles to minimize vulnerability.
- Minimization of attack surfaces.
- Secure coding techniques.
- Security focused testing.

Because no single defence of a connected system can guarantee impenetrability, the next best thing is to apply multiple levels of security so that if one level fails, others stand guard. One popular analogy for this defence in depth approach is that of a medieval castle [13] equipped to defend its inhabitants from siege through the use of towers, curtain walls, moats, mounds, gates, drawbridges, and other defensive mechanisms [14].

A multiple level approach to cybersecurity also makes a great deal of sense, so that if aggressors get past the first line of defence, then there are others in waiting.

However, not all systems are similarly critical. So, should every possible precaution be implemented on every occasion? And if not, what should drive the decision as to what applies, and when? To address that question, it is useful to focus on the relationship between two key defensive strategies – Domain Separation, and the HMI.

## Identifying high risk areas

According to Peterson, Hope and Lavenhar [16], *"Architectural risk assessment is a risk management process that identifies flaws in a software architecture and determines risks to business information assets that result from those flaws. Through the process of architectural risk assessment, flaws are found that expose information assets to risk, risks are prioritized based on their impact to the business, mitigations for those risks are developed and implemented, and the software is reassessed to determine the efficacy of the mitigations."*

Although enterprise computing is the primary focus of this and similar studies, the underlying premise of identifying and focusing attention on the components of the system at most risk makes sense, irrespective of the context.

Cyber risk (also known as cyber threat or security threat) is calculated by considering identified security threats, its degree of vulnerability, and the likelihood of exploitation such that:

*Cyber risk = Threat x vulnerability x information value*

Clearly the elements of the system considered most at risk by this, or any other measure will depend on the system itself. Examples of high-risk areas are likely to include:

- Files from outside of the network.
- Backwards compatible interfaces with other systems – old protocols, sometimes old code and libraries, hard to maintain and test multiple versions.
- Custom APIs – protocols etc – likely to involve errors in design and implementation.
- Security code - anything to do with cryptography, authentication, authorization (access control) and session management.

Consider that principle in relation to a system deploying domain separation technology – in this case, a separation kernel hypervisor (Figure 3).

It is easy to find examples of high-risk areas specific to this scenario. For instance, consider the gateway virtual machine. How secure are its encryption algorithms? How well does it validate incoming data from the cloud? How well does it validate outgoing data to the different domains?

In this context, the HMI is an example of a data endpoint. Is it feasible to inject rogue data? How is the application code configured to ensure that doesn't happen?

Another potential vulnerability arises because many systems need to communicate across domains. For example, suppose that the HMI includes an interface to initiate central locking. It belongs to a fairly benign domain, but in an emergency situation after an accident, unlocking the doors becomes an imperative, implying communication with a more critical domain. Any such communications demand that their implementation is secure.

With these high-risk software components identified, attention can be focused on those parts of the HMI code associated with them. The net result is a system where secure code does not just provide an additional line of defence, but it actively contributes to the effectiveness of the underlying architecture by "reinforcing" its weak points.

Optimizing the security of this application code involves the combined contributions of a number of factors, mirroring the multi-faceted approach to the security of the system as a whole.

As for castles, it is not merely the number of lines of defence that is it important. It is equally important to consider the extent to which each defence covers for the weaknesses in others.

*Figure 3: Deploying separation technology to help optimize security.*

# Secure HMI System design

All 12 of the CERT “best practices” and “bonus practices” have an impact throughout the development lifecycle. However, some of those practices are set in place during the design phase, and they are discussed in this section.

Traceability is a consistent thread throughout all phases. In the design phase, it is desirable to demonstrate that the design implements the requirements and architecture – and nothing more.

## Keep it simple

*“Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex”*

*- CERT Top Ten Practices*

*“Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made [...]”*

Some developers will spend time writing efficient code by limiting the number of statements. Although there may be situations where constraints make the application of optimization methods unavoidable, in general the efficiency of modern compilers limits the benefits of doing so. It is usually far more appropriate to make sure that the code is logically structured, easy to understand, and easily tested.

In everyday language, the words “complex” and “complicated” are largely interchangeable – so, if a system is complicated then it is functionally complex. But in programming circles, software complexity refers to something more specific; it is “*the extent to which a system is difficult to comprehend, modify and test, not to the complexity of the task which the system is meant to perform. Two systems equivalent in functionality can therefore differ greatly in their software complexity.*” [17]

In consequence, if an application is inherently complicated then it is especially important to ensure that the code is no more complex than is required to perform the task at hand. By definition, the more complex the code, the more difficult it is to understand, test and maintain, and the more likely it is that problems will arise.

## *Quantify "low complexity" using metrics like cyclomatic complexity and knots.*

For a "low complexity" requirement to be meaningful there has to be a mechanism to quantify it. There are multiple metrics to help do so. Some of the more well used metrics include:

- Cyclomatic Complexity
- Knots
- Essential Cyclomatic Complexity
- Essential Knots
- Controlled Size
- Single Entry/Exit Points for Procedures/Functions
- Code Comments Ratio with Executable line
- Unreachable Code
- Data and Control Coupling

For example, cyclomatic complexity is of the more commonly used and accessible metrics. Any meaningful source code will have multiple linear independent paths. Decision constructs such as if-else, while, and switch-case create branches and hence more of these paths. The higher the number of paths, the higher the cyclomatic complexity.

Clearly, some code is naturally more complex because it is performing a complicated task. Best practice is to use metrics like these as comparators to guard against any code being more complex than it needs to be.

## Adopt a secure coding standard

*"Develop and/or apply a secure coding standard for your target development language and platform"*

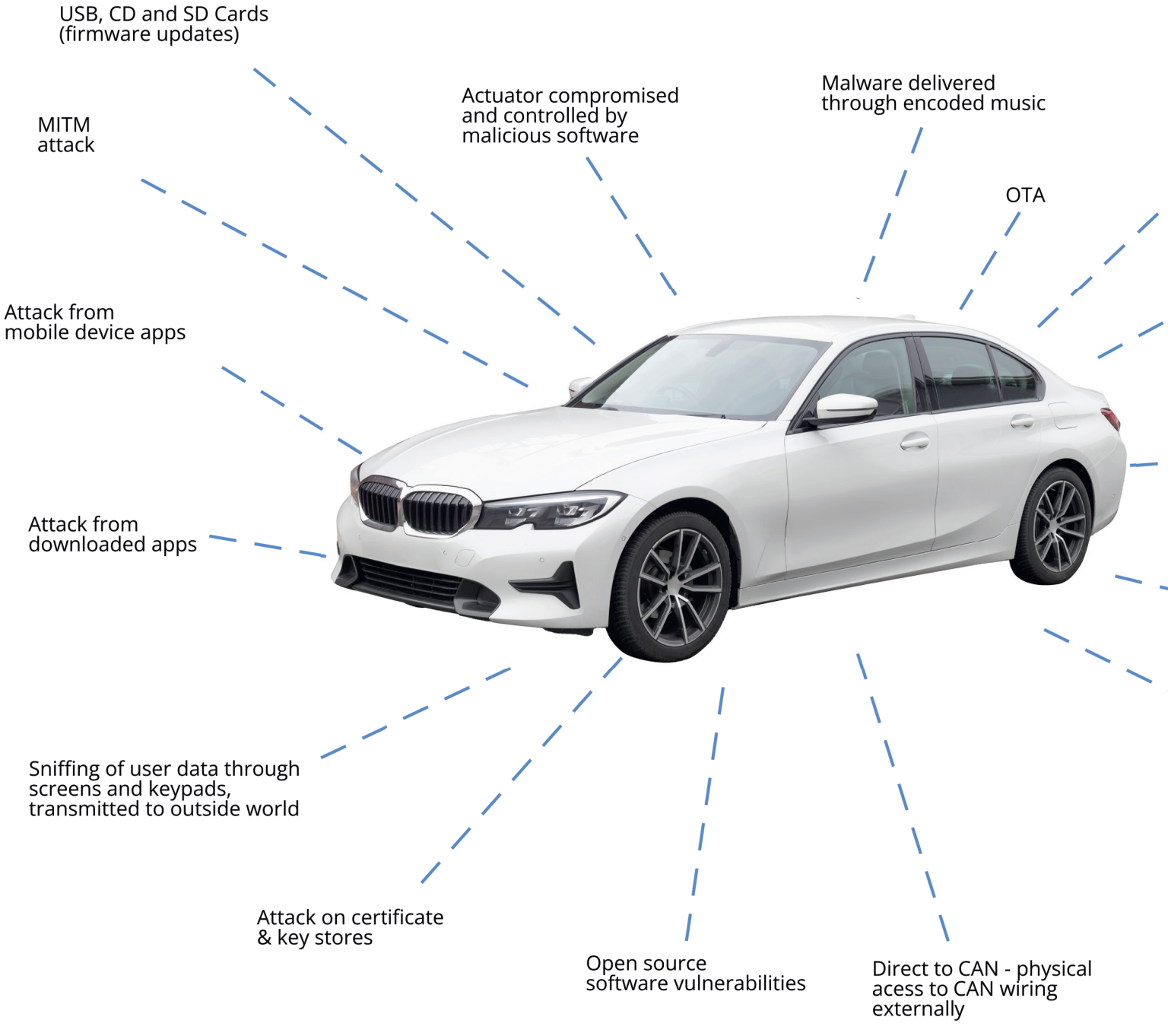
*- CERT Top Ten Practices*

In any high-level language, there are hundreds of instructions and constructs. Some of them are very easy to get wrong (especially in C and C++) and their incorrect use can lead to problems. Coding standards (also known as language subsets) were introduced to disallow the use of those error-prone functions and hence make the resulting code more likely to be error free. In short, they help developers write better, more reliable code. There are a number of potential sources of secure coding stan-

dards, and every opportunity to tune them to the needs of a particular development organization should be taken.

Given that the use of a secure coding standard is one of CERT's key guidelines, it is perhaps no surprise that the same organization has its own. CERT C++ is a coding standard designed for the development of safe, reliable and secure systems that adopts an "application centric" approach to the detection of issues.

MISRA C++ offers another option, despite a common misconception that its designers intended it only for safety-related, not security-related, projects.



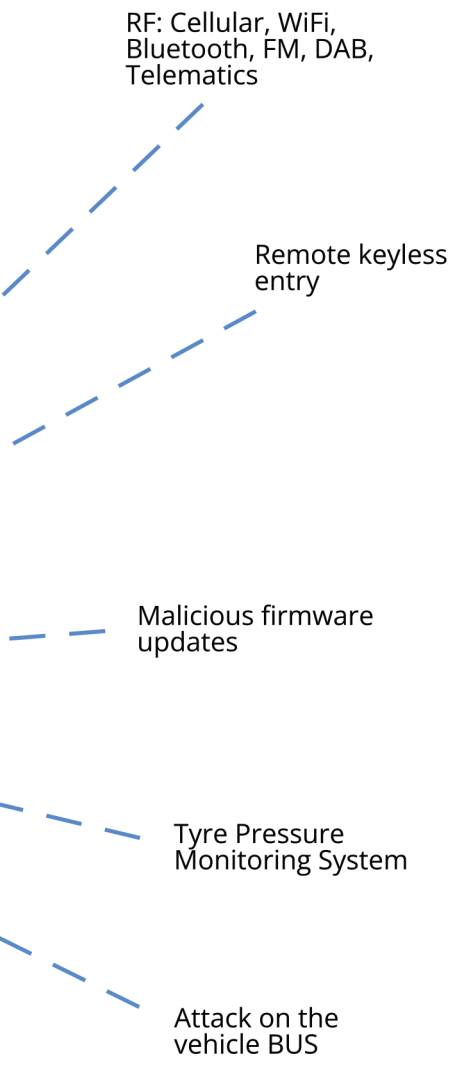
Qt offers its own coding conventions for QML. Although more a style guide than a coding standard, a consistency of style across a project and across a development team will inevitably lead to enhanced maintainability, particularly between developers. That will, in turn, lead to fewer mistakes and misunderstandings and hence make the inadvertent addition of vulnerabilities less likely.

## Validate input

*"Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files"*

*- CERT Top Ten Practices*

*Figure 4: Automotive Attack Surfaces and Untrusted Data Sources*




Exposure to the malicious intent of bad actors is an unfortunate consequence of connectivity across the sectors. Medical devices, industrial plant, power supplies and a host of other applications are potentially vulnerable.

Staying with automotive applications as a relatable example, Figure 4 suggests that there are many potential sources of untrusted data in today's connected car. This "validate inputs" practice clearly applies to each of those data sources, but perhaps less obviously it is also important to consider data from other domains as untrusted – and that includes input data into the HMI from elsewhere in the vehicle.

## Sanitize data sent to other systems

*"Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks.*

*This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem"*

*- CERT Top Ten Practices*

*Implement robust data sanitization procedures to ensure secure communication with complex subsystems, mitigating the risk of injection attacks and safeguarding system integrity.*

Suppose that an HMI needs to communicate data to a highly critical security domain – perhaps the ABS system in our example car. Despite the disparity in criticality, the HMI software may well have a much better understanding of what is acceptable data in the context of the transmission. It is therefore important to deploy secure "sanity check" code to assess its validity in the less critical domain.

## Default deny

*"Base access decisions on permission rather than exclusion. This means that, by default, access is denied, and the protection scheme identifies conditions under which access is permitted"*

*- CERT Top Ten Practices*

*"A default allow policy is generally easy to manage but at the same time, less secure because anything not specifically denied will be allowed."*

**Default allow** is a type of protection scheme where rules are defined to block particular actions. For example, content filtering rules will have a "Deny" action for unwanted categories. This is a type of policy where everything is allowed except for certain actions.

A default allow policy is generally easy to manage but at the same time, less secure because anything not specifically denied will be allowed.

In contrast, a **default deny** scheme dictates that all actions are to be blocked except those that have been explicitly allowed. Such a policy is inherently more secure but requires more developer input.

This practice is aligned with the principle of least privilege, and arguably a consequence of it. It is especially pertinent to inter-domain communications – and hence to communications to and from the HMI, and between the UI and business logic.

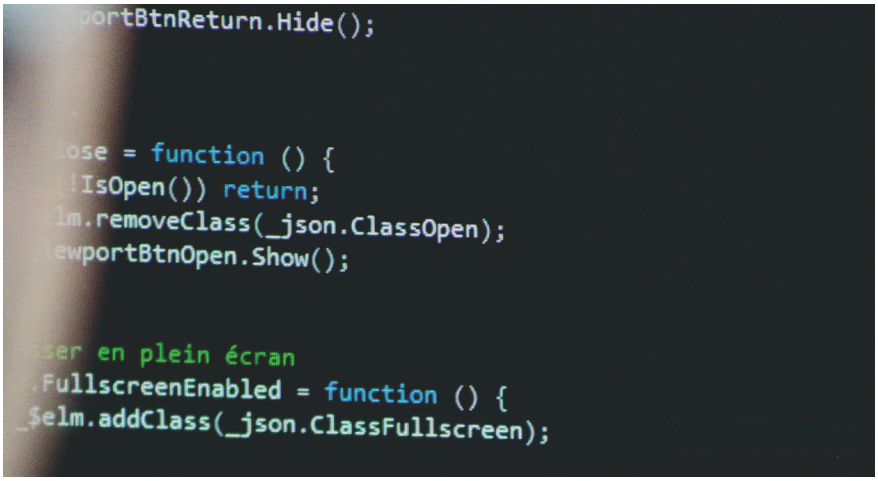



*Image 3: Person in front of a computer screen* ***Photo by: rivage***

# Secure HMI system development

All 12 of the CERT "best practices" and "bonus practices" have an impact throughout the development lifecycle. However, some of those practices are set in place during the development phase, and they are discussed in this section.

Traceability is a consistent thread throughout all phases. In the development phase, it is desirable to demonstrate that the implementation reflects the requirements, architecture, and design – and nothing more.

## Heed compiler warnings

*"Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws"*

*- CERT Top Ten Practices*

Many developers using compiled languages have a tendency to attend only to compiler errors during development and ignore the warnings. CERT's recommendation for compiled code is to set the warnings at the highest level available and ensure that all of them are attended to. SAST (Static Application/Analysis Software Test) tools identify additional and more subtle concerns than those exposed by compilers.

## Use effective quality assurance techniques

*"Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an*

*effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions"*

*- CERT Top Ten Practices*

The traditional approach to testing cybersecurity-critical code in the security market is largely reactive – so that the code is developed in accordance with relatively loose guidelines, and then it is tested by means of performance, penetration, load, and functional testing to identify and deal with any vulnerabilities. Although it is clearly preferable to ensure that the code is secure "by design", the tools used in the traditional reactive model still have a place. Their role in this scenario is to confirm that the system is secure, rather than to find out where it is not.

There are two primary types of test tool used in the development of a secure application:

**Static Application Security Testing (SAST)** tools help by analysing source code or compiled versions of code to help find security flaws, and check for adherence to coding standards. Such tools can help to detect issues during software development.

*Image 4: Apply effective quality assurance techniques during development to avoid introducing vulnerabilities.*

***Illustration by: Sjölund***

**Dynamic application security testing (DAST)** tools analyse an application while it's running. "White box" DAST tools can relate that execution to the source code. These include unit test tools, and code coverage tools. "Black box" DAST tools have no information relating to an application's internal interactions or designs at the system level, and no access or visibility into the source program. Penetration and fuzz test tools fall into this category.

## Model threats

*"Use threat modelling to anticipate the threats to which the software will be subjected. Threat modelling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases"*

*- CERT Bonus Secure Coding Practices*

High risk areas at the critical edge include data endpoints exemplified by the HMI. Communication between domains, particularly those of differing levels of criticality, should also be focal points throughout the development process. Addressing those focal points throughout the development lifecycle is key to optimizing the security of the system as a whole. ■

# Works Cited


[1] 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].

[2] K. Zetter, "Wired: "Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid"," 3rd March 2016. [Online]. Available: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/. [Accessed 11th February 2023].

[3] "Electric: High-Profile Company Data Breaches 2023," 3rd February 2023. [Online]. Available: https://www.electric.ai/blog/recent-big-company-data-breaches#:~:text=T%2DMobile%3A%20January%202023,from%20over%2037%20million%20customers.. [Accessed 11th February 2023].

[4] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho and Sarah Martin, "TrustZone Explained: Architectural Features and Use Cases," in 2016 IEEE 2nd International Conference on Collaboration and Internet Computing, Pittsburgh, PA, USA, 2016.

[5] Carnegie Mellon University, "Carnegie Mellon University - Software Engineering Institute - The CERT Division," Carnegie Mellon University, 2022. [Online]. Available: https://www.sei.cmu.edu/about/divisions/cert/index.cfm. [Accessed 13 April 2022].

[6] Seirdzio, "QT Forum: Best practice when decoupling logic and user interface," QT, 18th December 2018. [Online]. Available: https://forum.qt.io/topic/97767/best-practice-when-de-coupling-logic-and-user-interface. [Accessed 5th May 2023].

[7] Saltzer, J.H. and Schroeder, M.D., "The Protection of Information in Operating Systems," Proceedings of the IEEE , vol. 9, no. 63, pp. 1278-1308, 1975.

[8] J. Rushby, "Design and Verification of Secure Systems," in 8th ACM Symposium on Operating System Principles, Pacific Grove, California, 1981.

[9] M. Pitchford, "Applying MILS principles to design connected embedded devices supporting the cloud, multi-tenancy and App Stores," in 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), TOULOUSE, France, 2016.

[10] D. Serpanos and D. Stachoulis,, "Secure Memory for Embedded Tamper-proof Systems," in 14th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS), Mykonos, Greece, 2019.

[11] Helixstorm, "Immutable storage: What it is and why your business needs it," [Online]. Available: https://www.helixstorm.com/blog/what-is-immutable-storage-and-why-your-business-needs-it/. [Accessed 12th February 2023].

[12] Townsend Security, "The definitive guide to encryption key management fundamentals," [Online]. Available: https://info.townsendsecurity.com/definitive-guide-to-encryption-key-management-fundamentals. [Accessed 12th February 2023].

[13] 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].

[14] 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].

[15] Welsh Government, "Cadw: Beaumaris castle," Crown copyright, 2023. [Online]. Available: https://cadw.gov.wales/visit/places-to-visit/beaumaris-castle. [Accessed 25th February 2023].

[16] Peterson, Hope and Lavenhar, "Architectural risk analysis," CISA, 2005. [Online]. Available: https://cisa.gov/bsi/articles/best-practices/architectural-risk-analysis/architectural-risk-analysis. [Accessed 25th February 2023].

[17] Rajiv D. Banker, Srikant M. Datarand, and Dani Zweig., "Software Complexity and Maintainability," University of Minnesota and Carnegie Mellon University, 1998.


# Images


[1] ün LIU, üL. (2022). "A couple of people in a car" [photograph]. Unsplash. https://unsplash.com/photos/a-couple-of-people-in-a-car-jbiIhRsQ_-0

[2] Amélie Mourichon, AM. (2019). "Person writing on paper" [photograph]. Unsplash. hhttps://unsplash.com/photos/person-writing-on-printing-paper-wusOJ-2uY6w

[3] Rivage, Sigmund. (2019). "A close up of a persons face in front of a computer screen" [photograph]. Unsplash. https://unsplash.com/photos/a-close-up-of-a-persons-face-in-front-of-a-computer-screen-2qd9noyeuRE

[4] Sjölund, Philip. (2024). "Data breach" [illustration].

## About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build runtimes, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com



KDAB
Trusted Software Excellence

CERTIFIED ISO 9001:2015 COMPANY