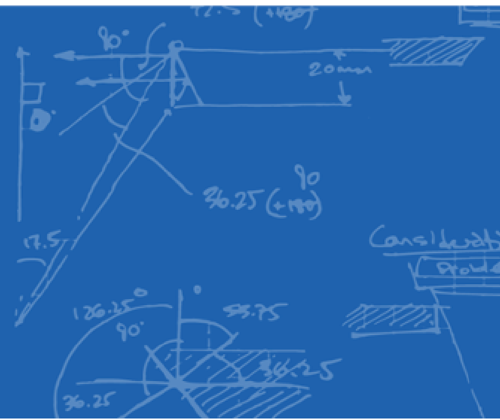
WHITE PAPER | NOVEMBER 2023

Best Practices for Developing Secure Embedded Systems



The Market Leader in Mechatronics and Detailed Design Engineering Services | simplexitypd.com

INTRODUCTION

Embedded systems are in millions of products that we use every day. It's easy to take it for granted that you can unlock your front door from your phone remotely and that your watch keeps track of the number of steps you take and your heart rate profile. But you don't want other people to be able to unlock your door or see your heart rate profile, or to hack into your home Wi-Fi. In other embedded systems, such as medical devices or industrial control systems, security breaches can have more serious consequences.

In fact, it is so important that in the White House's recently released National Cybersecurity Strategy, developing security into IoT devices is highlighted as one of the strategic objectives. Additionally, the strategy proposes to shift liability for insecure products and services from the consumer to "those entities that fail to take reasonable precautions to secure their software". Europe, too, will soon release regulations to require manufacturers to protect internet connected devices from unauthorized access.

The motivation to consider security in all appropriate embedded devices couldn't be clearer. Yet, security is an area that has been neglected for too long for embedded systems. And what exactly does security *mean* for embedded systems?

In this whitepaper, we will explore the following:

1. A proven process and methodology to enable development of secure IoT devices.
2. What it takes to analyze a real system for security issues, and how to manage mitigation of any identified security issues.
3. Several coding practices that can avoid introducing many of these security issues in the first place.

1. USE A DEVELOPMENT PROCESS FOCUSED ON SECURITY

We first look at a process that provides methods to support the development of a secure IoT device. This process encourages appropriate security activities to be followed throughout the product development process. This will help to ensure the appropriate analyses and techniques are used during development, which will help identify and mitigate security vulnerabilities in a cost-effective manner.

A Secure Software Development Lifecycle (SSDL) is a software development methodology with security activities at each stage of development. The key objectives of an effective SDL include identifying potential security risks and vulnerabilities early in the development process, defining and implementing security requirements, and integrating security testing and validation into every stage of development. By following a well-designed SDL, software developers can reduce the risk of security vulnerabilities and improve the overall security and quality of the software they produce. This can help to prevent data breaches, protect user privacy, and ensure that software systems are reliable and trustworthy.

Microsoft's Security Development Lifecycle (SDL) provides an excellent resource when creating a process for secure system design. Though not specific to embedded systems, the guideline's principles apply across any industry. The Microsoft SDL uses the following criteria to determine whether development of a device should follow a Secure Development Lifecycle.

- Will the system be deployed in a business or enterprise environment?

- Does the system process personally identifiable information or other sensitive information?
- Does the system communicate regularly over the Internet or other networks?

If any of the above answers are yes, then development should follow a Secure Development Lifecycle. Embedded systems routinely satisfy the above considerations, especially as related to network communications, so we must be serious about incorporating security into our development processes for embedded systems.

The following sections describe how Microsoft's SDL fits into a standard product development process. In this example, we are using Simplexity's phase-based Product Development Process although other product development processes can be employed in a similar fashion.

Phase 1 Requirements and Planning

During Phase 1, as we begin considering our project and software development plans, we must also begin planning the security approach. Key questions include: Is security applicable for this product? What level of security training is needed for the team?

The team should determine what quality standards to hold the firmware to. For example, here at Simplexity, we don't allow a branch to be merged to the mainline until it passes its static analysis checks and automated unit testing. In addition, we adhere to established coding standards which define practices that avoid common weaknesses and vulnerabilities.

We quickly turn to requirements gathering, including requirements related to security, and document them in the Product Requirements Document (PRD). Because security impacts fundamental architectural decisions, it is essential to define security requirements during Phase 1. These requirements also impact the definition of milestones and schedule, feeding back into the development plans.

Phase 2A Architecture and Feasibility

As an input to the software architecture being developed during this phase, we need to perform a risk assessment of the security and privacy of the application. This risk assessment follows a similar process to a medical device risk assessment except that we're considering the probability and impact of known threats rather than harm to a patient.

We derive the software requirements specification (SRS) from the product requirements specified in Phase 1. In addition to specifying the features of the firmware application, the SRS must also describe how to implement functionality provided by a particular feature in a secure fashion.

These requirements must be reviewed to ensure that they accurately and completely describe the intended use of a feature as well as describing how to securely utilize the feature. As with the Product Requirements, it is important to specify security features before making architectural decisions that limit your design options. Security specifications may drive critical architectural decisions such as processor selection, memory size, and communication technology (e.g. Wi-Fi, Bluetooth) that must be made during this phase.

Threat modeling is the process of identifying potential threats to a system, determining the likelihood and potential impact of those threats, and identifying countermeasures to mitigate the risks. This process provides a foundation for understanding the potential attack surface of a system and the types of attacks that may be possible. Threat modeling will be discussed in more detail below.

Once potential threats have been identified through threat modeling, we turn our focus to attack surface reduction. Attack surface reduction aims to limit the ways in which an attacker can gain access to or compromise the system. These strategies can be designed to specifically address the potential threats identified in the threat modeling process.

This phase is also where we decide on the tools used for the project. Simplexity has a default codebase and set of tools and processes used, but at times clients have specific tools preferences. In terms of security, this is when we define the tools and security checks that will be used for the remainder of the project. Detailed design and implementation cannot proceed without development tools in place.

Attack surface reduction techniques

Using secure boot: Secure boot ensures that only authorized firmware is loaded during system boot-up, preventing unauthorized code from executing.

Restricting firmware updates: Limiting firmware updates to authorized personnel and ensuring that the updates come from a trusted source can reduce the risk of malicious updates or firmware tampering.

Disabling unused hardware and software features: Disabling unused features that are not required for system operation can help to reduce the attack surface of the system. For example, a serial port used for debugging identified as a potential security risk should be disabled before release.

Applying the principle of least privilege: Limit the access of embedded devices or components to only the resources they need to operate effectively and securely.

Phase 2B/2C detailed design and implementation

While the hardware team works on the detailed design of the first prototypes, the firmware team iterates through the design-code-test cycles for each feature, including security features. Detailed design fleshes out the details outlined in the architecture—the details of the requirement to do secure boot can be ironed out in this phase, for example. As each feature is implemented, it should also be unit tested. The unit testing is as automated as possible.

At this stage, the team must review functions and APIs being used and prohibit any functions that are determined to be unsafe for that application. For example, if the project is avoiding dynamic allocation, printf may not be allowed since many implementations allocate memory. Using header files to list banned files and replacements (e.g., banned.h, strsafe.h), allows the compiler to check for the existence of those functions and use a safer implementation. We will highlight some techniques for writing secure C/C++ code later.

Static analysis should be performed on the source code. Static analysis can catch errors in the code such as buffer overflow and can help ensure that secure coding policies are being followed. At Simplexity, static analysis is a standard part of our continuous integration process. The tool is enabled at the beginning of a project, and a release cannot be completed until the static analysis checks pass.

Of course, static analysis does not replace a manual code review. Most modern tools such as GitHub or Atlassian's Bitbucket can be configured to require a code review, which happens within the tool, before merging a change to the main branch of code. Code reviews are best practice in general and are an important tool in ensuring secure code.

At this point the team has performed static analysis and unit and integration testing, so now attention must turn to doing run-time analysis and testing of the whole system to prove that it works as defined in the requirements. In addition to executing tests against the requirements, we use run-time dynamic analysis tools such as Percepio's Tracealyzer, which enables the developer to monitor CPU and memory usage. This type of analysis helps identify vulnerabilities related to memory leaks, buffer overflows, stack overflows or side-channel timing attacks.

Fuzz testing is a type of dynamic program analysis that involves providing invalid or unexpected inputs to an application to see how it responds. The goal is to identify any unexpected behavior or crashes that may indicate a security vulnerability. The intended use of the device along with the functional requirements drive the specific fuzz testing strategy for a particular product.

By the end of Phase 2C, the firmware is feature-complete, including security features. Engineering level testing, including unit and integration testing, dynamic program analysis is complete.

Phase 3 verification and transfer to manufacturing

By Phase 3, the implementation is complete, and significant engineering testing has been completed. The system verification test protocols should be complete and ready to execute, so that we can have formal documentation of our test results. Before system verification, the software team prepares a *candidate* release build, which will be the version used for the testing.

This phase is also the appropriate time to work with our clients on a software maintenance plan, which for security, needs to include an Incidence Response Plan. As technology changes, new threats arise, even for applications with no known vulnerabilities when they were released. Most likely the clients will take on this responsibility, but the most important thing is for the responsibility to be identified. The plan identifies the sustaining engineering team, who to contact in a security emergency, on-call contacts, and security servicing plans for third-party code.

A Final Security Review (FSR) is a process of evaluating and verifying the security measures that have been implemented in a system or application before it is released or deployed into production. It should be performed during Phase 3 and happens before the official release of the firmware. A security advisor, along with other stakeholders and developers, reviews the security activities completed during the project. The team will review the threat models, change requests, tool output (such as static analysis output), and performance compared to the quality standards agreed upon in Phase 1. The results of the review are one of the following:

- **Pass** – All known security and privacy issues have been solved.
- **Pass with exception** – There are some remaining issues to be resolved in the next release.
- **Fail with escalation** – The security team cannot approve the release in its current form. The team must address critical issues and reassess in another FSR or escalate to management.

At this point, the firmware has been through functional verification as well as passed security tests such as fuzz testing. If the Final Security Review passes, the candidate release, including release notes which contain lists of known issues, new features, and defects fixed in this version, can be formally released to manufacturing. All relevant information is packaged up and archived, including the specifications, source code, executables, and documentation.

The goal of a Security Development Lifecycle is to integrate security into the software development process from the very beginning, rather than treating it as an afterthought. The SDL aims to ensure that security is considered at every phase of the software development lifecycle, from architecture to release to manufacturing, and that appropriate security measures are implemented and tested thoroughly. Now that we understand the process for developing a secure product, let's look at how to model the product under development with the goal of identifying and mitigating security threats.

2. MODEL YOUR SYSTEM TO IDENTIFY SECURITY THREATS

Threat modeling is a process for identifying critical assets within a product, discovering potential threats to those assets, evaluating the likelihood of the threat, and defining security requirements to mitigate the threats.

We will explore the process of creating a threat model for a simple, fictitious product. We will use the Microsoft Threat Modeling Tool to create our threat model.

Product Definition

Our product will be a simple remote temperature sensor, known as TS1. The device consists of two physical parts – a hub with BLE and Wi-Fi capabilities and a temperature sensor with BLE capabilities. The sensor will monitor the temperature and send the temperature reading over BLE to the hub. The hub will store a history of temperature readings and respond to requests from a phone app for temperature data over Wi-Fi. We won't delve into the detailed requirements here, but instead make some basic assumptions about the BLE and Wi-Fi interactions between devices. Both transports are frequently found in IoT products, so understanding the security implications inherent with their use is important.

To begin, it's helpful to identify some properties of the product:

- What are the external dependencies?
- What are the entry points?
- What are the components of the system?

What are the trust levels between these components?

In our example, we have several external dependencies:

- Mobile app
- Network infrastructure to support the Wi-Fi connection

We also have several entry points:

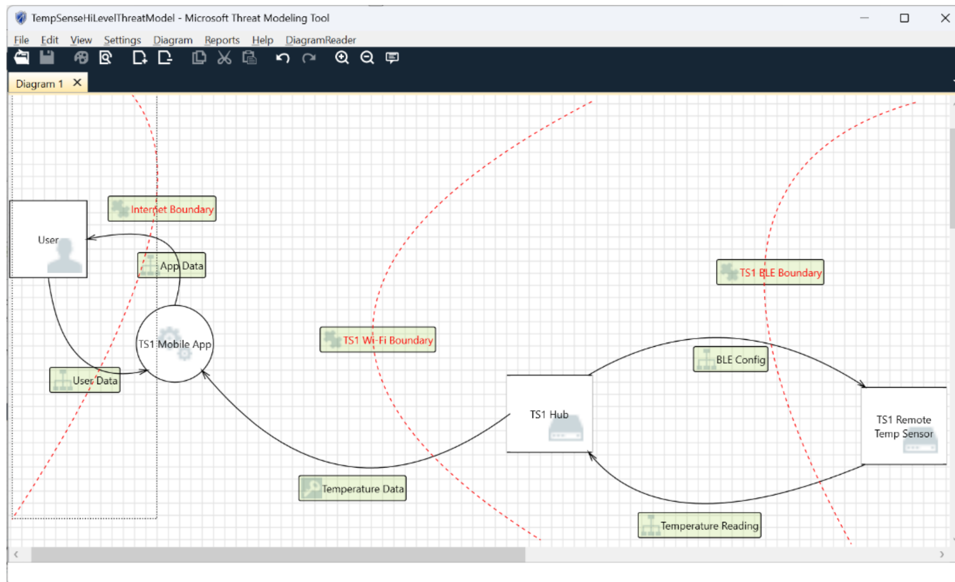
- BLE interface
- Wi-Fi interface

- Temperature sensor

Data Flow Diagrams and Threat Identification

When these properties have been identified, you can then draw a high-level data flow diagram (DFD), showing the components and the trust levels between these components.

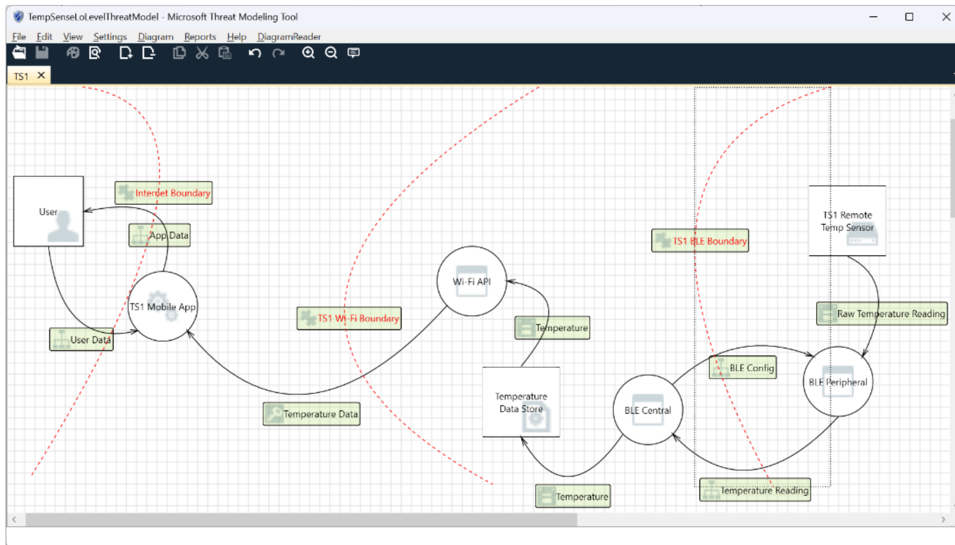
For TS1, a high-level DFD may look like this:



In our diagram above, we've used the following elements:

- Processes - Circles
- Data Stores – Squares with the bold top and bottom edges
- External Interactor – Squares with all four edges bold
- Data Flow – Solid black arcs
- Trust Boundary – Dashed red arcs

The high-level DFD defines the scope of the product. It clarifies how data flows through the system as well as how it changes as it flows. Using the high-level DFD as a reference, we can then iteratively decompose the application into multiple processes. As we generate lower-level DFD's of the system, we will continue to iterate and refine on the external dependencies, entry points, and trust levels. A low-level DFD for TS1 may look like this:



Selecting the “Analysis View” under the View menu results in 36 threats identified. A report can also be generated which details each threat.

The Analysis View of our low-level DFD for TS1 looks like this:

TempSenseLoLevelThreatModel* - Microsoft Threat Modeling Tool

Threat List

ID	Diagram	Changed By	Last Modified	State	Title	Category	Description	Justification	Interaction	Priority
180	TS1		Generated	Not Started	Potential Data F	Repudiation	BLE Peripheral c		BLE Config	High
181	TS1		Generated	Not Started	Data Flow Sniffi	Information Dis	Data flowing ac		BLE Config	High
182	TS1		Generated	Not Started	Weak Credentia	Information Dis	Credentials on t		BLE Config	High
183	TS1		Generated	Not Started	Potential Proce:	Denial Of Servi	BLE Peripheral c		BLE Config	High
184	TS1		Generated	Not Started	Data Flow BLE C	Denial Of Servi	An external age		BLE Config	High
185	TS1		Generated	Not Started	Elevation Using	Elevation Of Pri	BLE Peripheral r		BLE Config	High

Export Csv 36 Threats Displayed, 36 Total

Threat Properties

ID: 181 Diagram: TS1 Status: Not Started Last Modified: Generated

Title: Data Flow Sniffing

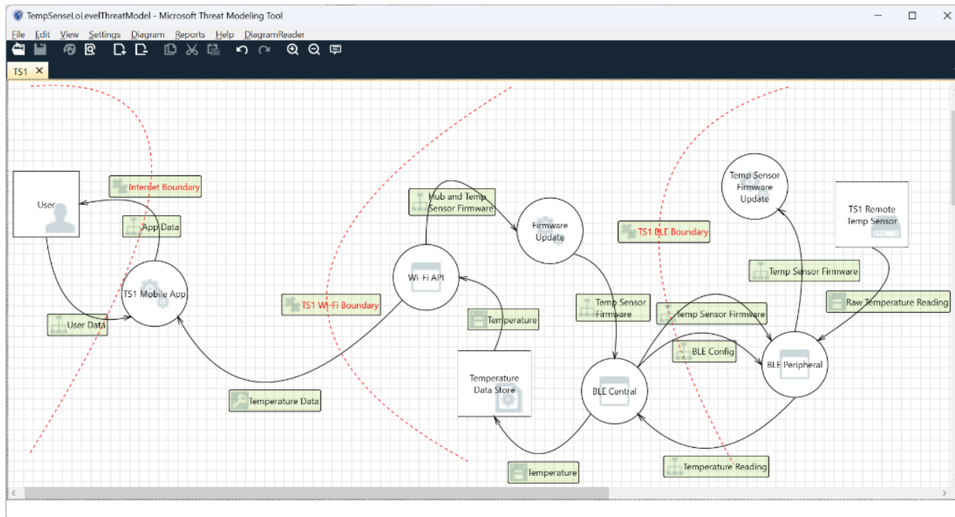
Category: Information Disclosure

Description: Data flowing across BLE Config may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.

Threat Properties | Notes - no entries

As each threat that is identified is analyzed, the Status field can be updated to indicate the current state of this threat. The Status field defaults to “Not Started”. Other options are “Needs Investigation”, “Not Applicable”, or “Mitigated”.

As you continue to investigate threats, you may discover missing diagram elements. As these are added, new threats may be identified. For instance, the entire firmware update process was overlooked in our initial analysis. If we add this to our low-level diagram DFD, it may look like this:



When we re-run the threat analysis, we now have 62 threats identified.

Investigate Threats

Now that we have a list of potential threats, the next step is to investigate each of them. You may decide that some threats are not applicable, others are addressed by the current implementation, and yet others will require changes to mitigate the threat. In any case, each threat should be reviewed, and its status updated.

Let’s take a closer look at one of the threats identified by the tool. Threat ID 181 identifies a potential threat to the BLE Config dataflow between the BLE Central and the BLE Peripheral. The description of this threat says:

> *Data flowing across BLE Config may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.*

Not only do we get a description of the threat, but also a potential mitigation. By encrypting the communications between the BLE Central and the BLE Peripheral, we can prevent an attacker that is sniffing this connection to gain any useful information, other than the timing of our communications. If an attacker is using a tool such as Wireshark to eavesdrop on the communications between the BLE Central and the BLE Peripheral, none of our data will be exposed. We'll plan to enable encryption on the BLE link and consider this threat mitigated. Add this plan to the Notes field of this threat and change the Status to Mitigated.

One down, 61 to go!

Once all threats have been reviewed and mitigation measures identified for each, you'll have made significant progress on developing a secure temperature sensor.

We've highlighted the importance of identifying external dependencies, entry points, and system components, as well as establishing trust levels between them. We've developed multiple levels of data flow diagrams using the Microsoft Threat Modeling Tool, which in turn has identified potential threats to our temperature sensor. We've reviewed each threat and developed appropriate mitigation strategies for each threat. But the work doesn't end there. Systems are rarely static. External dependencies may change, and most products go through multiple revision cycles. The threat model should be reviewed on a regular basis to identify new potential threats and adapt to changing environments.

3. AVOID INTRODUCING SECURITY THREATS

Embedded systems predominantly use C and C++ for the firmware. Some common coding mistakes in those languages can lead to security vulnerabilities that can be exploited by an attacker. We will discuss five primary vulnerabilities, their possible use by an attacker, and ways to mitigate these problems.

1. Buffer overflows

Buffer overflows occur when a program writes more data into a buffer than its allocated size, leading to memory corruption. Attackers can exploit this vulnerability to overwrite adjacent memory and execute arbitrary code. Buffer overflows are a significant concern in C and C++ due to the manual memory management they offer.

To mitigate buffer overflows, developers should avoid using unsafe library functions. `strcpy` and even the safer `strncpy` should be avoided. Although `strncpy` is safer than `strcpy`, it can still lead to vulnerabilities. `strncpy` will limit the number of characters copied, but there is no guarantee that the destination string will be NULL-terminated.

Use `strlcpy` instead. This function limits the number of characters copied to one less than the number requested and also NULL-terminates the end of the destination string. Both of these qualities are crucial to ensure that data fits within the allocated buffers and that subsequent calls to `strlen` do not cause

vulnerabilities. The return value from `strlcpy` should still be checked to verify that the destination buffer was large enough to hold all the characters.

```
if (strlcpy(dst, src, size) >= size) {

    // Handle loss of data error!

}
```

String format functions, such as `sscanf` and `sprintf`, are also potential sources of buffer overflows. There is no size check on %s which could cause overflow. In the statement, `sprintf(str,"%s",src), src` is interpreted as the format string, and there is no check on its size. A malicious string could be entered such as "%182d", causing overflow. To mitigate these issues, it is better to include the field widths, use the safer `snprintf` function, and check the return value to determine the number of characters written. The statement above could be re-written more safely as follows:

```
char str[16];
char *src;

if (snprintf(str, 16, "%16s", src) >= 16) {
  // Handle string not terminated error!
}
```

Note also that there is a safe version, `sscanf_s`, in C11 Annex K, which does run-time size checking.

Additionally, `memcpy` and `memmove` should be avoided in favor of the "safe" versions, `memcpy_s` and `memmove_s`. Each includes an additional parameter specifying the maximum number of bytes to copy. These functions are also part of C11 Annex K.

2. Integer vulnerabilities

Integer vulnerabilities arise from improper handling of integers, leading to overflows, underflows, or unexpected behavior. These vulnerabilities can result in data corruption, crashes, or even security breaches.

To mitigate integer vulnerabilities, developers should perform input validation and properly check for boundary conditions. It's crucial to use appropriate data types with sufficient range to accommodate expected values and avoid undefined behavior. Static analysis can detect many integer errors and should always be used in a secure embedded system.

In this case, using a signed rather than unsigned `int` for the size of our buffer can result in unexpected behavior.

```
void myCopy(char *dest, char *src, int bufLen)

{

    int maxBufLen = 20;
```

```
    if (bufLen < maxBufLen) {
        strncpy(dest, src, bufLen);
        // If bufLen == -1, we just copied ~4 billion bytes!
    }
}
```

3. Concurrency issues

Concurrency issues arise when multiple threads or processes access shared resources simultaneously without proper synchronization. These vulnerabilities can lead to race conditions, deadlocks, or data inconsistencies.

Concurrency issues are among the most difficult classes of problems to debug. Several techniques can help from introducing these issues in the first place:

- Understand your system well enough so that you know what thread of execution each part of your code runs on. Note that you may have functions that can run on multiple threads, including interrupt threads. These areas should be reviewed closely for concurrency issues.
- Avoid the use of global variables whenever possible. The use of global variables increases the chances of introducing a concurrency defect.
- Use the "volatile" qualifier when declaring variables that can be accessed from multiple threads of execution.
- Consider the need for protecting reads and writes to each global and volatile variable with some form of concurrency protection.

To address concurrency issues, developers should use thread-safe synchronization mechanisms, such as mutexes, semaphores, or condition variables, to protect critical sections of code. Proper design, utilizing thread-safe data structures, and avoiding unprotected shared data access can ensure data integrity and prevent conflicts among concurrent executions. Thorough testing and stress testing with simulated concurrent scenarios can help uncover and address potential concurrency vulnerabilities.

4. File system vulnerabilities

File system vulnerabilities occur when file operations are not properly handled, leading to unauthorized access, file tampering, or information disclosure. C and C++ provide low-level access to file system operations, making it essential to handle them securely.

Embedded systems frequently interact with external storage or file systems for data logging, configuration files, or firmware updates. File system vulnerabilities can expose the system to unauthorized access, data corruption, or malicious code injection.

To mitigate file system vulnerabilities, developers should carefully validate user input and enforce proper file permissions to restrict unauthorized access. Careful input sanitization, utilizing MCU-specific secure file APIs, and adopting the principle of least privilege can help prevent file system vulnerabilities. Data read from a file should be treated as untrusted until it has passed an integrity check. Always test for errors whenever files are opened or read.

Utilizing secure file system APIs and implementing secure file handling functions can help prevent common file system attacks such as path traversal or directory traversal. Employing cryptographic techniques like digital signatures or secure boot mechanisms can ensure the integrity and authenticity of firmware or software updates for embedded systems.

5. Inadequate error checking

Inadequate error checking refers to the lack of proper validation and handling of errors during program execution. This can have severe consequences, including system failures or safety hazards. Ignoring or mishandling errors can result in unexpected behavior, leaving the system vulnerable to security breaches or unreliable operation. For example, unhandled exceptions can be used as Denial-of-Service attacks, or error messages may give attackers insights to the program's implementation.

To address inadequate error checking, developers should diligently check return values of system calls, library functions, and input validation routines. Robust error handling, including graceful degradation, appropriate error messages, and logging, is crucial in embedded systems to provide meaningful diagnostics and facilitate troubleshooting. Employing techniques like watchdog timers can help detect and recover from critical errors or system failures.

Developers working with C and C++ must be aware of common coding vulnerabilities to ensure the security and reliability of their applications. By understanding and addressing buffer overflows, integer vulnerabilities, concurrency issues, file system vulnerabilities, and inadequate error checking, developers can implement best practices to strengthen the security of their code. Regular code reviews, adherence to secure coding guidelines, and utilizing tools like static analyzers can further enhance the resilience of C and C++ applications against potential exploits.

CONCLUSION

We have shown how the key principles of Microsoft's SDL can be integrated with a typical product development process to reduce security vulnerabilities and improve product quality. We also explored what it takes to perform threat modeling on the product to identify and mitigate specific security threats. Finally, we explored some best practices during coding to avoid introducing many of these security issues.

Ensuring security in an embedded device is an ongoing process. Even if you have diligently followed the steps outlined above during development, your product may still be exposed to security risks. The threat landscape is always changing, and your device will be exposed to new threats after shipping that were not identified or didn't exist during development. Be prepared to handle security issues that arise in the field and be responsive to fixing those issues as they arise. Doing so will enable you to stay one step ahead of hackers trying to attack your device.

ABOUT THE AUTHORS

Katie Elliott is the Director of Firmware & Quality Engineering at Simplexity Product Development working out of the Seattle office. She has a BS and MS in Electrical Engineering from the University of Washington. Katie has over 20 years of experience writing software and firmware for embedded systems, with a recent focus on connected medical devices.

Brian Peavey is a Principal Firmware Engineer at Simplexity Product Development. He has a BS in Electrical Engineering from the University of the Pacific. Brian has 40+ years of experience writing firmware for a wide range of applications, with a focus on firmware infrastructure, low-level drivers, and connected IoT devices.

To LEARN MORE about Simplexity, review Simplexity's Product Development Process or contact them about your next design engineering project.

http://www.simplexitypd.com