Digital Signatures and the Beginning of the World

David Brown
Introduction

- About MCUboot, an IoT bootloader
  - Systems with 0.5-1 MB flash, and 16-256K RAM
  - Constrained, small, yet connected to the world
- Bootloader is the beginning of trust
  - Without it, no trust
  - Needs to verify the next image should be run and is untampered
- Digital Signatures
  - Public Key cryptography
  - Public Key in bootloader
  - Private key used offline to sign images
- What have we learned with MCUboot
Digital Signatures

OEM or Factory

Header
magic, size, flags, ...

Image

SHA256

246ba73116ee7b9e...

sign

Private key

On Device, in bootloader

Header
magic, size, flags, ...

Image

SHA256

246ba73116ee7b9e...

Signature: 4b381f8...

verify

Public Key

Good or Bad Signature?
Signature Algorithms

- **RSA**
  - Old, well understood
  - Problems are understood and not too hard to get verification right
  - Many attacks, such as cache and timing, apply to the signing of messages, not verification, makes our job easier
  - Keys and signatures are huge:
    - RSA-2048 are 256 bytes for signature, ~270 for public key

- **Elliptic Curves**
  - More complex math
  - Much shorter keys/sigs, e.g. ~91 bytes for public key, as little as ~32 bytes for signature
  - How to choose curves (and who do you trust)
  - We support NIST P-224 and P-256 curves with ECDSA
  - Soon, will be adding Ed25519
Curve Problems

- Elliptic Curve cryptography is sensitive to the choice of curve chosen.
- Many use NIST-defined curves, these have magic numbers, and many are suspicious these can be subverted.
- Implementations tend to be tied to a specific curve (general EC code is typically 1/10 the speed).
- Looking into Ed25519 to avoid these issues.
Storage Formats

- A signature is generally a few large numbers
  - ECDSA
    - Two large numbers
    - Standard defines an encoding
    - End result is wrapped in ASN.1 DER encoded block
  - RSA:
    - Single large number
    - Standards define ASN.1 DER encoded block holding it
  - Ed25519
    - Just 64 bytes, doesn’t need encoding or wrapping

- RSA weaknesses
  - “Signing” incorrect things can leak information about private key (see references at end)
  - PKCS #1 v1.5 defines a block to prevent this
  - PSS scheme wraps hash to be signed in complex block to prevent this, PSS is possibly more secure
Implementations

● 1. Never design your own cryptography
● 2. Never implement your own cryptography
● As such, we use some existing implementations
  ○ mbed TLS [https://tls.mbed.org/](https://tls.mbed.org/)
  ○ TinyCrypt [https://github.com/01org/tinycrypt](https://github.com/01org/tinycrypt)
  ○ NaCL and TweetNaCL [https://nacl.cr.yp.to/](https://nacl.cr.yp.to/) and [https://tweetnacl.cr.yp.to/](https://tweetnacl.cr.yp.to/)
mbed TLS

● Pros
  ○ Reasonably complete TLS/DTLS
  ○ Includes numerous crypto primitives, sufficient to implement RSA and some ECDSA
  ○ Contains a small, simple, ASN.1 DER decoder
  ○ MCUboot uses it for RSA and ECDSA, with the P-224 curve

● Cons
  ○ Some parts are a bit large (think goal of 16K of codespace total for bootloader)
  ○ bignum math requires malloc/free
TinyCrypt

- Very small “focused” crypto library
- Has a small set of primitives and some building blocks
- Has most of the parts needed for ECDSA with P-256
- We use for this signature
- Still need the ASN.1 decoder from mbed TLS
NaCL, and TweetNaCL

- Similar implementations of Ed25519
- NaCL Ed25519 is about 100KB of Code
- TweetNaCL for Ed255 is about 3KB of Code
- Yes, 3K
- But, NaCL is about 35 times faster
- Signature verification with TweetNaCL is about 1-2 seconds
Ed25519 “problems”

- Defined to hash message with a nonce prefix
- Both implementations require a RAM buffer slightly larger than image to sign or verify
- This doesn’t work with small RAM
- It is possible to sign a hash rather than the message
- Defeats some of the protections in place
- Could be fixed, is an implementation issue, not a protocol/algorithms issue
MCUboot

- Choices
- Configure which signing algorithm is used, brings in appropriate crypto libraries
- Different tradeoffs for different devices
  - RSA has large keys, but is fast
  - ECDSA has small keys with cost of time
  - Ed25519 is “best of both worlds”, as long as you have lots of ROM available. Otherwise, it is only useful with 1-2 seconds to boot are acceptable
**imgtool.py**

- Most docs about keys end up with list of crypto ‘openssl’ commands
- Commands for RSA, and ECDSA are quite different
- OpenSSL doesn’t support Ed25519
- We wrote ‘**imgtool.py**’
  - key generation
  - extracting public key for MCUboot
  - signing images
Some references

- https://mcuboot.com/
- https://github.com/runtimeco/mcuboot
- RSA weakness with chosen plaintext
Thank You

#SFO17
BUD17 keynotes and videos on: connect.linaro.org
For further information: www.linaro.org