Embedded Office

Security in Embedded Systems

This whitepaper gives an introduction to your software development team on how to achieve a certain level of security in embedded systems.

Embedded Office, M. Riegel

2022-04-26

Contents

1 Security in Embedded Systems 2
1.1 Threat Modeling . . . 2
1.1.1 IEC 62443 Security Levels . . . 3
1.1.2 IEC 62443 Security Requirements . . . 4
1.1.3 Security Zones . . . 4
1.2 Secure Components . . . 5
1.2.1 Interface Contracts . . . 5
1.2.2 Enforce Expectations . . . 5
1.2.3 Communication Protocol Verification . . . 6
1.2.4 Isolate Components . . . 6
1.2.5 Security Checklist . . . 6
1.2.6 Cryptographic Algorithms . . . 6
1.2.7 System Configuration . . . 7
1.2.8 System Logging . . . 7
1.3 Secure Deployment . . . 8
1.3.1 IT Environment . . . 8
1.3.2 Software Transmission . . . 8
1.4 Active Maintenance . . . 9
1.5 Secure Update and Secure Boot . . . 9
1.5.1 Secure Update . . . 10
1.5.2 Secure Boot . . . 10

2 Conclusion 12

3 Appendix - Threat Awareness 13
3.1 Message Forgery . . . 13
3.2 Replay Attack . . . 13
3.3 Denial of Service Through Division by Zero . . . 13
3.4 Information Loss Through Buffer Over-Read . . . 14
3.5 Remote Code Execution Through Buffer Overflow . . . 14
3.6 Timing Attack Against RSA Implementation . . . 14
3.7 Deliberate Power Off . . . 15
3.8 Abuse Non-Standard Encoding . . . 15

4 Embedded Office Profile 16
4.1 About the Author . . . 16
4.2 Contact . . . 16

1 Security in Embedded Systems

In a controlled environment, preventing accidental misuse and hardware faults was enough to achieve safe behavior. When an unrecoverable state is detected, the system can enter a state with limited or no functionality and is still considered safe.

In an uncontrolled environment, various forms of sabotage could impact the system's safety or availability. Preventing those is only possible by considering security at each step of the lifecycle:

1. **Thread Modeling** - While designing, the product owner team must identify the security requirements.
2. **Secure Components** - The software developers must implement the security requirements correctly during implementation. In addition, the developers must implement all other requirements so that they do not add vulnerabilities to the system.
3. **Secure Deployment** - Whenever a company transfers software (e.g., from supplier to manufacturer), the development team must ensure integrity and authenticity.
4. **Active Maintenance** - During the system's lifetime, the product manufacturer must take and fix all discovered weaknesses in any component.
5. **Secure Update and Boot** - When updating the system, the product manufacturer must ensure the integrity and authenticity of the new software.

1.1 Threat Modeling

The first step to achieving a secure system is identifying the assets or values that a system provides. The product owner team then use threat modeling to analyze how an attacker can threaten those values. Threat modeling should be integrated into the design process and done at all abstraction levels. See Appendix *Threat Awareness* for some example threats.

When specifying the high-level requirements, software architects also identify abstract threats. These threats will lead to additional requirements to mitigate those threats. This process is integrated in each design step accordingly.

Values that many embedded systems have to protect:

- System Safety: Most types of attacks can impact the safety of a system.
- System Availability: If attackers can shut down the system, the system becomes useless to the customer.
- Business Secrets: If attackers have access to the firmware, they can extract included trade secrets.
- Legal Compliance: If attackers can cause the system to violate laws (send spam emails, spy on people), this will cause legal repercussions for the producer.
- Company Reputation: A malfunction of the system could reflect poorly on the producer.

After identifying the values of our system, the product owner team need to estimate the costs and resources an attacker is willing to invest in attacking those values. With this information, the team can define the required level of security that the product shall achieve. From this level of security, the developers can derive security requirements to mitigate risks.

1.1.1 IEC 62443 Security Levels

To classify the level of security a component achieves, the IEC 62443 defines 5 Security Levels (SL):

- **Security Level 0**: No special requirement or protection is required.
- **Security Level 1**: Protection against unintentional or accidental misuse.
- **Security Level 2**: Protection against intentional misuse by simple means with few resources, general skills, and low motivation.
- **Security Level 3**: Protection against intentional misuse by sophisticated means with moderate resources, system-specific knowledge, and moderate motivation.
- **Security Level 4**: Protection against intentional misuse using sophisticated means with extensive resources, system-specific knowledge, and high motivation.

When developing a system according to Functional Safety standards, the security level SL-1 is covered without further activities.

The terms “few”, “moderate” and “high” aren’t well defined for higher security levels. However, a common understanding for the security levels is:

- SL-2 protects against a hobbyist or angry former employee that consults publicly available information about security and the system he wants to attack.
- SL-3 protects against professional hackers who intend to make money by blackmail or by selling the exploit or the information they can extract.
- SL-4 protects against professional hacker groups that receive extensive funding from companies or governments.

1.1.2 IEC 62443 Security Requirements

IEC 62443 lists functional requirements that a component must implement to meet a security level.

For example, "*A human that interacts with the system …*":

- SL-1: must be identified and authenticated.
- SL-2: must be uniquely identified and authenticated (no shared admin account).
- SL-3: must be uniquely identified and authenticated via multi-factor authentication if accessing from an untrusted network
- SL-4: must be uniquely identified and authenticated via multi-factor authentication on all networks

1.1.3 Security Zones

Barriers that improve security (walls, doors, security guards, firewalls, virtualization technology, etc.) divide a system into zones, and each zone achieves the lowest security level of any of its components.

While the IEC 62443 addresses operational technology security in automation and control systems, the defined security levels are a valuable tool for any discussion about security.

Important

Security is not an attribute that a system either has or doesn't have. Thinking of security as the effort an attacker has to invest makes it comparable with the value that is to be protected.

This statement allows bringing security's cost into the equation.

1.2 Secure Components

The second step to achieving a secure system is to ensure that each component is designed and implemented securely and provides a secure interface to other components.

A component developer must follow the “secure by design” principle by considering the requirements of the applicable security standard and by incorporating the findings of the threat modeling at each step of the design process.

1.2.1 Interface Contracts

Besides implementing the functional requirements of a component correctly, the security of an implementation relies on closely following the contract of all the other components used. Contract, in this case, means the requirements that a component has of its callers. For example:

- Each memory block that is taken from a pool shall be returned to the same pool (the pool implementation does not track a block’s size or which pool it came from).
- The argument to a function shall not be zero (because the argument may be used in a division without being checked).
- A function must shall be called from within a critical section (because it operates on a shared data structure without using synchronization).

The reasons to place such restrictions on the caller can be many (performance, flexibility, portability, …) and are not always documented. Nevertheless, the security of the system relies upon following these requirements.

1.2.2 Enforce Expectations

When implementing a component, the software developers should increase the security by enforcing those contracts where possible.

Because software can’t enforce all expectations by implementation, the component author must describe the remaining unenforced expectations very clearly in the component’s documentation. Then, the component users will follow these expectations in the rest of the system. The adherence to those

contracts is checked using proper software development techniques like reviews, static analysis, testing, etc.

1.2.3 Communication Protocol Verification

Components implementing communication endpoints must ensure that all communication messages comply with the agreed-upon and documented protocol.

Specifically, this means verifying every type, value range, size, and encoding of every field of every message, but also meta data like the number of messages per time, the sender address, and the expected order of messages. The software must check enumerations against a safe list instead of excluding items from a deny list. The reaction to protocol violations must itself be designed not to be abusable.

1.2.4 Isolate Components

Sometimes the product owner team wants to use a component that does not meet the level of security of our system. In that case, it may be possible to execute that component in an isolated environment with limited privileges.

One way to do this is to restrict memory access with an MPU and CPU time with a preemptive scheduler. The damage from exploiting a weakness of the component is now limited to the isolated environment and does not affect the rest of the system.

1.2.5 Security Checklist

You can only consider a component as secure if it has a reasonably simple API that guides toward correct usage. Such a component needs detailed, up-to-date documentation, including a security manual: A checklist of every step that the user has to take to ensure the secure usage of the component (e.g., “run this validation”, “compile with these options” or “include your public key in this constant”).

1.2.6 Cryptographic Algorithms

The security of a system most likely relies on some cryptographic operation, either encryption/decryption or creation/validation of signatures and checksums. Those cryptographic operations only provide security if:

- Well analyzed, standardized, state of the art algorithms are used, which are

- expertly implemented and
- can be replaced by something more secure in the future.

Very few people have the knowledge and way of thinking to design good cryptographic algorithms. New algorithms are published, and after crypto analysis experts have failed to break them for years, they may be considered secure (for the time being).

Implementing an algorithm securely is almost as tricky. Countless hurdles have to be overcome, from choosing the suitable padding scheme to protecting against timing attacks or using the cryptographic primitives in the correct mode.

Important

It is a widely accepted best practice to rely on the solution of a reputable third party.

1.2.7 System Configuration

If an embedded device has configuration parameters that influence its security, it should have secure defaults. For example, a password-protected interface should either have a unique password or force the user to change the password before the device becomes operational.

Using a standard password and forcefully suggesting to the user for changing the password is typically not enough. In the same way, the system design should enable encryption by default instead of recommending the user to enable it later.

1.2.8 System Logging

Logging security-relevant events is essential for auditing and analyzing security breaches but challenging to implement. The integrity and confidentiality of the log may be a value that product owner team need to analyze during threat modeling.

For analysis, it is desirable to include the largest amount of information and detail in the log. However, system resources constrain this decision. If it is not possible to ensure the log's confidentiality, many valuable pieces of information must be left out. Furthermore, laws like the GDPR may forbid us from logging some data, or require them to be deleted after a short period of time.

Another aspect the software developer must consider is log flooding. If attackers do something that produces many log records, the less relevant information could overwrite the critical information.

1.3 Secure Deployment

The third step to achieving a secure system is to ensure that software, both compiled and in source, and all documentation is only stored on secure media and transmitted over a secure channel.

Instead of trying to abuse a system's weakness, an attacker might find it easier to create such a weakness by manipulating parts of the design, code, or binary before it is put on the embedded system.

1.3.1 IT Environment

The attacker could achieve this by accessing or manipulating the developer's machine, the server with the source code management system, or manipulating the software during transmission.

Measures to protect against manipulation of developer machines and servers:

- Proper IT rights management
- Regularly updating all used software
- Limiting physical access
- Security policies (e.g., employees have to lock their computer when not sitting in front of it)

1.3.2 Software Transmission

There are also various measures to prevent the software from being manipulated while it is transmitted:

- Establish a secure communication channel using PGP or X.509 certificates to sign (and encrypt) all communication.
- In parallel to sending a delivery by email, communicate a cryptographically secure fingerprint over a different channel (phone, letter, encrypted chat, etc.)
- Use a data transfer portal secured via TLS, X.509 certificates, authentication, and authorization.

1.4 Active Maintenance

The fourth step to achieving a secure system is to ensure that all components stay secure when working in operation.

Research on protocols, cryptography, and libraries could discover a significant weakness at any time, and the product end-users must update security systems as a consequence of this. Vendors of software components need to establish a system to inform all of their clients of discovered weaknesses. Users of those libraries need to receive this information to create and deploy updates for the software of their systems.

1.5 Secure Update and Secure Boot

The fifth step to achieving a secure system is to ensure the integrity and authenticity of all software that runs on the system. In principle, this can be achieved by:

- Only the manufacturer can install software
- The update process is secure, or
- The boot process is secure.

When end-user can't update software, a system with a known vulnerability must be deactivated and replaced by an improved device. While this could be a viable solution in some cases, the usual embedded system will need a way to get new firmware installed on it.

When an attacker gains physical access to the embedded system, manipulations in the contents of the stored firmware are possible. In such a situation, the boot process needs to be secured. Otherwise, a secure boot is unnecessary, and a secure update process is adequate. The secure update process does not affect boot times and is adequate on most embedded systems.

Both mechanisms have to ensure the application's:

- **Firmware Authenticity**: A trusted party created the firmware
- **Firmware Integrity**: No other party modified the firmware
- **Firmware Age**: The update must provide a newer firmware than the currently installed one (preventing rollback to a more vulnerable version)

The product manufaturer should use a digital signature scheme to verify the firmware authenticity. Message Authentication Codes (MAC) provide the same level of security but require a shared secret between the software supplier and the embedded device. Multiple devices should not use the same shared secret.

For the authentication scheme, the product needs a root of trust. This root of trust is a certificate or public key that the embedded system can use to verify the signature of the firmware.

The authentication scheme typically checks the firmware integrity, too.

Finally, bootcode must verify the age of the software. The bootcode stores the version number of the firmware at a secure place where it can not be manipulated. Then the bootcode will not install or boot a firmware with a lower version number.

1.5.1 Secure Update

With a Secure Update scheme the bootcode verifies the software once after receiving it in a trusted storage (internal Flash) and then use it on subsequent boots.

The update process's challenges are that the entire firmware often does not fit in RAM, and that the attacker could cut the power, preventing security checks from running.

1.5.2 Secure Boot

With a Secure Boot scheme the bootcode loads and verifies the software on every reset. This allows us to store the software on an insecure media (external Flash, SDCard, network server, etc.), which makes updates simpler and the hardware cheaper to produce. It does, however, use more RAM and spend more time on each boot.

The bootcode still need some trusted storage for the firmware loader and the root of trust (the cryptographic key to verify the firmware).

After reset, the bootcode loads the firmware into RAM, checks authenticity, integrity and version, decrypts it and executes it from RAM.

If your application contains a security vulnerability which allows an attacker to modify the bootcode, they can take complete control of the device. There are various forms of hardware support which can protect against such an attack:

- One Time Programmable (OTP) boot records pin the checksum of the bootcode or trust anchor.
- Hardware Security Modules (HSMs) are dedicated coprocessors with elevated rights and tamper resistant storage.
- Cryptographic coprocessors with key stores allow using a key (for decryption or verification) while preventing any system part from reading or changing it.

To leverage such hardware support, the bootcode is designed typically with a three stage boot process:

1. The hardware verifies the Boot Manager and Root of Trust
2. The Boot Manager verifies and executes the Boot Loader.
3. The Boot Loader loads, verifies, decrypts and executes the application.

The split of boot loader and boot manager is made, so the boot manager is as simple as possible. Bugs are unlikely and it will only need to be updated under rare circumstances. The boot loader contains all the complexity of device drivers and network protocols, so bugs are more likely. Because the hardware is not involved, updating the boot loader is easier.

2 Conclusion

We started our travel into the security of Embedded Systems by thinking about the question: “What is security, and do we need security in Embedded Systems?”. Further on, we have taken a rough view of the development process of Embedded Systems by highlighting five aspects that improve the security of the resulting product:

1. **Thread Modeling**
2. **Secure Components**
3. **Secure Deployment**
4. **Active Maintenance** and
5. **Secure Update and Boot**

Since each step can fill whole books, the focus of this paper is *Secure Components* to engage with the interested Embedded Software developer. Finally, we append a small exemplary list of threats to help your following discussion of why security is significant for upcoming developments in Embedded Systems.

3 Appendix - Threat Awareness

Today, companies rarely train their embedded software developers in IT security. Unsurprisingly, these developers have difficulty imagining the efforts the attackers now routinely take to detect and exploit weaknesses in a system. This missing knowledge leads them to underestimate the impact that even seemingly trivial bugs can have. The discussed attacks below are not a complete checklist of attacks to protect from. Instead, the intention is to raise security awareness with these examples and provide a helpful list as an argumentative aid. When talking with coworkers, managers, or customers who can not imagine how an attacker could exploit your product, these examples might help.

3.1 Message Forgery

An attacker with access to the communication channel (like Ethernet, CAN, SPI, UART, …) forges a valid message.

For example: in an electronic door lock system with a fingerprint reader, the attacker sends an “open the door” message to the door lock instead of the fingerprint reader.

You prevent this attack by authenticating all messages. For example, the door lock checks an electronic signature or message authentication code to ensure that the received message comes from the fingerprint reader.

3.2 Replay Attack

The attacker records a valid message during an authorized fingerprint and injects that message at a later point in time to open the door while none is looking.

You may try to prevent this attack with a simple message counter that constantly increases. The idea is that the door lock detects the forged message if you receive a message counter with the same or lower value than a previous one.

If the attacker can prevent the original message from reaching the receiver, the counter method no longer works. Instead, each transmission could include a timestamp, and all systems need to have synchronized clocks.

3.3 Denial of Service Through Division by Zero

Imagine a protocol definition with an integer field where the possible values exclude zero.

The attacker sends a zero in such a protocol message.

If you do not check the received value and use it in a division, the CPU triggers an exception. Then, the system enters a safe state and stops responding.

3.4 Information Loss Through Buffer Over-Read

In 2014 the Heartbleed bug (CVE-2014-0160) was found. To exploit it, an attacker sets a length field to a higher number than the protocol allows in this situation.

A missing check in faulty OpenSSL implementations leads to reading and sending the attacker more bytes from a buffer on the heap than the buffer was large. This additional data often leaked secret keys in other objects on the heap to the attacker.

3.5 Remote Code Execution Through Buffer Overflow

The CPU stores the return address of a function call on the stack and works with a descending stack (like ARM or x86).

Imagine a function is called and allocates a fixed-sized communication buffer on the stack. The address of the communication buffer is always lower than that of the stored return address.

A faulty protocol implementation copies an incoming message to that buffer without validating the length of the message. As a result, if the message length is larger than the allocated buffer, the return address gets overwritten with data from the message content.

The overwritten return address is loaded from the stack and executed when the function returns. Then, through guesswork or analyzing the firmware, the attacker causes the CPU to execute the instructions he sent.

3.6 Timing Attack Against RSA Implementation

Without details of the RSA encryption scheme, there is an easy-to-understand attack against poor implementations of the private key operation (generating a signature or decrypting a message). At its core, there is an algorithm step where we raise the message data to the power of the private key:

```
m^k
```

(there are a lot of modulo operations that do not matter here)

The following algorithm efficiently implements this exponentiation:


```
x = 1
for every bit in k:
    x = x * x
    if bit:
        x = x * m
```

The time that this operation takes to complete is directly proportional to the number of bits equal to 1 in the private key. An attacker could use this information when searching for the key.

3.7 Deliberate Power Off

An insecure update mechanism receives an update file, stores it in flash memory, and validates the signature. If the signature is invalid, the update mechanism erases the flash memory again.

An attacker could cut off the power supply while the verification checks the signature, before the update mechanism erases the flash memory. Then, on the next boot, the insecure software is executed.

3.8 Abuse Non-Standard Encoding

In unicode, the *forward-slash* (`/`) has the code point U+002F. In UTF-8 it should be represented as the single byte `0x2F`. A naive UTF-8 decoder might also decode the two-byte sequence `0xC0 0xAF` as the same forward slash. This is called overlong encoding and decoders should not accept such a byte sequence according to RFC 3629 (released in 2003, there were no such requirements in RFC 2269).

Imagine a system that serves some files from a file system. Only files from one directory may be served. The file system expects filenames to be encoded using UTF-8. In an attempt to reduce computation time and software complexity, it does not check for overlong encoding.

To enforce that all requested files come from a specific directory, the communication endpoint rejects all requests where the file name contains a *forward-slash*. It does so by scanning for the byte `0x2F`. Any request which does not contain this byte is accepted.

Because the communication endpoint and the file system do not precisely agree on the API contract, an attacker can use the byte sequence '0xC0 0xAF' to sneak a slash past the protocol validation into the file system and access any file on the filesystem.

A similar bug (CVE-2008-2938) was found in Apache Tomcat.

4 Embedded Office Profile

Embedded Office was founded in 2003 in Germany, focusing on functional safety and industrial security consulting. The engineers are certified experts in embedded software development for safety-critical systems. The unique Safety Mentoring service guarantees the success of software approvals for use in products according to IEC 61508, ISO 26262, IEC 62304, EN 50128, or DO-178C. Furthermore, provided software components like the "*Flexible Safety RTOS*" and complete safety platform integrations are completing the service to minimize efforts for new projects in the safety and security domain.

4.1 About the Author

Matthias Riegel is one of the safety and security experts at Embedded Office. Since 2015 he has been working on many safety and security related projects for our customers. In 2018 Matthias was certified as "Cyber Security Specialist" by TÜV Rheinland.

4.2 Contact

Embedded Office GmbH & Co. KG
Friedrich-Ebert-Str. 20/1
88239 Wangen
Germany

Website: https://www.embedded-office.com

