DIMA PROJECT OFFICIAL DOCUMENTATION

“Watchdog”

Claudio Rizzo
800471

Emanuele Uliana
799256

4/7/2014

Teacher: Prof. Luciano Baresi

Version 1.0

Contents

1 Project context and purpose

1.1 Context

1.2 Purpose

2 Project planning

2.1 Time schedule

3 Requirements analysis

3.1 Actors

3.2 Functional requirements

3.2.1 Mobile phones association

3.2.2 Mobile phone remote localization

3.2.3 Mobile phone remote mark

3.2.4 Mobile phone remote alarm triggering

3.3 Non-functional requirements

3.3.1 Privacy and security: problems and solutions

The remote control of a cellphone is a critical activity and has many security and privacy requirements: the next paragraphs show them briefly: for in-depth explainations see the design section (4.3).

Sender authentication
While the sender (telephone) authentication plays indeed a key role, it's even more crucial the authentication of the person behind a control message; that's the reason for employing a password based authentication scheme: in the initialization wizard the user is required to insert a password which is going to be needed to send a message to that telephone (the basic assumption is the password is known only by the mobile owner and by some people, possibly no one, he trusts). The password is stored hashed with SHA-256 in the application preferences, along with the hashing salt (a random token) to avoid both time-to-memory attacks (such as rainbow tables) and the equality of two hashes generated from two equal passwords; the salt is sent to another telephone after the process of public keys authentication (See section 4.3.6).

Message integrity/authentication/non forgeability/non repudiation
The command messages have some specific security requirements (plus confidentiality which is explained in the next paragraph:

Integrity
The message received must be exactly the one sent: every transmission error or tampering must be detected and cause the abort of the current command session: no retransmission is done.

Authentication
The receiver must have a secure way to understand which telephone the received message comes from.

Non forgeability
Nobody should be able to forge a command message which is both valid and correctly authenticated.

Non repudiation
The sender must not be able to deny he sent a specific message (if he actually did it).

Digitally signing every command message can ensure integrity, authentication, non repudiation and a weak defense against non forgeability: symmetric encryption (and in particular AES-256 in GCM mode of operation) is needed for full protection.

Message confidentiality
No one should be able to detect that and which command is sent to a mobile phone, so the command message is encrypted with the symmetric cipher AES-256 in GCM mode of operation (used for performance reasons and for a supplementary integrity check).

Asymmetric keys management
Digital signatures (and shared secrets computation as we will see) require asymmetric cryptography: in the initialization wizard the application generates and stores in the preferences a key pair based on the elliptic curves; the reasons for this choice are performances and the smaller key length with respect to other keys (like RSA and DSA ones) at a fixed level of security. This makes the 140 characters (bytes) Android limit for a single sms no more a problem.

Symmetric key/initialization vector management
AES-256, being a symmetric cipher, encrypts and decrypts a specific message with the same key, and, given the communication channel is not secure, the two parts must agree on the same key in some way; in particular ECDH is used to compute a common secret once and for all, then, when in need to send a message, the sender picks up a random 32 bytes salt, forwards it to the receiver, then both parts use a keyschedule algorithm (PBKDF2 with HMAC-SHA-256) to derive the same key starting from the secret and the salt. Furthermore the GCM mode of operation requires for every message the sender to generate a 12 bytes random initialization vector and to send it to the receiver.

Public keys mutual authentication
While dealing with asymmetric cryptography, the main problem is to bind a public key with a real user to avoid active Man-In-The-Middle (MITM from now on) attacks. Neither a Pulic Key infrastructure (PKI) or a Web Of Trust (WOT) is employed, because they are both potentially insecure for various reasons (in the PKI case the presence of a trusted element, a certification autorithy hierarchy, which may be compromised/untrusted/fake; in the WOT case the presence of a net of trusted elements, the ones who signed a specific public key, which might be fake/bad persons; furthermore a key with no signatures is not automatically a fake one, but there isn't a way to tell), so the application uses a modified version of the Socialist Millionaire Protocol (SMP) to authenticate to each one each other key; this requires the two parts to have a common secret (an answer to a particular question set up on the fly by the users during the SMP), which is easy to achieve, since the two users are likely to be the same person or two people who trust themselves.

3.3.2 Human friendly interface and transparency

3.3.3 Performances

We chose the crypto algorithms with an eye on the performances of the whole system: the key idea is the bottleneck must be the sms and not the computation time required by the encryptions/decryptions; for this reason the command messages are encrypted with a symmetric algorithm and not with RSA or ElGamal (or another asymmetric algorithm), since symmetric cryptography is faster than asymmetric at least by two orders of magnitude (they are very likely to be 3 anyway); however to do ECDH and ECDSA the application needs also an asymmetric key pair, which is generate during the intitial wizard once an for all, so an acceptable overhead. The public keys mutual validation (SMP + ECDH in practice) takes some time, but it's done only one time per association, which means two telephones have to do it only when they associate themselves. Finally the digital signature/verification process are quite fast and so is the key-derivation from the secret and the salt.

3.4 Use cases

3.4.1 Initialization wizard

3.4.2 Mobile phones association

3.4.3 Remote control: localization

3.4.4 Remote control: mark stolen/lost/both/found

3.4.5 Remote control: alarm triggering/untriggering

4 Design

4.1 Application Architecture

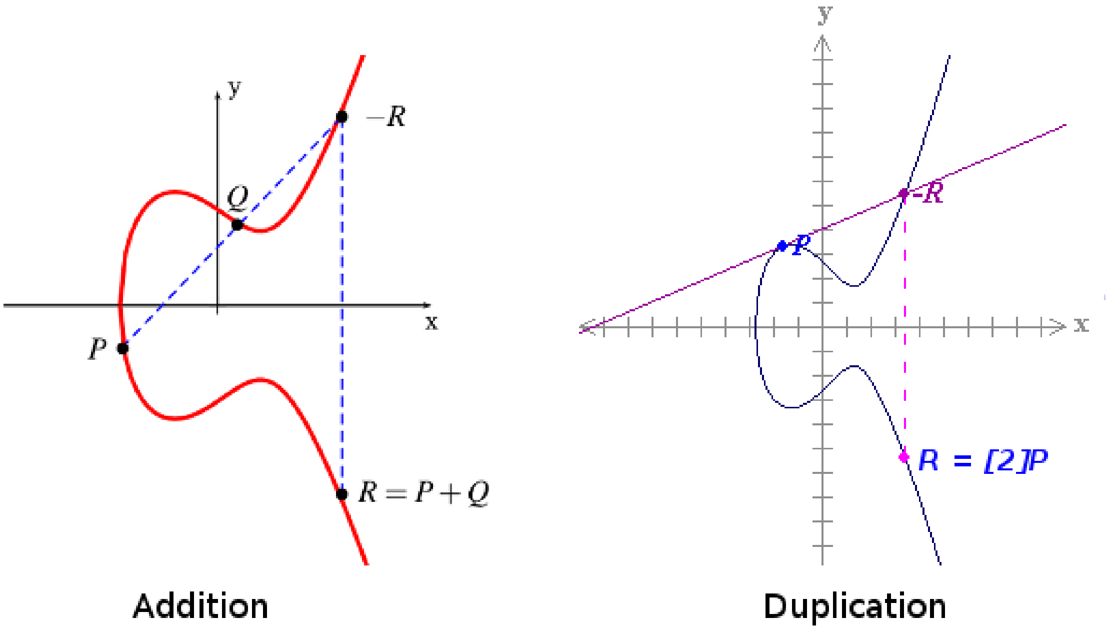
4.2 Design Patterns

4.3 Crypto protocols and alogrithms

The application makes a heavy use of cryptography, so we needed a good crypto provider for java, which we believed to have found in Bouncycastle; however its libraries are not convertible into the Dalvik format, so we had to rely on Spongycastle, an unofficial Bouncycastle porting for Android. Unluckily some algorithms/protocols we had intention to use (namely FH-MQV/ECMQV and the native SMP) are not supported (no java implementation for them found), so we ended up using ECDH and a homemade version of SMP instead.

4.3.1 Elliptic Curves key pair generation

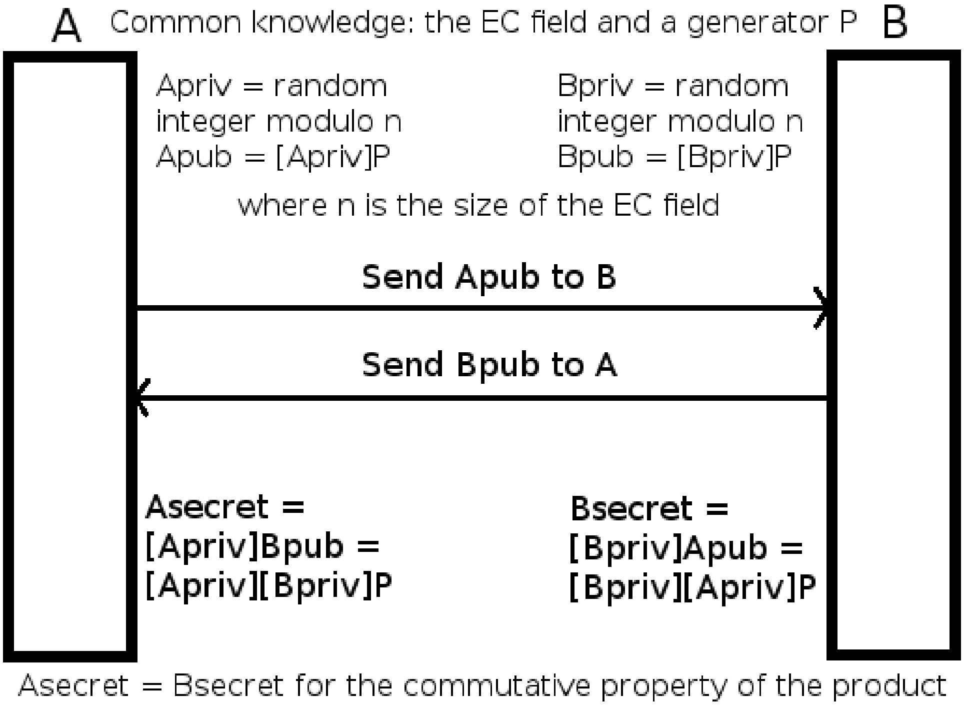
The public/private key pair generation is done by using elliptic curves for performance and memory complexity reasons (at a fixed security level the EC keys are more than 10 times shorter compared to the RSA/DSA/ElGamal ones): a message in Android has a maximum length of 140 characters (bytes), so we had no choice if we didn't (and we didn't since it would have screwed up our crypto layer) want to use multipart messages. The curve used is a NIST standard: "secp256r1", also known as "prime256v1", which generates 256 bits long keys (we actually thought to use "secp521r1" for enhanced security, but the keys were too long). Using a named curve which is also a standard has a few advantages: first, it generates automatically all the parameters needed, second, its security has been widely tested by the cryptanalists of all the world. The generated keys are encoded into byte arrays in different ways: the private ones using the "PKCS8 encoded key specifiers", the public ones with the "X509 encoded key specifiers". It's always possible with a key factory to decode both encodings leading to keys identical to the pre-encoding ones. The keys are generated this way: given the order of the EC group and a randomly chosen point of the curve (different from the one at infinity), which is also a generator of the group, the private key is that point and the public key is computed as [priv]P, where the multiplication denoted by [integer]Point is reduced to a sequence of application of doubling a point and addition between two points. The pictures below shows graphically how these operations are defined for elliptic curves.



Picture 1: ECDH schema

4.3.2 Elliptic Curves Diffie Hellman key exchange

The Diffie-Hellman key exchange (DH) is a key-agreement protocol used to generate a common secret between the two parts over an insecure channel; the computed secret is guaranteed to be the same for both and it's in the form of a byte array (in our case with length 32). From that secret then a deterministic key generation algorithm is able to extract a symmetric key usable for encryption/decryption. The Elliptic Curves DH (ECDH) works like this: every part computes a key pair over the same EC (we use the ones computed in the wizard), then both send their own public key to the other; now they multiply (for a proper definition on multiplication in a EC field) the public key of the other by their own private key: the result is the same due to the public/private EC keys mathematical properties as shown in the picture below.


A
Common knowledge: the EC field and a generator P
B
Apriv = random integer modulo n
Apub = [Apriv]P
Bpriv = random integer modulo n
Bpub = [Bpriv]P
where n is the size of the EC field
Send Apub to B
Send Bpub to A
Asecret = [Apriv]Bpub = [Apriv][Bpriv]P
Bsecret = [Bpriv]Apub = [Bpriv][Apriv]P
Asecret = Bsecret for the commutative property of the product

Picture 1: ECDH schema

The problem with ECDH is the lack of authentication of the received public key: an active MITM could impersonate user A with user B and user B with user A by tricking them into believe his public key belongs to the other side while actually it's not true. This single point of failure is solved by embedding ECDH into SMP (see section 4.3.6).

4.3.3 Elliptic Curves Digital Signature Algorithm

Nowadays ECDSA is the best known algorithm for computing digital signature with a reasonable size, high performances and very high security: it's the EC variant of the DSS-DSA algorithm and it works like this: assumed the signer (sender) has a keypair based on EC (Apriv = s, Apub = [s]P), and the receiver knows Apub and trusts it, and both parts know the curve and its parameters (n = size of the group, P a generator of the group), then:

1. The signer chooses a random integer $r\ mod\ n$ such that $n > 0$ and $GCD(r, n) = 1$
2. The signer computes $[r]P = (x, y)$
3. If $x = 0$ goto step 1, else the signer stores x as k
4. The signer computes $r^{-1}\ mod\ n$ and $e = SHA - 1(m)$ where m is the message to sign
5. The signer computes $z = r^{-1}(e + sk)\ mod\ n$
6. If $z = 0$ goto step 1, else the signature is made by (k, z)

Verification:

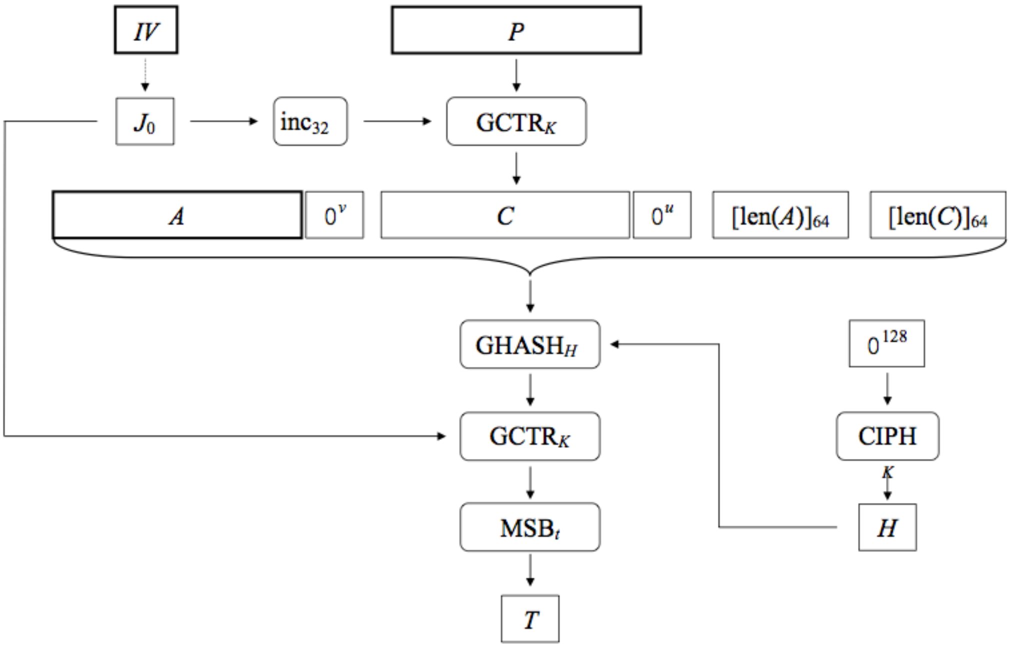
1. The receiver checks whether k and z are $mod\ n$ and positive not null integers, if not, the signature is not valid
2. The receiver computes $e = SHA - 1(m)$ and $w = z^{-1}\ mod\ n$
3. The receiver computes $u_1 = ew\ mod\ n$ and $u_2 = kw\ mod\ n$
4. The receiver computes a point $X = [u_1]P + [u_2][s]P$ and stores it as (x, y)
5. If X is the point at infinity or is not verified $x = k\ mod\ n$, the signature is not valid, otherwise is valid.

4.3.4 PBKDF2 with HMAC-SHA-256

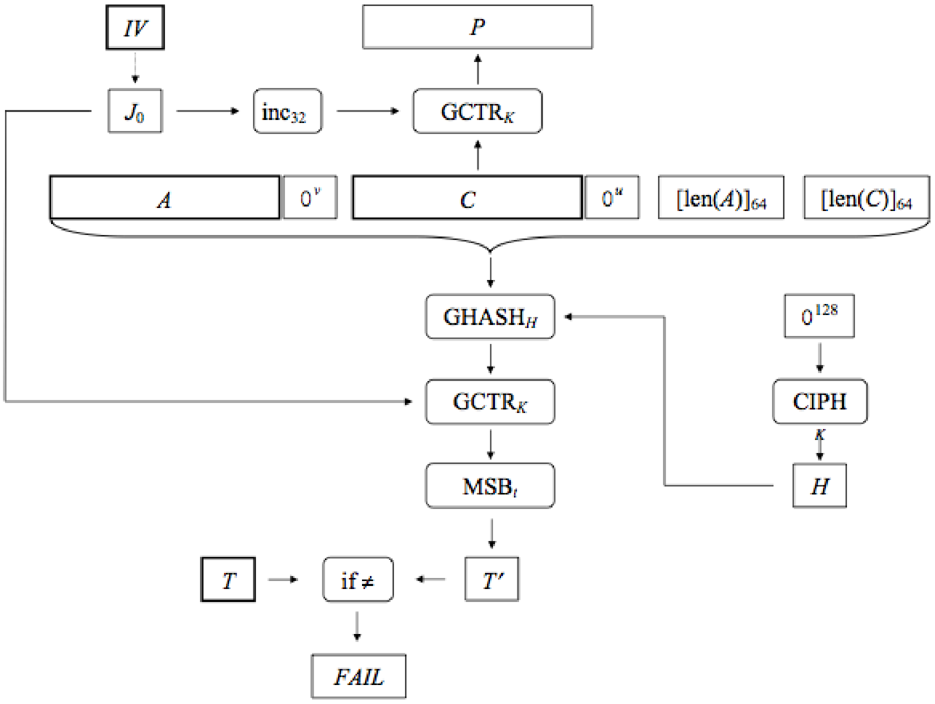
The Password Based Key Derivation Function #2 is a deterministic key-derivation algorithm which generates a symmetric crypto key with a desired length by taking in input a password or a passphrase (in our case the secret from ECDH) and a salt (which makes possible to derivate different keys from the same password) and doing a certain number (4096 in our case) of application of a pseudorandom function (HMAC-SHA-256 in our case) to them in the so called "key stretching", which makes very hard the use of rainbow tables for cryptanalysis.

4.3.5 AES-256-GCM

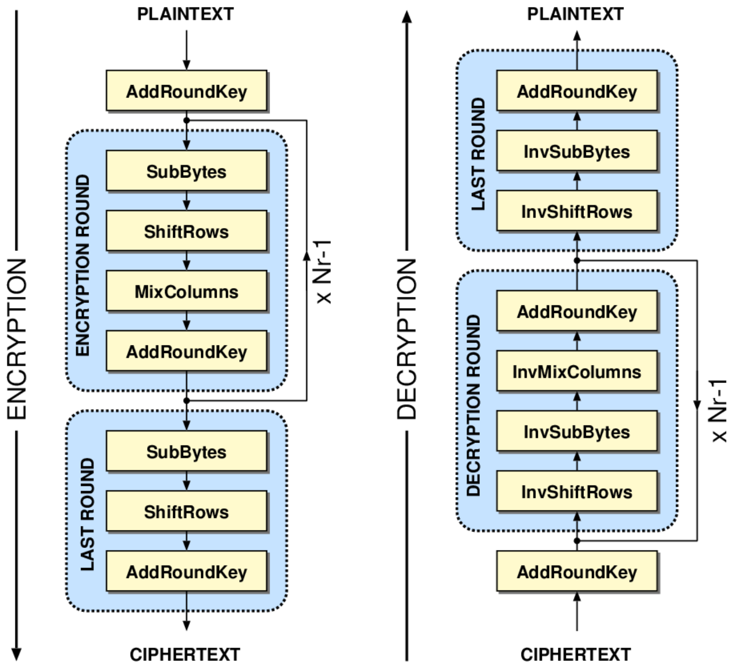
The Advanced Encryption Standard is the symmetric encryption standard algorithm (NIST FIPS-197) almost worldwide since 2001, and it's known for high performances and security against cryptanalysis; in the application we employed its version with a 256 bits key, which has a security margin comparable to RSA with a 15360 bits key. Actually, since it's used in Galois/Counter Mode (GCM), it does not encrypt directly the plaintext, but it's instead used together with a 12 bytes initialization vector IV (which doesn't need to be secret, but unpredictable for every encryption, so the best choice is to pick it up randomly) to generate a pseudorandom keystream, which is then XORed to the plaintext: basically the block cipher simulates a stream cipher (so no padding is needed for the plaintext, whose length, thus, does not need to be a multiple of 16 bytes, like in direct encryption, which is a good thing, considering the 140 bytes limit for Android messages) which simulates a One Time Pad (OTP), the only cipher perfectly secure. Obviously AES-GCM is not perfectly secure because, frist, soon or later, a new key will be equal to one used in the past (not so soon though), second, the key is shorter than the plaintext and, third, the keystream is only pseudorandom. In addition to this the GCM mode generates also a 16 bytes Message Authentication Code (MAC) which is prepended to the ciphertext in order to provide an additional integrity check. Before the decryption (which needs of course the same IV and the same key) the message integrity is verified and, if it's not the case, an exception is raised and the command session is aborted. In GCM mode it's also possible to attach to the ciphertext some non-encrypted data (called Additional Authenticated Data, AAD) which are used along with the ciphertext itself to produce the MAC; in our case this feature is useless, so no AAD. The three pictures below show the encryption and the decryption under GMC (P = plaintext, K = key, C = ciphertext, A = AAD, T = MAC) and the AES encryption/decryption used to generate the keystream.



Picture 1: Encryption in GCM mode of operation



Picture 2: Decryption in GCM mode of operation



Picture 3: Encryption/Decryption in AES

4.3.6 Socialist Millionaire Protocol

The Socialist Millionaire protocol purpose is to mutual authenticate public keys so that a logical binding is created between a key and a phisical device; this solves the active MITM problem. Since a java implementation of the original SMP was not available, we coded our own version of it which also embeds ECDH to spare time: the whole protocol is based on messages exchanging.

Public key request

Public key sending

Question sending

Hash sending

Ack and password salt sending

Second half

Error management

4.3.7 Command Protocol

First message

Second message

Third message

Fourth message

Error management

Timeout management

5 Testing

5.1 Cryptography

5.2 Protocols

6 Installation and usage manual

6.1 Installation

6.2 Usage

6.2.1 Initialization wizard

6.2.2 Change application settings

6.2.3 Send a command message