

Final Security Good Practices

Mobile Platform	Web Application
Application domain type	m-Health
Authentication	Yes
Authentication schemes	Biometric-based authentication ; Channel-based authentication ; Factors-based authentication
Has DB	Yes
Type of data storage	Local Storage (Centralized Database)
Which DB	
Type of data stored	Confidential Data ; Critical Data
User Registration	Yes
Type of Registration	The users will register themselves
Programming Languages	Java ; HTML5
Input Forms	Yes
Upload Files	Yes
The system has logs	Yes
The system has regular updates	Yes
The system has third-party	Yes
System Cloud Environments	Public Cloud
Hardware Specification	Yes
HW Authentication	Basic Authentication (user/pass)
HW Wireless Tech	3G ; 4G/LTE ; 5G ; Bluetooth ; Wi-Fi ; GPS ; NFC
Data Center Physical Access	Yes

SQL Injection

An SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either:

- stop writing dynamic queries;
- prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

Primary Defenses:

- Option 1: Use of Prepared Statements (with Parameterized Queries)
- Option 2: Use of Stored Procedures
- Option 3: White List Input Validation
- Option 4: Escaping All User Supplied Input

Additional Defenses:

- Also: Enforcing Least Privilege
- Also: Performing White List Input Validation as a Secondary Defense

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, Cyrillic, 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

Session Management

A web session is a sequence of network HTTP request and response transactions associated to the same user. Modern and complex web applications require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables - such as access rights and localization settings - which will apply to each and every interaction a user has with the web application for the duration of the session.

Additionally, web applications will make use of sessions once the user has authenticated. This ensures the ability to identify the user on any subsequent requests as well as being able to apply security access controls, authorized access to the user private data, and to increase the usability of the application. Therefore, current web applications can provide session capabilities both pre and post authentication.

Session ID Properties

In order to keep the authenticated state and track the users progress within the web application, applications provide users with a session identifier (session ID or token) that is assigned at session creation time, and is shared and exchanged by the user and the web application for the duration of the session. The session ID is a name=value pair.

With the goal of implementing secure session IDs, the generation of identifiers (IDs or tokens) must meet the following properties:

Session ID Name Fingerprinting

The name used by the session ID should not be extremely descriptive nor offer unnecessary details about the purpose and meaning of the ID. It is recommended to change the default session ID name of the web development framework to a generic name, such as id.

Session ID Length

The session ID must be long enough to prevent brute force attacks, where an attacker can go through the whole range of ID values and verify the existence of valid sessions. The session ID length must be at least 128 bits (16 bytes).

Session ID Entropy

The session ID must be unpredictable (random enough) to prevent guessing attacks, where an attacker is able to guess or predict the ID of a valid session through statistical analysis techniques. For this purpose, a good PRNG (Pseudo Random Number Generator) must be used. The session ID value must provide at least 64 bits of entropy (if a good PRNG is used, this value is estimated to be half the length of the session ID).

Session ID Content (or Value)

The session ID content (or value) must be meaningless to prevent information disclosure attacks, where an attacker is able to decode the contents of the ID and extract details of the user, the session, or the inner workings of the web application.

The session ID must simply be an identifier on the client side, and its value must never include sensitive information. The meaning and business or application logic associated to the session ID must be stored on the server side, and specifically, in session objects or in a session management database or repository.

The stored information can include the client IP address, User-Agent, e-mail, username, user ID, role, privilege level, access rights, language preferences, account ID, current state, last login, session timeouts, and other internal session details. If the session objects and properties contain sensitive information, such as credit card numbers, it is required to duly encrypt and protect the session management repository. It is recommended to create cryptographically strong session IDs through the usage of cryptographic hash functions such as SHA256.

Inactivity Time Out*

Authenticated sessions should timeout after determined period of inactivity - 15 minutes is recommended.

Login & Logout

New session IDs should be created on login (to prevent session fixation via XSS on sibling domains or subdomains). Upon logout the session ID should be invalidated on the server side and deleted on the client via expiration/overwriting the value.

Cookies

The session ID exchange mechanism based on cookies provides multiple security features in the form of cookie attributes that can be used to protect the exchange of the session ID:

Secure Attribute The Secure cookie attribute instructs web browsers to only send the cookie through an encrypted HTTPS (SSL/TLS) connection. This session protection mechanism is mandatory to prevent the disclosure of the session ID through MitM (Man-in-the-Middle) attacks. It ensures that an attacker cannot simply capture the session ID from web browser traffic.

HttpOnly Attribute The HttpOnly cookie attribute instructs web browsers not to allow scripts an ability to access the cookies via the DOM document.cookie object. This session ID protection is mandatory to prevent session ID stealing through XSS attacks.

SameSite Attribute SameSite allows a server define a cookie attribute making it impossible to the browser send this cookie along with cross-site requests. The main goal is mitigate the risk of cross-origin information leakage, and provides some protection against cross-site request forgery attacks.

Domain and Path Attributes The Domain cookie attribute instructs web browsers to only send the cookie to the specified domain and all subdomains. If the attribute is not set, by default the cookie will only be sent to the origin server. The Path cookie attribute instructs web browsers to only send the cookie to the specified directory or subdirectories (or paths or resources) within the web application. If the attribute is not set, by default the cookie will only be sent for the directory (or path) of the resource requested and setting the cookie.

Expire and Max-Age Attributes Typically, session management capabilities to track users after authentication make use of non-persistent cookies. This forces the session to disappear from the client if the current web browser instance is closed. Therefore, it is highly recommended to use non-persistent cookies for session management purposes, so that the session ID does not remain on the web client cache for long periods of time, from where an attacker can obtain it.

- Ensure that sensitive information is not comprised, by ensuring that sensitive information is not persistent / encrypting /stored on a need basis for the duration of the need
- Ensure that unauthorized activities cannot take place via cookie manipulation Ensure secure flag is set to prevent accidental transmission over "the wire" in a non-secure manner
- Determine if all state transitions in the application code properly check for the cookies and enforce their use
- Ensure entire cookie should be encrypted if sensitive data is persisted in the cookie
- Define all cookies being used by the application, their name and why they are needed

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

Cross Site Scripting (XSS)

Given the way browsers parse HTML, each of the different types of slots has slightly different security rules. When you put untrusted data into these slots, you need to take certain steps to make sure that the data does not break out of that slot into a context that allows code execution.

HTML entity encoding is okay for untrusted data that you put in the body of the HTML document, such as inside a "div" tag. It even sort of works for untrusted data that goes into attributes, particularly if you're religious about using quotes around your attributes. But HTML entity encoding doesn't work if you're putting untrusted data inside a "script" tag anywhere, or an event handler attribute like onmouseover, or inside CSS, or in a URL.

XSS Prevention Rules

- Never Insert Untrusted Data Except in Allowed Locations - The first rule is to deny all;
- HTML Escape Before Inserting Untrusted Data into HTML Element Content;
- Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes;
- JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values;
- HTML escape JSON values in an HTML context and read the data with JSON.parse;
- Ensure returned Content-Type header is application/json and not text/html;
- CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values;
- URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values;
- Sanitize HTML Markup with a Library Designed for the Job;
- Prevent DOM-based XSS;

- Use HTTPOnly cookie flag;
- Implement Content Security Policy;
- Use an Auto-Escaping Template System;
- Use the X-XSS-Protection Response Header;
- Properly use modern JS frameworks like Angular (2+) or ReactJS.

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_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 $K1 \neq K2 \neq K3$, 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

Access Control

Authorization is the process where requests to access a particular resource should be granted or denied. It should be noted that authorization is not equivalent to authentication - as these terms and their definitions are frequently confused. Authentication is providing and validating identity. Authorization includes the execution rules that determines what functionality and data the user (or Principal) may access, ensuring the proper allocation of access rights after authentication is successful.

Web applications need access controls to allow users (with varying privileges) to use the application. They also need administrators to manage the applications access control rules and the granting of permissions or entitlements to users and other entities.

Role Based Access Control

Access decisions are based on an individual's roles and responsibilities within the organization or user base. An RBAC access control framework should provide web application security administrators with the ability to determine who can perform what actions, when, from where, in what order, and in some cases under what relational circumstances.

Advantages:

- Roles are assigned based on organizational structure with emphasis on the organizational security policy
- Easy to use
- Easy to administer
- Built into most frameworks
- Aligns with security principles like segregation of duties and least privileges

Problems:

- Documentation of the roles and accesses has to be maintained stringently.
- Multi-tenancy can not be implemented effectively unless there is a way to associate the roles with multi-tenancy capability requirements e.g. OU in Active Directory
- There is a tendency for scope creep to happen e.g. more accesses and privileges can be given than intended for. Or a user might be included in two roles if proper access reviews and subsequent revocation is not performed.
- Does not support data based access control

Areas of caution:

- Roles must be only be transferred or delegated using strict sign-offs and procedures.
- When a user changes his role to another one, the administrator must make sure that the earlier access is revoked such that at any given point of time, a user is assigned to only those roles on a need to know basis.
- Assurance for RBAC must be carried out using strict access control reviews.

Discretionary Access Control

Discretionary Access Control is a means of restricting access to information based on the identity of users and/or membership in certain groups. Access decisions are typically based on the authorizations granted to a user based on the credentials he presented at the time of authentication. The owner of information or any resource is able to change its permissions at his discretion.

Advantages:

- Easy to use;
- Easy to administer;
- Aligns to the principle of least privileges;
- Object owner has total control over access granted.

Problems:

- Documentation of the roles and accesses has to be maintained stringently;
- Multi-tenancy can not be implemented effectively unless there is a way to associate the roles with multi-tenancy capability requirements;
- There is a tendency for scope creep to happen e.g. more accesses and privileges can be given than intended for.

Areas of caution: * While granting trusts; * Assurance for DAC must be carried out using strict access control reviews.

Mandatory Access Control

Ensures that the enforcement of organizational security policy does not rely on voluntary web application user compliance. MAC secures information by assigning sensitivity labels on information and comparing this to the level of sensitivity a user is operating at. MAC is usually appropriate for extremely secure systems including multilevel secure military applications or mission critical data applications.

Advantages:

- Access to an object is based on the sensitivity of the object;
- Access based on need to know is strictly adhered to and scope creep has minimal possibility;
- Only an administrator can grant access.

Problems:

- Difficult and expensive to implement;
- Not agile.

Areas of caution:

- Classification and sensitivity assignment at an appropriate and pragmatic level;
- Assurance for MAC must be carried out to ensure that the classification of the objects is at the appropriate level.

Permission Based Access Control

Is the abstraction of application actions into a set of permissions. A permission may be represented simply as a string based name, for example "READ". Access decisions are made by checking if the current user has the permission associated with the requested application action.

The has relationship between the user and permission may be satisfied by creating a direct relationship between the user and permission (called a grant), or an indirect one. In the indirect model the permission grant is to an intermediate entity such as user group.

A user is considered a member of a user group if and only if the user inherits permissions from the user group. Systems that provide fine-grained domain object level access control, permissions may be grouped into classes. The system can be associated with a class which determines the permissions applicable to the respective domain object.

In such a system a "DOCUMENT" class may be defined with the permissions "READ", "WRITE" and "DELETE"; a "SERVER" class may be defined with the permissions "START", "STOP", and "REBOOT".

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 expects his user to upload a document, the application expects to receive a document for which the intended use will be for reading/printing/archiving. The document should not alter its 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 handles the 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.

Logging and Error Handling

Purpose of logging

Application logging should be always be included for security events. Application logs are invaluable data for:

- Identifying security incidents;
- Monitoring policy violations;
- Establishing baselines;
- Assisting non-repudiation controls;
- Providing information about problems and unusual conditions;
- Contributing additional application-specific data for incident investigation which is lacking in other log sources;
- Helping defend against vulnerability identification and exploitation through attack detection.

Each log entry needs to include sufficient information for the intended subsequent monitoring and analysis. It could be full content data, but is more likely to be an extract or just summary properties.

The application logs must record "when, where, who and what" for each event.

Where to record event data

- When using the file system, it is preferable to use a separate partition than those used by the operating system, other application files and user generated content;
- For file-based logs, apply strict permissions concerning which users can access the directories, and the permissions of files within the directories;
- In web applications, the logs should not be exposed in web-accessible locations, and if done so, should have restricted access and be configured with a plain text MIME type (not HTML).
- When using a database, it is preferable to utilize a separate database account that is only used for writing log data and which has very restrictive database , table, function and command permissions;
- Use standard formats over secure protocols to record and send event data, or log files, to other systems e.g. Common Log File System (CLFS) or Common Event Format (CEF) over syslog; standard formats facilitate integration with centralised logging services.

Which events to log

- Input validation failures e.g. protocol violations, unacceptable encodings, invalid parameter names and values;
- Output validation failures e.g. database record set mismatch, invalid data encoding
- Authentication successes and failures;
- Authorization (access control) failures;
- Session management failures e.g. cookie session identification value modification
- Application errors and system events e.g. syntax and runtime errors, connectivity problems, performance issues, third party service error messages, file system errors, file upload virus detection, configuration changes;
- Application and related systems start-ups and shut-downs, and logging initialization (starting, stopping or pausing);

- Use of higher-risk functionality e.g. network connections, addition or deletion of users, changes to privileges, assigning users to tokens, adding or deleting tokens, use of systems administrative privileges, access by application administrators, all actions by users with administrative privileges, access to payment cardholder data, use of data encrypting keys, key changes, creation and deletion of system-level objects, data import and export including screen-based reports, submission of user-generated content - especially file uploads.

Data to exclude

- Application source code;
- Session identification values (consider replacing with a hashed value if needed to track session specific events);
- Access tokens;
- Sensitive personal data and some forms of personally identifiable information (PII) e.g. health, government identifiers, vulnerable people;
- Authentication passwords;
- Database connection strings;
- Encryption keys and other master secrets;
- Bank account or payment card holder data;
- Data of a higher security classification than the logging system is allowed to store;
- Commercially-sensitive information;
- Information it is illegal to collect in the relevant jurisdictions;
- Information a user has opted out of collection, or not consented to e.g. use of do not track, or where consent to collect has expired.

Error Handling

User Facing Error Messages

Error messages displayed to the user should not contain system, diagnostic or debug information.

Formatting Error Messages

Error messages are often logged to text files or files viewed within a web browser.

- Text based log files: Ensure any newline characters (%0A%0C) are appropriately handled to prevent log forging;
- Web based log files: Ensure any logged html characters are appropriately encoded to prevent XSS when viewing logs.

Recommended Error Handling Design

- Log necessary error data to a system log file;
- Display a generic error message to the user;
- If necessary provide an error code to the user which maps to the error data in the log file. A user reporting an error can provide this code to help diagnose issue.

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

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 receiver 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 protocol 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.

Third-Party Applications

Many social networks also offer the possibility to create additional applications that extend the functionality of the network. The two major platforms for such applications are the Facebook Platform and Open Social. While applications designed for the Facebook Platform can only be executed in Facebook, Open Social is a combined effort to allow developers to run their applications on any social network that supports the Open Social platform (e.g., MySpace and Orkut).

Requirements for a secure third-party applications

- Data Privacy;
- Data Authentication;
- Data Authorization.

Apps that process or query sensitive information should run in a trusted and secure environment. To create this environment, the app can check the device for the following:

- PIN - or password-protected device locking;
- Recent Mobile Platform or OS version;
- USB Debugging activation;
- Device encryption;
- Device rooting (see also "Testing Root Detection").