Cryptography basics for embedded developers

Embedded Linux Conference, San Diego, 2016
"If you think cryptography is the solution to your problem, then you don't understand your problem"

- Roger Needham
Cryptography basics are important

- Misuse of cryptography is common source of vulnerabilities
  - “41 of the 100 apps selected [...] were vulnerable [...] due to various forms of SSL misuse.” *

- Understanding crypto basics will improve the security of devices
  - Important for anyone using cryptography (e.g. libraries)

- Think about security requirements for your product
  - Can it be attacked? Why would it? How?
  - Consider how cryptography can be applied correctly to support your requirements

- Reduce the risk of your product being compromised

About me

- **Eystein Stenberg**
  - CS/Crypto master’s
  - 7 years in systems, security management
  - eystein@mender.io

- **Mender.io**
  - Over-the-air updater project for Linux/Yocto
  - Under active development
  - Open Source

- Reach me after on email or exhibitor hall
The mandatory legal note

- Some use of cryptography / software has legal implications
- Most notably: export restrictions in the USA
- I will only consider technological aspects, not legal ones
Session overview

● Our goals

● Crypto basics and pitfalls
  ○ Encryption
  ○ Signatures & Message Authentication Codes
  ○ Secure hashing
  ○ Key management

● Crypto for embedded
  ○ Expensive operations
  ○ Alternatives
Attacker motivation

- Why would someone attack your product?
- Can someone *make money* from a compromise? How much?
- All crime starts with *a motive*
Your goal is to *lower attacker ROI*

- It is always *possible* to compromise

- Lower Return on Investment (ROI) for attacker; either
  - *Decrease value* of successful attack
  - *Increase cost* of successful attack

- Focus on increasing cost of attack in this session
Decreasing value of attack can be effective too
CIA concepts implemented with crypto primitives

- **Confidentiality**
  - Is there something secret?
  - Primitives: encryption

- **Integrity**
  - Should we detect altering of information?
  - Primitives: secure hashing, signatures, MAC

- **Authenticity**
  - Do we need to know who create/request information?
  - Primitives: signatures, MAC
Symmetric encryption: one shared secret key

- Use for **confidentiality**
- Efficient, relatively low resource consumption
- Typical key & block sizes: 128, 192, 256 bit
- Difficult to keep **shared** things **secret**
- Note **block cipher mode** when encrypting large volumes of data with same key
- Example: AES (Advanced Encryption Standard) + CBC mode
Pitfall: Use insecure symmetric block cipher *mode*

Original  
Encrypted with ECB mode  
Encrypted with CBC mode

Source: Larry Ewing
Asymmetric encryption: public and private key

- Use for **confidentiality of little data** (e.g. symmetric key) with multiple parties
  - Very compute-intensive operation (~1000 x symmetric)
  - Large volume of ciphertext can leak information about private key
- Advantage over symmetric: safe to share public key with anyone
- Examples: RSA (key/block size ~4096 bits), Elliptic Curve (key/block size ~256 bits)
Message Authentication Code (symmetric)

Generation:
- Message
- MAC alg.
- MAC1

Verification:
- Message
- MAC alg.
- MAC2

Send Message & MAC

- Use for **authenticity**
- Efficient, typical key & MAC sizes: 160, 256 bit
- Difficult to keep **shared** things **secret**
- If you need confidentiality too, look at Authenticated Encryption (AE/AEAD)
- Example: HMAC-SHA256
Digital signature (asymmetric)

**Generation:**
- Message
- Secure hash
- hash1
- Sign
- Signature
- Send Message & Signature

**Verification:**
- Message
- Secure hash
- hash2
- Verify
- Signature
- Accept/Reject

- Use for **authenticity**
- Less efficient than MAC (~1000x), but no shared secret
- Common misconception: “signing is encrypting with private key”
- Examples: DSA (key/block size ~4096 bits), ECDSA (key/block size ~256 bits)
Cryptographically secure hashing

Given hash, infeasible to generate a message that yields the hash.

Infeasible to modify a message in such a way that it generates the same hash.

Infeasible to find any two messages that yield the same hash.

Hash is efficient to compute.
Hash function implementations

- Insecure if it does not meet all four criteria

- Secure hash algorithm (SHA) family
  - SHA-256, SHA-384, SHA-512 (number denotes bits of output)

- Insecure hash algorithms
  - MD5 (128 bits): Attack that can find two messages with same hash in seconds
  - SHA-1 (160 bits): Attack reduced collision to 63-bit operation (ideal is 160/2 = 80)

- Bottom line: use SHA-256 (or larger) if you use it for security
All cryptography is based on **keys**

- If someone can make you use the wrong key, security is broken
  - Need secure {ID, key} mappings

- Secure key exchange **requires a pre-existing secure channel** (barring quantum crypto)
  - Typically inserted during **provisioning** (e.g. web-browsers, phone apps, ...)

- It is a notoriously hard problem, especially in **many-to-many conversations** (e.g. web)
Using the right key: Public Key Infrastructure (PKI)

- Most common way to “solve” the key exchange problem
- Delegate problem with absolute trust to one (or more) Certificate Authority (CA)
  - If CA says it’s the right binding by signing \{ ID, key \}, we will trust him
- Still need to securely obtain CA’s key (pre-existing secure channel, e.g. provisioning)
- Introduces a single point of compromise for the entire system (CA’s private key)
- Complex to manage (keep the CA secure, rekeying CA, cert issue, cert revocation, ...)

Alice

Hello, I’m Alice, please store my Key A

The Evil Network

Bob

My CA vouches for this being Alice’s key, so I accept.

Certificate

Secure channel

CA’s key

Alice, Key A, CASign
Avoid CA certificates, trust public keys directly (to varying degrees)

Web of trust; OpenPGP (GPG/PGP)
  ○ Like a distributed CA
  ○ “I trust T & J, T & J trusts A, so I trust A”

 Might be a better fit for one-to-many (e.g. clients w/ single server)
  ○ Simpler, avoids the run-your-own-CA complexities
  ○ Limited use of certificates anyway here (sent just to client and server)
Key management

- Some keys need to be exchanged
- All security breaks if secret keys are compromised
- The hardest part of implementing cryptography

Some tips
- Don’t share secret keys between many devices
- Use asymmetric cryptography
- Store secret keys on non-removable media with strict file permissions
- Ensure that keys can be decommissioned / rotated
- Consider hardware-assistance (only operations are available to software, not keys)
Implementing cryptography in embedded

- We need it to be efficient!
  - Cryptography is based on advanced mathematical operations

- Asymmetric cryptography is very expensive on CPU/memory
  - Order of 1000x of symmetric counterparts typically
  - Use it sparingly
  - Use Elliptic Curve Cryptography (ECC)

- Look for hardware support (crypto processor)
Use Elliptic Curve Cryptography over RSA/DSA

- Typically aim for 128-bit security level or higher today (but it's up to you)
- RSA/DSA requires 12x the key size at this level
- TLS with ECC is 3-10x faster (CPU time) at this level*

* Source: Performance Analysis of Elliptic Curve Cryptography for SSL, V. Gupta, S. Gupta, S. Chang
Cryptography basics that will improve your security

● Key management is hard
  ○ At least you are aware
  ○ Consider trust-based key exchange
  ○ Avoid putting a single secret all over the place

● Use industry standard libraries and high-level functions
  ○ Never ever ever implement your own cryptographic algorithms!

● Consider ECC over RSA for performance in asymmetric crypto

● Use SHA-256 (or higher) for secure hashing
Is there a secret backdoor?