Final Security Good Practices

Architeture	IoT System
Application domain type	Smart Home
Authentication	Username and Password
Has DB	Yes
Type of data storage	SQL
Which DB	MySQL
Type of data stored	Personal Information ; Critical Data
User Registration	Yes
Type of Registration	Will be a administrator that will register the users
Programming Languages	C/C++
Input Forms	Yes
Upload Files	No
The system has logs	Yes
The system has regular updates	No
The system has third-party	No
System Cloud Environments	Public Cloud
Hardware Specification	Yes
HW Authentication	TPM (Trusted Platform Module)
HW Wireless Tech	Wi-Fi ; Bluetooth
Data Center Phisical Access	Yes

IoT Security

Internet of Things (IoT) devices fall into three main categories:

- Sensors, which gather data
- Actuators, which effect actions
- Gateways, which act as communication hubs and may also implement some automation logic.

All these device types may stand alone or be embedded in a larger product. They may also be complemented by a web application or mobile device app and cloud based service. IoT devices, services and software, and the communication channels that connect them, are at risk of attack by a variety of malicious parties,

Malicious intent commonly takes advantage of poor design, but even unintentional leakage of data due to ineffective security controls can also bring dire consequences to consumers and vendors. Thus it is vital that IoT devices and services have security designed in from the outset.

Classification of Data

- Define a data classification scheme and document it.
- Assess every item of data stored, processed, transmitted or received by a device and apply a data classification rating to it.
- Ensure the security design protects every data item and collections of items against unauthorised viewing, changing or deletion,to at least its classification rating or higher.

Physical Security

- Any interface used for administrationor test purposes during development should be removed from a production device, disabled or made physically inaccessible.
- All test access points on production units must be disabled or locked, for example by blowing on-chip fuses to disable JTAG.
- If a production device must have an administration port, ensure it has effective access controls, e.g. strong credential management, restricted ports, secureprotocols etc.
- Make the device circuitry physically inaccessible to tampering, e.g. epoxy chips to circuit board, resin encapsulation, hiding data and address lines under thesecomponents etc.
- Provide secure protective casing and mounting options for deployment of devices in exposed locations.
- For high-security deployments, consider design measures such as active masking or shielding to protect against side-channel attacks

Device Secure Boot

- Make sure the ROM-based secure boot function is always used. Use a multi-stage bootloader initiated by a minimal amount of read-only code
- Use a hardware-based tamper-resistant capability (e.g. a microcontroller security subsystem, Secure Access Module (SAM) or Trusted Platform Module (TPM)) to store crucial data items and run the trusted authentication/cryptographic functions required for the boot process. Its limited secure storage capacity must hold the read-only first stage of the bootloader and all other data required to verify the authenticity of firmware.
- Check each stage of boot code is valid and trusted immediately before running that code. Validating code immediately before its use can reduce the risk of attacks
- At each stage of the boot sequence, wherever possible, check that only the expected hardware is present and matches the stage's configuration parameters.
- Do not boot the next stage of device functionality until the previous stage has been successfully booted.

- Ensure failures at any stage of the boot sequence fail gracefully into a secure state, to ensure no unauthorised access is gained to underlying systems, code or data. Any code run must have been previously authenticated.

Secure Operating System

- Include in the operating system (OS) only those components (libraries, modules, packages etc.) that are required to support the functions of the device.
- Shipment should include the latest stable OS component versions available.
- Devices should be designed and shipped with the most secure configuration in place.
- Continue to update OS components to the latest stable versions throughout the lifetime of a deployed device.
- Disable all ports, protocols and services that are not used.
- Set permissions so users/applications cannot write to the root file system.
- If required, accounts for ordinary users/applications must have minimum access rights to perform the necessary functions. Separate administrator accounts (if required)will have greater rights of access. Do not run anything as root unless genuinely unavoidable.
- Ensure all files and directories are given the minimum access rights to perform the required functions.
- Consider implementing an encrypted file system.

Application Security

- Applications must be operated at the lowest privilege level possible, not as root. Applications must only have access to those resources they need.
- Applications should be isolated from each other. For example, use sandboxing techniques such as virtual machines, containerisation, Secure Computing Mode (seccomp), etc
- Ensure compliance with in-country data processing regulations.
- Ensure all errors are handled gracefully and any messages produced do not reveal any sensitive information.
- Never hard-code credentials into an application. Credentials must be stored separately in secure trusted storage and must be updateable in a way that ensures security is maintained.
- Remove all default user accounts and passwords.
- Use the most recent stable version of the operating system and libraries.
- Never deploy debug versions of code. The distribution should not include compilers, files containing developer comments, sample code, etc.
- Consider the impact on the application/system if network connectivity is lost. Aim to maintain normal functionality and security wherever possible.

Credential Management

- A device should be uniquely identifiable by means of a factory-set tamper resistant hardware identifier if possible.
- Use good password management techniques, for example no blank or simple passwords allowed, never send passwords across a network (wired or wireless) in clear text, and employ a secure password reset process.
- Each password stored for authenticating credentials must use an industry standard hash function, along with a unique salt value that is not obvious (for example, not a username).
- Store credentials or encryption keys in a Secure Access Module (SAM), Trusted Platform Module (TPM), Hardware Security Module (HSM) or trusted key store if possible.
- Aim to use 2-factor authentication for accessing sensitive data if possible.
- Ensure a trusted & reliable time source is available where authentication methods require this, e.g. for digital certificates.
- A certificate used to identify a device must be unique and only used to identify that onedevice. Do not reuse the certificate across multiple devices.
- A "factory reset" function must fully remove all user data/credentials stored on a device.

Encryption

- When configuring a secure connection, if an encryption protocol offers a negotiable selection of algorithms, remove weaker options so they cannot be selected for use in a downgrade attack.
- Store encryption keys in a Secure Access Module (SAM), Trusted Platform Module (TPM), Hardware Security Module (HSM) or trusted key store if possible.
- Do not use insecure protocols, e.g. FTP, Telnet.
- It should be possible to securely replace encryption keys remotely
- If implementing public/private key cryptography, use unique keys per device and avoid using global keys. A device's private key should be generated by that device or supplied by an associated secure credential solution, e.g. smart card. It should remain on that device and never be shared/visible to elsewhere.

Network Connections

- Activate only those network interfaces that are required (wired, wireless - including Bluetooth etc.).
- Run only those services on the network that are required.
- Open up only those network ports that are required.
- Run a correctly configured software firewall on the device if possible.
- Always use secure protocols, e.g. HTTPS, SFTP.
- Never exchange credentials in clear text or over weak solutions such as HTTP Basic Authentication.
- Authenticate every incoming connection to ensure it comes from a legitimate source.
- Authenticate the destination before sending sensitive data.

Logging

- Ensure all logged data comply with prevailing data protection regulations.
- Run the logging function in its own operating system process, separate from other functions.
- Store log files in their own partition, separate from other system files.
- Set log file maximum size and rotate logs.
- Where logging capacity is limited, just log start-up and shutdown parameters, login/access attempts and anything unexpected.
- Restrict access rights to log files to the minimum required to function.
- If logging to a central repository, send log data over a secure channel if the logs carry sensitive data and/or protection against tampering of logs must be assured.
- Implement log "levels" so that lightweight logging can be the standard approach, but with the option to run more detailed logging when required.
- Monitor and analyse logs regularly to extract valuable information and insight.
- Passwords and other secret information should not ever be displayed in logs.

(https://www.iotsecurityfoundation.org/wp-content/uploads/2019/03/Best-Practice-Guides-Release-1.2.1.pdf)[https://www.iotsecurityfoundation.org/wp-content/uploads/201

Input Validation

Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components.

Implementing input validation

- Data type validators available natively in web application frameworks
- Validation against JSON Schema and XML Schema (XSD) for input in these formats.
- Type conversion (e.g. Integer.parseInt() in Java, int() in Python) with strict exception handling
- Minimum and maximum value range check for numerical parameters and dates
- Minimum and maximum length check for strings.
- Array of allowed values for small sets of string parameters (e.g. days of week).
- Regular expressions for any other structured data covering the whole input string (^...$) and not using "any character" wildcard (such as . or \S)

If it's well structured data, like dates, social security numbers, zip codes, e-mail addresses, etc. then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input. If the input field comes from a fixed set of options, like a drop down list or radio buttons, then the input needs to match exactly one of the values offered to the user in the first place. Free-form text, especially with Unicode characters, is perceived as difficult to validate due to a relatively large space of characters that need to be whitelisted. The primary means of input validation for free-form text input should be:

- Normalization: Ensure canonical encoding is used across all the text and no invalid characters are present.
- Character category whitelisting: Unicode allows whitelisting categories such as "decimal digits" or "letters" which not only covers the Latin alphabet but also various other scripts used globally (e.g. Arabic, Cyryllic, CJK ideographs etc).
- Individual character whitelisting: If you allow letters and ideographs in names and also want to allow apostrophe ' for Irish names, but don't want to allow the whole punctuation category.

Client Side vs Server Side Validation
Be aware that any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

Email Validation Basics
Many web applications do not treat email addresses correctly due to common misconceptions about what constitutes a valid address. Specifically, it is completely valid to have an mailbox address which:

- Is case sensitive in the local portion of the address (left of the rightmost @ character).
- Has non-alphanumeric characters in the local-part (including + and @).
- Has zero or more labels.

Following RFC 5321, best practice for validating an email address would be to:

- Check for presence of at least one @ symbol in the address.
- Ensure the local-part is no longer than 64 octets.
- Ensure the domain is no longer than 255 octets.
- Ensure the address is deliverable.

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Input_Validation_Cheat_Sheet.md

Cryptography

An architectural decision must be made to determine the appropriate method to protect data at rest. There are such wide varieties of products, methods and mechanisms for cryptographic storage.The general practices and required minimum key length depending on the scenario listed below:

Good practices:

- Cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG (even if they are NIST certified). All of these should be marked as insecure and should not be used and removed from the application and server.
- Key lengths are in-line with industry standards and provide protection for sufficient amount of time. A comparison of different key lengths and protection they provide taking into account Moore's law is available online.
- Cryptographic means are not mixed with each other: e.g. you do not sign with a public key, or try to reuse a keypair used for a signature to do encryption.
- Cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

Recommended Algorithms: * Confidentiality algorithms: AES-GCM-256 or ChaCha20-Poly1305; * Integrity algorithms: SHA-256, SHA-384, SHA-512, Blake2; * Digital signature algorithms: RSA (3072 bits and higher), ECDSA with NIST P-384; * Key establishment algorithms: RSA (3072 bits and higher), DH (3072 bits or higher), ECDH with NIST P-384; * Application must be capable of using end-to-end encryption via SSL / TLS in relation to sensitive data in transit and at rest.

Additionally, you should always rely on secure hardware (if available) for storing encryption keys, performing cryptographic operations, etc.

Secure Cryptographic Storage Design:

- All protocols and algorithms for authentication and secure communication should be well vetted by the cryptographic community.
- Ensure certificates are properly validated against the hostnames users whom they are meant for.
- Avoid using wildcard certificates unless there is a business need for it
- Maintain a cryptographic standard to ensure that the developer community knows about the approved ciphersuits for network security protocols, algorithms, permitted use, cryptoperiods and Key Management.
- Only store sensitive data that you need

Use strong approved Authenticated Encryption
CCM or GCM are approved Authenticated Encryption modes based on AES algorithm.

Use strong approved cryptographic algorithms

- Do not implement an existing cryptographic algorithm on your own, no matter how easy it appears. * Instead, use widely accepted algorithms and widely accepted implementations.
- Only use approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.
- Do not use weak algorithms, such as MD5 or SHA1.
- Avoid hashing for password storage,instead use Argon2, PBKDF2, bcrypt or scrypt.
- See NIST approved algorithms or ISO TR 14742 "Recommendations on Cryptographic Algorithms or Algorithms", key size and parameters by European Union Agency for Network and Information Security.
- If a password is being used to protect keys then the password strength should be sufficient for the strength of the keys it is protecting. * When 3DES is used, ensure K1 != K2 != K3, and the minimum key length must be 192 bits .
- Do not use ECB mode for encrypting lots of data (the other modes are better because they chain the blocks of data together to improve the data security).

Use strong random numbers

- Ensure that all random numbers, especially those used for cryptographic parameters (keys, IV's, MAC tags), random file names, random GUIDs, and random strings are generated in a cryptographically strong fashion.
- Ensure that random algorithms are seeded with sufficient entropy.
- Tools like NIST RNG Test tool can be used to comprehensively assess the quality of a Random Number Generator by reading e.g. 128MB of data from the RNG source and then assessing its randomness properties with the tool.

The following libraries are considered weak random numbers generators and should not be used:

C library: random(), rand(), use getrandom(2) instead
Java library: java.util.Random() instead use java.security.SecureRandom instead
For secure random number generation, refer to NIST SP 800-90A. CTR-DRBG, HASH-DRBG, HMAC-DRBG are recommended

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cryptographic_Storage_Cheat_Sheet.md

Authentication and Integrity

Introduction

This cheat sheet provides a simple model to follow when implementing transport layer protection for an application. Although the concept of SSL is known to many, the actual details and security specific decisions of implementation are often poorly understood and frequently result in insecure deployments. This article

establishes clear rules which provide guidance on securely designing and configuring transport layer security for an application. This article is focused on the use of SSL/TLS between a web application and a web browser, but we also encourage the use of SSL/TLS or other network encryption technologies, such as VPN, on back end and other non-browser based connections.

Architectural Decision

An architectural decision must be made to determine the appropriate method to protect data when it is being transmitted. The most common options available to corporations are Virtual Private Networks (VPN) or a SSL/TLS model commonly used by web applications. The selected model is determined by the business needs of the particular organization. For example, a VPN connection may be the best design for a partnership between two companies that includes mutual access to a shared server over a variety of protocols. Conversely, an Internet facing enterprise web application would likely be best served by a SSL/TLS model.

TLS is mainly a defence against man-in-the-middle attacks. An TLS Threat Model is one that starts with the question "What is the business impact of an attacker's ability to observe, intercept and manipulate the traffic between the client and the server".

This cheat sheet will focus on security considerations when the SSL/TLS model is selected. This is a frequently used model for publicly accessible web applications.

Providing Transport Layer Protection with SSL/TLS

Benefits *

The primary benefit of transport layer security is the protection of web application data from unauthorized disclosure and modification when it is transmitted between clients (web browsers) and the web application server, and between the web application server and back end and other non-browser based enterprise components.

The server validation component of TLS provides authentication of the server to the client. If configured to require client side certificates, TLS can also play a role in client authentication to the server. However, in practice client side certificates are not often used in lieu of username and password based authentication models for clients.

TLS also provides two additional benefits that are commonly overlooked; integrity guarantees and replay prevention. A TLS stream of communication contains built-in controls to prevent tampering with any portion of the encrypted data. In addition, controls are also built-in to prevent a captured stream of TLS data from being replayed at a later time.

It should be noted that TLS provides the above guarantees to data during transmission. TLS does not offer any of these security benefits to data that is at rest. Therefore appropriate security controls must be added to protect data while at rest within the application or within data stores.

Good Practices *

Use TLS, as SSL is no longer considered usable for security;

- All pages must be served over HTTPS. This includes css, scripts, images, AJAX requests, POST data and third party includes. Failure to do so creates a vector for man-in-the-middle attacks;
- Just protecting authenticated pages with HTTPS, is not enough. Once there is one request in HTTP, man-in-the-middle attacks are possible, with the attackers being able to prevent users from reaching the secured pages.

The HTTP Strict Transport Security Header must be used and pre loaded into browsers. This will instruct compatible browsers to only use HTTPS, even if requested to use HTTP. Cookies must be marked as Secure.

Basic Requirements *

Access to a Public Key Infrastructure (PKI) in order to obtain certificates;

- Access to a directory or an Online Certificate Status Protocol (OCSP) responder in order to check certificate revocation status;
- Agreement/ability to support a minimum configuration of protocol versions and protocol options for each version.

[https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.md]

File Uploading

Into web applications, when we expect upload of working documents from users, we can expose the application to submission of documents that we can categorize as malicious. We use the term "malicious" here to refer to documents that embed malicious code that will be executed when another user (admin, back office operator...) will open the document with the associated application reader.
Usually, when an application expect his user to upload a document, the application expect to receive a document for which the intended use will be for reading/printing/archiving. The document should not alter is content at opening time and should be in a final rendered state.

The most common file types used to transmit malicious code into file upload feature are the following:

- Microsoft Office document: Word/Excel/Powerpoint
- Adobe PDF document: Insert malicious code as attachment.
- Images: Malicious code embedded into the file or use of binary file with image file extension.

For Word/Excel/Powerpoint/Pdf documents:

Detect when a document contains "code"/OLE package, if it's the case then block the upload process. For Images document: Sanitize incoming image using re-writing approach and then disable/remove any "code" present (this approach also handle case in which the file sent is not an image).

Upload Verification

- Use input validation to ensure the uploaded filename uses an expected extension type
- Ensure the uploaded file is not larger than a defined maximum file size

Upload Storage

- Use a new filename to store the file on the OS. Do not use any user controlled text for this filename or for the temporary filename.
- Store all user uploaded files on a separate domain. Archives should be analyzed for malicious content (anti-malware, static analysis, etc).

Public Serving of Uploaded Content

- Ensure the image is served with the correct content-type (e.g. image/jpeg, application/x-xpinstall)

Beware of "special" files

The upload feature should be using a whitelist approach to only allow specific file types and extensions. However, it is important to be aware of the following file types that, if allowed, could result in security vulnerabilities.

"crossdomain.xml" allows cross-domain data loading in Flash, Java and Silverlight. If permitted on sites with authentication this can permit cross-domain data theft and CSRF attacks. Note this can get pretty complicated depending on the specific plugin version in question, so its best to just prohibit files named "crossdomain.xml" or "clientaccesspolicy.xml".

".htaccess" and ".htpasswd" provides server configuration options on a per-directory basis, and should not be permitted.

Application Regular Updates

Mobile devices and platforms, such as, for example, smartphones, typically provide the capability for operating system (OS), firmware (FW) and applications updates or re-installations with reduced user involvement. The user involvement may often be limited to clicking an icon or accepting an agreement. While this reduced level of involvement may provide convenience and an improved user experience, it fails to address the issue of secure user authentication.

Mobile devices and platforms, such as smartphones, typically provide features for operating system (OS), firmware (FW) upgrades, and applications or reinstallations with reduced user engagement. User engagement may be limited to clicking an icon or accepting a contract. While this reduced level of engagement can provide convenience and enhance the user experience, it does not address the issue of secure user authentication. Thus, it is necessary to create a secure channel that provides confidentiality, integrity, authentication and data updating.

Requirements for a secure software update:

Data Confidentiality: the contents of transmitted data should be kept confidential. This also includes software updates. Thus, secure channels between the mobile device and the network manager must be set up. The standard approach to keep sensitive data secret is to encrypt the data with a key that is shared only between the intended receivers;

Data integrity: it must be possible to ensure that data packets have not been modified in transit. For mobile devices, control requests, and software updates it is critically important to verify that the contents in the packets have not been tampered with;

Data Authentication: To prevent an attacker from injecting packets it is important to make sure that the receiver can verify the sender of the packets. Data authentication ensures this property such that the re- ceiver can verify that the received packets really are from the claimed sender. For example, for software updates, data authentication is needed such that the device can verify that the received software comes from a trusted source. Data authentication can be achieved using a MAC or Digital Signature;

Data Freshness: to protect against replay attacks, e.g., during the key establishment phase, the proto- col must ensure that the messages are fresh. Data freshness ensures the security property that the data is recent and that an attacker is not replaying old data.