Cybersecurity Primer for IoT/Embedded Devices

September 22, 2021 - Version 3.0

timesys

Cybersecurity Primer for IoT/Embedded Devices

© 2021 Timesys Corporation
Corporate Headquarters
1905 Boulevard of the Allies
Pittsburgh, PA 15219
UNITED STATES
T: +1.412.232.3250
Toll-free: 1.866.392.4897
www.timesys.com

For more information about Timesys
and cybersecurity for embedded open source systems,
please visit us at www.timesys.com/security.

Contents

With the rise in the number of cybersecurity breaches combined with driving factors like cybersecurity laws/regulations, industry compliance, and customer cybersecurity requirements, developing a forward looking strategy to keep an IoT device secure throughout its life cycle is a challenging task. This guide provides an overview of an IoT device security lifecycle and highlights all the considerations in securing and maintaining IoT devices (Figure 1).

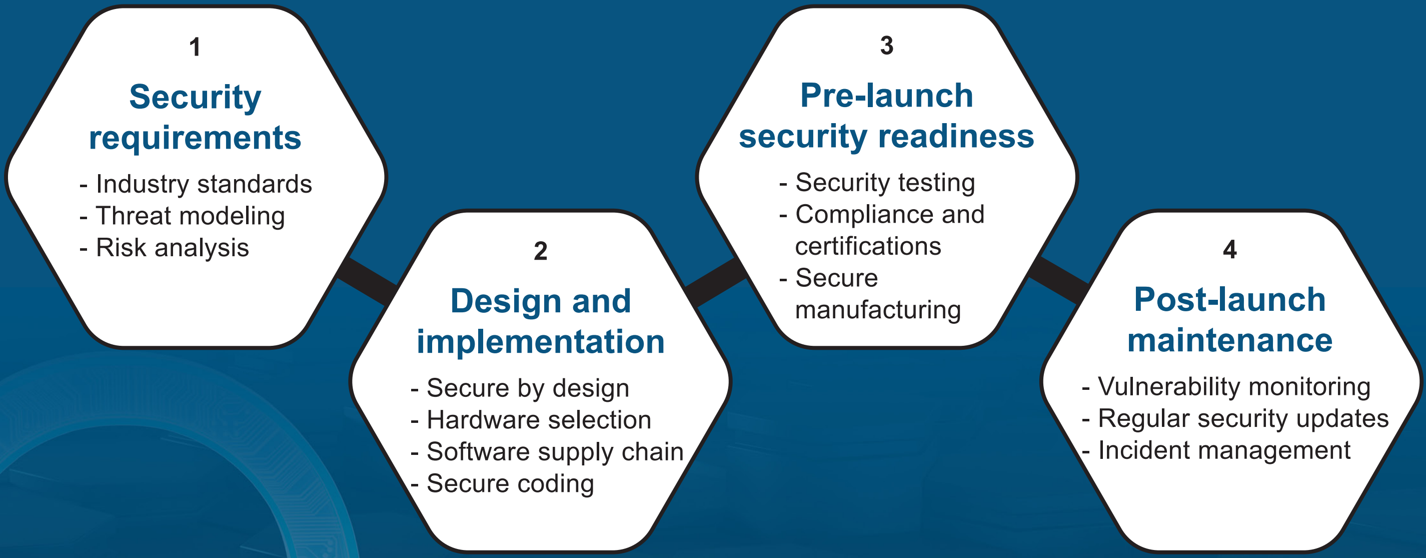


Figure 1: Device Security Lifecycle Overview

1. Security requirements

Device security requirements (Figure 2) are typically derived based on industry standards/best practices, applicable laws/regulations and by evaluating the threats in the intended operating environment of the device. Since no device is going to be 100% secure, one needs to weigh the risk of not addressing all threats with the cost of implementing cybersecurity controls to mitigate the threats.

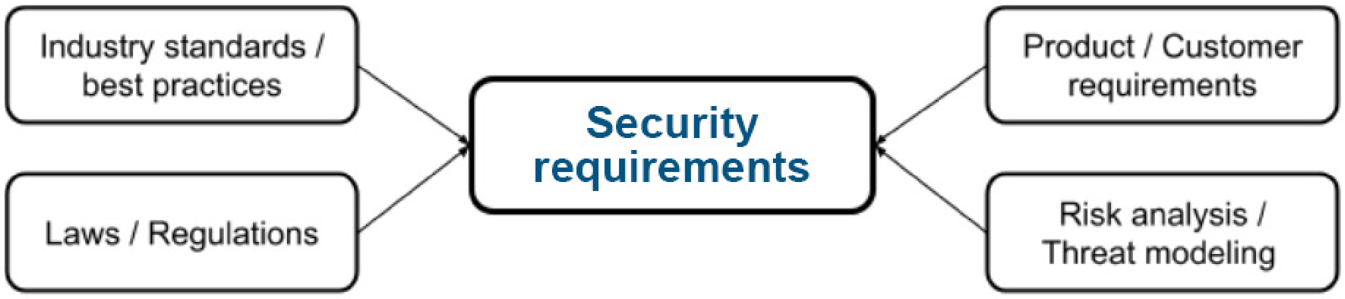

Industry standards / best practices
Laws / Regulations
Security requirements
Product / Customer requirements
Risk analysis / Threat modeling

Figure 2: Cybersecurity requirement input

1.1. Industry standards and best practices

Depending on the device, in recent years most industry sectors have established cybersecurity guidelines for device manufacturers. This can act as a baseline for device cybersecurity requirements. Below is a sample list of industry standards/guidelines for cybersecurity (Table 1). While at first glance the list looks overwhelming, the commonality between each of the standards is very high, even across industries. We will cover the commonalities between standards in later sections of this guide.

IoT / Consumer Electronics	Medical	Industrial	Automotive
• NIST 8259 • ETSI EN 303 645 • IoTSF • SESIP • ARM PSA • CSA IoT Security Controls	• FDA-2018-D-3443 (Premarket) • FDA-2015-D-5105 (Postmarket) • IEC 62304 • NEMA MDS • MDCG 2019-16	• IEC 62443 • NIST SP 800-82 • NERC CIP • ISA99	• UNECE WP.29 • ISO/SAE 21434 • NHTSA Vehicle Cybersecurity Best Practices

Table 1: Security standards across industries

1.2. Laws and regulations

In addition to industry standards, various laws have recently been passed to improve the cybersecurity of devices (Table 2). Some of the laws are enforced when manufacturers want to sell devices to the federal government, while some laws are applicable to manufacturers selling devices to consumers, and still others are a voluntary set of recommended baseline measures for device manufacturers. This is an additional consideration when developing your cybersecurity requirements.

Americas	EMEA	APAC
• H.R.1668: IoT Cybersecurity Improvement Act • California SB-327 • Oregon HB 2395 (2019)	• European Cyber Security Act	• Singapore CLS (Cybersecurity Labelling Scheme) • Australia Code of Practice

Table 2: Prominent IoT legislations

1.3. Threat modeling

While the industry standards and best practices act as a generic guideline to developing security requirements, one needs to consider requirements from the context of the product. This is done as part of the threat modeling exercise (Figure 3), where we consider:

- **Assets:** Listing the assets to be protected (e.g. intellectual property, customer data, etc.) — What is the impact of not protecting the asset?
- **Threats:** Identifying threats in the context of the operating environment (e.g. attack vectors, input/output data flow, etc.) — What is the likelihood of the threat? Who are we trying to protect against?
- **Vulnerabilities:** Identifying the weaknesses in the system and existing countermeasures if any.
- **Risk:** Assessing the risk based on the consequence of not protecting assets, likelihood of threat, and existing safeguards.
- **Priority:** Once the risk is assessed and cost of mitigation is evaluated, prioritize additional mitigations.

There are various methods available for threat modeling such as STRIDE, DREAD, PASTA, OCTAVE, CVSS, etc; any one of them can help with the above aspects.

Further reading: CMU threat modeling blog.

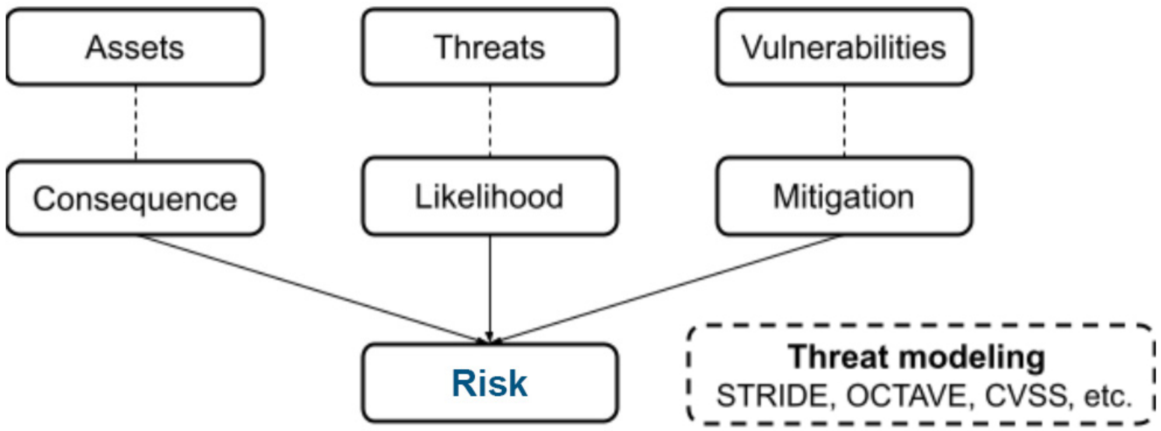


Figure 3: Threat modeling overview

1.4. Product and customer requirements

Some companies have developed their own internal cybersecurity guidelines that the product teams need to follow. These internal requirements are typically developed over time leveraging various standards, regulations, security best practices, and lessons learned. They also incorporate requirements flowing down from their customers via request for quote (RFQ), audits, results of penetration testing of their devices in the field, etc.

Device security requirements are typically derived based on industry standards/best practices, applicable laws/regulations and by evaluating the threats in the intended operating environment of the device. Since no device is going to be 100% secure, one needs to weigh the risk of not addressing all threats with the cost of implementing cybersecurity controls to mitigate the threats.

2. Design and implementation

Once the requirements are flushed out, the next step is to implement the cybersecurity controls and countermeasures (defenses against the threats). The below sections cover designing security for both hardware and software using best-in-class techniques.

2.1. Hardware requirements

Two of the key IoT device security requirements are software integrity/authenticity and data confidentiality. Implementing these requirements is not possible without processor/System on Chip (SoC) support. So the first step when designing a product is to ensure the processor that is being chosen supports certain security features such as:

- Secure boot
    - Customer programmable keys
    - Key revocation support
    - Easy access to code signing tools and detailed security documentation
- Secure key storage
- Secure debug options
- Hardware acceleration for cryptographic operations
- Tamper detection
- Trusted Execution Environment
- Secure memory / on-the-fly external bus encryption
- Hardware random number generator

While security features built in to modern processors can support common product security requirements, sometimes having additional off-chip components such as Trusted Platform Module (TPM) or Secure Elements can help ease the implementation of certificate management and device identity.

2.2. Security by Design

Secure by design refers to designing products to be foundationally secure. Typically it also incorporates the concept of defense in depth, i.e. having multiple layers of defense such that the breach of any single measure does not compromise the whole system. Below is the list of common security requirements leveraging various industry standards and recommended best practices for IoT device security (Table 3). Each of these requirements has a corresponding design solution and an example implementation on a Linux-based IoT device.

Now, let's explore some of the above requirements in more detail:

Security requirement	Solution	Example implementation (embedded Linux device)
Software integrity, authenticity	Secured boot, Chain of trust	Signed bootloader, Signed kernel/dtb/ramfs (FIT), Signed rootfs (dm-verity)
Data confidentiality (data at rest)	Encrypted storage	dm-crypt, fs-crypt
Data confidentiality (data in transit)	Secure communication	Secure protocols (TLS, SSH etc.) and link layer security (WPA etc.)
Software isolation	Sandboxing (Hardware/ Software)	Trusted Execution Environment (ARM TrustZone, OP-TEE), Containers (Docker, LXC, systemd-nspawn), Application Sand-box (AppImage, Flatpak)
Device identity	Hardware ID, Certificates	Store certificates in integrity protected / authenticated file system (dm-verity) or OP-TEE backed filesystem
Unique passwords	Password best practices	Linux Pluggable Authentication Modules (PAM)
Secure software update	Authenticated/Encrypted images with rollback protection	Popular OTA software (SWUpdate, OSTree, RAUC, Mender) all include security options
Reduce attack surface	Hardening	• Disable unused ports, services/ features, weak protocols/ciphers • Enable kernel protection options • Enable compiler protection options (-fstack-protector) • Run services as non-root
Prevent unauthorized use	Access control + Principles of least privilege	• Discretionary Access Control (DAC) — file permissions • Mandatory Access Control (MAC) — SELinux, AppArmor
Resilient to outages	Firewalls	iptables
Detect cybersecurity events	Security event logging	Auditd, go-audit
	Active defense (Behavioral based on Machine Learning)	AWS IoT Device Defender, Azure IoT Edge Defender agents
Contain cybersecurity events	Key revocation, tamper protect response	N/A (hardware specific)

Table 3: Summary of Secure by Design controls

Software integrity, authenticity:

The goal of this requirement is to ensure the integrity and authenticity of the software before executing the software. This is achieved by digitally signing each piece of software and verifying the signature on the device before executing that piece of software. This process is referred to as “Secure boot” and “Chain of Trust.” Typically on MPU based devices running embedded Linux, it starts with the processor ROM code verifying the signed bootloader, which in turn verifies a signed Linux kernel which then extends the verification to the filesystem. In case the signature verification process fails, the device stops booting and can signal a security breach.

Further reading: Secure boot and Chain of Trust paper

Data confidentiality:

The goal of this requirement is to ensure the protection of any secrets (e.g. customer data) and additionally to achieve anti-clone / anti-counterfeit / IP protection functionality. Data confidentiality is achieved by encrypting data using a key. In typical desktop computers, the user entered password is used indirectly to derive/protect the key used for encryption. However, on standalone embedded devices, this is not an option. Hence a hardware-backed secure storage mechanism is required. Most processors support this by allowing programming of keys to a secure non-volatile area, or by using a unique secure key built-in to the processor to protect the device key.

Further reading: Encrypted storage blog

Secure communications:

The goal of this requirement is to secure all external communication to/from the device. This is achieved using encryption and authentication. For IoT devices communicating over a network, the authentication is typically done using pre-installed trusted certificates, which in turn is used to share temporary session keys used for encrypting the communication with external devices. Various open source software (e.g. mbedTLS, openSSL, etc.) implement secure protocols such as TLS which can be used to secure external communication. Additionally, it is recommended that data link layer security be enabled where possible (e.g. Bluetooth, WiFi via WPA, etc.).

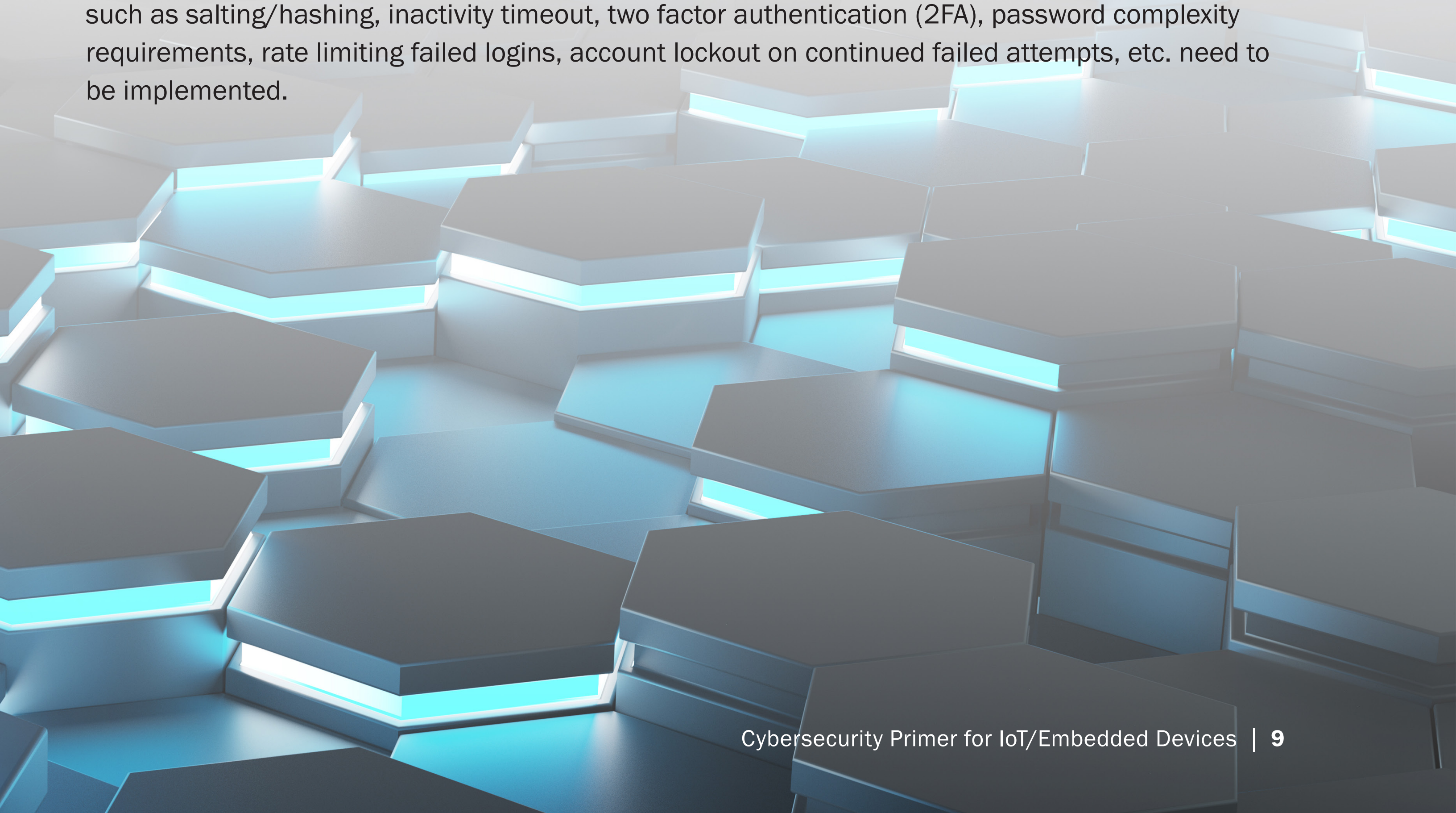
Software isolation:

The goal of this requirement is to limit the access in case of a security breach. On Linux based devices, there are multiple ways of achieving software isolation. Containers and application sandboxes are two typical ways to isolate software applications. In order to truly isolate an application, a hardware supported Trusted Execution Environment can be used to run a secure operating system such as OP-TEE in conjunction with Linux. The secure OS can then be used to run trusted applications in a secure environment.

Further reading: Trusted software development using OP-TEE

Unique passwords:

One of the most common attacks on IoT devices is hijacking the device using known default global passwords. For this reason, any passwords used on the device need to be unique such that any password leak is self contained to that device only. Additionally, password security best practices such as salting/hashing, inactivity timeout, two factor authentication (2FA), password complexity requirements, rate limiting failed logins, account lockout on continued failed attempts, etc. need to be implemented.



Secure software update:

Having a mechanism to update software in the field is a must-have for addressing any vulnerabilities after launch of the device. Equally important is the security of the update mechanism. Some considerations when implementing software updates are:

- Authentication of the device and the download server using certificates
- Download of the images using secure protocols (e.g. HTTPS/TLS)
- Verifying the signature of the image bundle using public key cryptography
- Encrypted image bundle with a shared key
- Verify no unauthorized rollback of images (anti-roll back)
- Images additionally checked as part of secure boot and chain of trust

Further reading: Secure software update webinar

Reduce attack surface:

The purpose of hardening is to reduce the attack surface and make the device more difficult to hack.
Best practices in device hardening include:

- Protecting or disabling debug hardware ports (JTAG / Serial, etc.)
- Enabling kernel and compiler protection features (Further information: Linux kernel hardening webinar)
- Disabling unused services, packages, features
- Disabling configurations that are known to be exploitable

Hardening can also entail physical hardware security. For example, off-chip components like TPM are prone to bus attacks (e.g. probing the I2C bus using an analyzer/scope). One mitigation option is to cover the TPM chip with an RF shield and under-fill it with epoxy resin (check with your country's right to repair bill for legality), making it difficult to get to the I2C pins. Additionally, burying the I2C traces on the PCB and overlaying it with a tamper-detect trace can ensure that a tamper-detect response can power down the board, rendering the attack infeasible.

2.3. Secure coding practices

While this guide mostly focuses on device/OS security principles, security of your proprietary applications is an equally critical aspect of device security. Coding guidelines need to include secure coding practices (References: OWASP and CERT) and code reviews need to hold software developers accountable.

Additionally, awareness of the most critical or dangerous software weakness as listed in OWASP Top Ten and Mitre's CWE Top 25 should help promote defensive coding. Section 3.1 below covers the tools that developers can leverage to build secure applications.

2.4. Software supply chain security

Software supply chain security (Figure 4) has gotten a lot of attention recently with the SolarWinds attack. As downstream consumers of open source projects, it is important to verify the provenance of the source code being downloaded to reduce the risk of supply chain attacks. Some upstream projects provide GPG/PGP signatures which can be used to verify the authenticity of software. Signatures can be verified using git signed tags/commits (e.g. Yocto release PGP signature) or on the downloaded source package (e.g. Linux kernel release PGP signature). Additionally it is recommended that any open source software that is being included must follow a criteria to be deemed as safe. The Open Source Security Foundation has a project called "Scorecards" that can be used to judge if an open source project is safe to use.

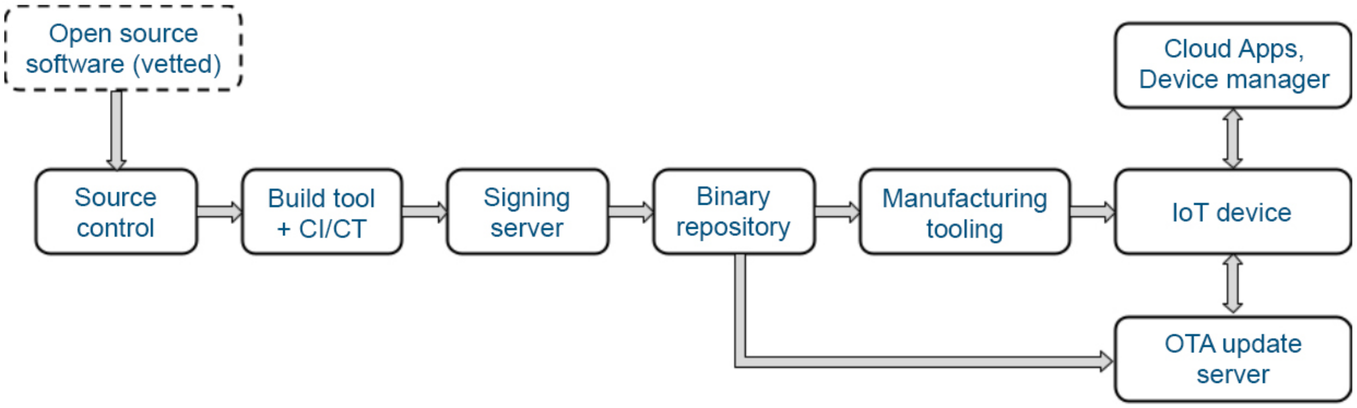


Figure 4: Software supply chain security components

Another consideration from a security maintenance perspective is to pick open source software with a committed long term support (LTS) roadmap, i.e. bug fixes and security fixes are provided for a given version of the software for a extended duration of time. This applies to all pieces of software, including operating systems (Linux kernel, Zephyr, FreeRTOS, etc.), user space libraries (OpenSSL, Python, etc.), and Build Systems (Yocto, Buildroot, etc.); each of these provide LTS versions.

Once an open source project is vetted, the rest of the internal source/build/signing/manufacturing and cloud infrastructure needs to be secured. There are multiple industry standards and best practices for securing the IT infrastructure; for example, ISO 27001 can be used as a guideline. Additional process steps such as two-person reviews, hermetic builds, and reproducible builds can be incorporated as per the Software supply chain integrity SLSA guidelines published by Google. Lastly, for security practices to excel within an organization, a culture of security needs to be promoted along with adequate training.

3. Pre-launch security readiness

In order to prepare for launching the product, the following needs to be considered from a security perspective:

3.1. Security Testing

Security test plan:

Similar to having a test plan for software functionality, there needs to be a test plan for testing cybersecurity controls and countermeasures to ensure the security requirements are met. The emphasis of this test plan would be around negative test cases (e.g. unsigned/tampered images do not boot, expired certificates are rejected, verifying development keys are removed, etc.).

Security testing tools:

There are a multitude of tools that can report software security weaknesses and vulnerabilities. These are typically run by the development team, and sometimes in conjunction with the security team, to identify coding issues, configuration issues, and vulnerabilities in 3rd party software. The table below (Table 4) provides commonly used tools that can act as a starting point for securing your software.

Tool category	What it does	Example tools
Static Application Security Testing (SAST)	Analyzes source code and reports security weaknesses (e.g. Use After Free, NULL Pointer Dereference, etc.)	Coverity, SonarQube, cppcheck, Infer
Web Application security	Tests a web application in its operating state and reports security weaknesses	OWASP ZAP, Burp Suite
Fuzzing tools	Inputs massive amounts of random data (fuzz) to the application in an attempt to make it crash, and in turn to find security loopholes	AFL++, Syzkaller, OSS-fuzz
Audit and compliance tools	Audits the target device against security best practices, which can then be used to harden the device	Lynis, OpenSCAP
Software Composition Analysis (SCA)	Generates a Software Bill of Materials (SBOM) and reports known vulnerabil-ities in the software, sometimes also providing information regarding patches, mitigations, exploits, etc.	Vigiles, BlackDuck, Open-VAS

Table 4: Summary of popular security tools

Penetration testing:

The goal of penetration testing, sometimes referred to as pen testing, is to simulate a cyberattack against the device and check for exploitable vulnerabilities. While some of the above tools do some level of penetration testing, it lacks the knowledge of the system as a whole to devise advanced attacks typically carried out by hackers. So having a dedicated internal pen testing team, or hiring an outside firm, are commonly employed strategies. Additionally, if hardware hacks are of concern, there are companies that perform hardware pen testing as well (e.g. side channel attacks via fault injection, differential power analysis, power glitches, etc).

3.2. Certifications/Compliance

Some regulated industries might require submission of documents related to cybersecurity with test results (self-certified) or get an external certification or 3rd-party audit. Here are a list of common external certifications and labs for IoT devices:

Certifications: SESIP, Common Criteria, UL 2900-2-1 (FDA recognized), CTIA IoT Security, PSA Certified

Labs: Riscure, Brightsight, UL, Serma Safety Security, Applus Laboratories, ECSEC labs, CAICT

3.3. Secure Manufacturing

As part of the product launch, safeguards must be put in place for protecting any device secrets at the time of manufacturing. Any required tooling to securely program the devices, install device certificates, and provision devices must be addressed. Some processors additionally support mechanisms to simplify contract manufacturing.

Further reading: [NXP i.MX manufacturing protect app note]

Similar to having a test plan for software functionality, there needs to be a test plan for testing cybersecurity controls and countermeasures to ensure the security requirements are met.

4. Post-launch maintenance

Device security does not stop at securing the product at the design phase. Maintaining the security of the device through its support period (typically 10-15 years for IIoT) is equally important. Below are some of the processes that need to be in place to achieve the same.

4.1. Vulnerability monitoring and periodic security updates

A key aspect of cybersecurity is the long term security maintenance of the device. Most devices use open source software which can contain known vulnerabilities, a.k.a. Common Vulnerabilities and Exposures (CVE). In order to monitor known vulnerabilities in 3rd-party software, one needs to generate a Software Bill of Materials (SBOM) containing a list of software components and associated versions used in the product. Using the SBOM, the list of publicly known vulnerabilities can be obtained from sources like National Vulnerability Database (NVD), security issue trackers, and bulletins. With more than 300 new vulnerabilities being reported each week, manual monitoring is an inefficient and error prone process. Various tools are available to automate the process. Some of the key features to check for are:

Various tools are available to automate the process. Some of the key features to check for are:

- Automatic accurate SBOM generation (Optimized for embedded: plugs into build systems such as Yocto or Buildroot)
- Good vulnerability data (multiple sources, very few false positives, timely reporting)
- Intelligent filtering (based on enabled configurations: Linux kernel / U-Boot config; attack vectors, severity, etc.)
- Identifies availability of fixes/mitigations/exploits
- Supports your software development lifecycle (SDLC) workflow (CI/CD, Jira integration, automatic vulnerability release note generation, etc.)

Further reading: Evaluating vulnerability monitoring tools for embedded systems

Once the vulnerabilities are identified, the next step is to analyze the applicability, the exposure/risk, and then prioritize appropriately. This process is commonly referred to as vulnerability triage. To remediate the prioritized vulnerabilities in a timely manner, a release cadence policy needs to be established.

Further reading: Vulnerability management and triaging

See Figure 5 for the full vulnerability management process workflow.

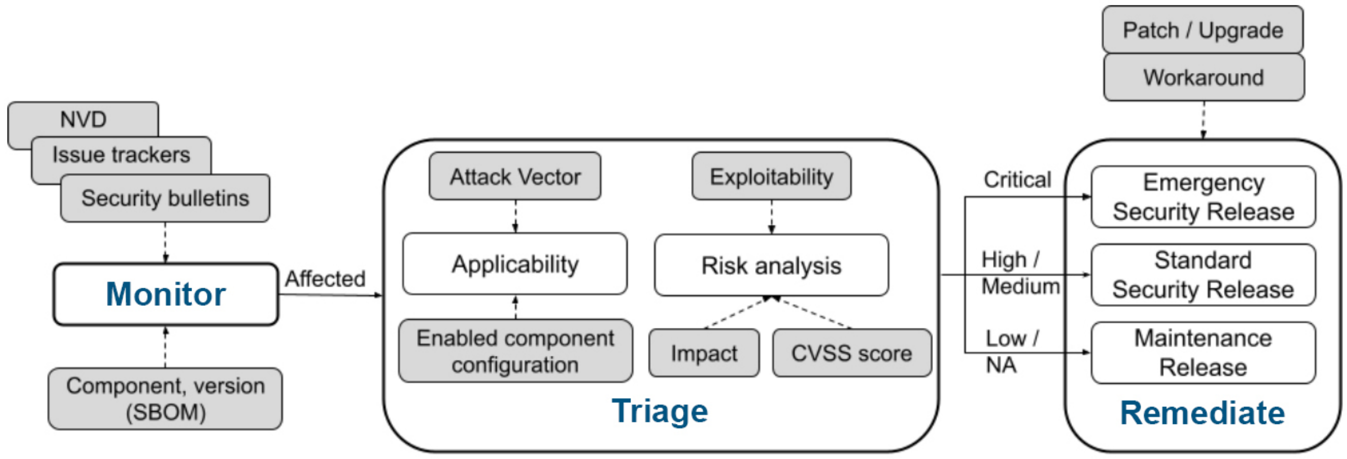


Figure 5: Risk-based vulnerability management process workflow

Once a fix for the vulnerability is available, in order to incorporate the fix into the device, the product team needs to decide between one of two methods: updating to a newer version of the software package that incorporates the fix, or selectively backporting that specific security fix into the version of the software package being used by the device. Updating to the latest version of the software is typically recommended from a security perspective. However, it may not always be practical: there could be API changes necessitating an update of custom software, resulting in a long test cycle. Our recommendation for open source software packages that support LTS releases is to update to the latest minor LTS version, as APIs are not expected to change between minor LTS releases. For the rest of the software, it's a decision to be made by the product team on a case by case basis. Once the updated product software incorporating the fix is validated and released, the devices in the field need to be updated with the latest software in a timely manner to keep the exploitable window to a minimum.

Further reading: **The many challenges of Linux OS security maintenance**

4.2. Vulnerability reporting and disclosure program

Security researchers and end customers might uncover security vulnerabilities in the product. These are typically vulnerabilities in your product/proprietary application software and may not directly relate to 3rd party software used in the product. In order for researchers and customers to securely and responsibly report their findings, companies should publish vulnerability reporting and disclosure information on their website. Information regarding the expected timelines for acknowledgement and status updates are also typically published. Finally, a policy needs to be in place to publish all valid vulnerabilities and notify end users, CVE Numbering Authorities (**CNA**), and/or concerned authorities.

Additionally, a bug bounty program can be established to encourage researchers to find and report security issues before the hackers find their way into the device. Platforms such as HackerOne and BugCrowd are popular to establish these programs.

Further reading: OWASP vulnerability disclosure **cheat sheet**

4.3. Incident monitoring and management

Most IoT devices follow an enrolling/provisioning process to connect to the cloud, after which they are made available in a cloud IoT device management platform. The device health can then be monitored using the platform to audit/detect any potential breaches. Cybersecurity incident management deals with responding to security breaches. One needs to plan for:

Detection: Implementing a process to monitor for potential breaches either via audit logs or automation of anomaly detection

Triage/Analysis: Verifying the breach is real and prioritizing the response based on risk/impact

Communication: Notifying all concerned parties of the breach and plan of action

Containment and Eradication: Containing/isolating the threat (where possible) and then starting the recovery/eradication process

Post-incident analysis: Compiling lessons learned and improving the security posture to prevent future incidents

Additionally, the security of the cloud platform itself needs to be monitored and audited as per the cloud security best practices (e.g.: CAIQ).

Further reading: NIST SP 800-61

4.4. Decommissioning

Protecting the privacy of customer data needs to be planned for in case of field returns, transfer of device ownership, or decommissioning. This is typically handled by creating a publicly documented procedure to return the device to factory default state while securely erasing any customer data.

Device security does not stop at securing the product at the design phase. Maintaining the security of the device through its support period (typically 10-15 years for IIoT) is equally important.

5. Takeaway

Navigating the maze of industry cybersecurity standards, implementing the necessary countermeasures, and maintaining IoT product security throughout its lifecycle is a complex process. It takes tremendous planning, a large investment of resources, and dedicated cybersecurity expertise. Nevertheless, implementing security features early in device development is key to maintaining strong security in the field. It is recommended to staff development teams up front to account for cybersecurity tasks, or to offload these tasks to a third party security expert. This allows the core application team to focus on the value-added product software. Additionally, investing in the right cybersecurity tools, along with test automation, can reduce costs while improving the overall security of your product. The combination of understanding security requirements, implementing security by design, security testing, and vulnerability monitoring with periodic security updates gives your device the best possible security posture.

Learn more about how [Timesys Device Security Solutions]() can help jump start your device security journey, with best in-class tools and services for the full lifecycle of your device.