Secure Vehicle Communication

API Attacks on Tamper Resistant Modules

Levente Buttyan
BME / CrySyS Lab
buttyan@crysys.hu

The need for tamper resistant modules

- implementing security services in vehicular networks requires cars to store sensitive data
  - cryptographic keys (secret keys, private keys), event logs, …
- sensitive data needs to be protected from unauthorized access
- cars operate in a “hostile” environment
  - unsupervised access to all parts of a car by potential attackers
  - incentives to compromise data
- attack detection in “real-time” is impossible
  - tampering with cars may be detected by authorities at regular inspections, but only months (or perhaps years) later
- attacks should be prevented
  - needs tamper resistant hardware in cars

Tamper resistant modules

- a tamper resistant module is a custom computer in tamper resistant packaging
  - hardware support for cryptographic functions
  - tamper detection and reaction circuitry
  - internal battery and clock
  - …
  - API
- the API of a tamper resistant module is a software layer through which the module’s functions are exposed to the external world

An example API

- key_export
  - inputs
    - key token: $E_{MK}(K)$
    - key encryption key token: $E_{MK}(KEK)$
  - outputs
    - exported key token: $E_{KEK}(K)$

- key_import
  - inputs
    - external key token: $E_{KEK}(K)$
    - key encryption key token: $E_{MK}(KEK)$
  - outputs
    - imported key token: $E_{MK}(K)$

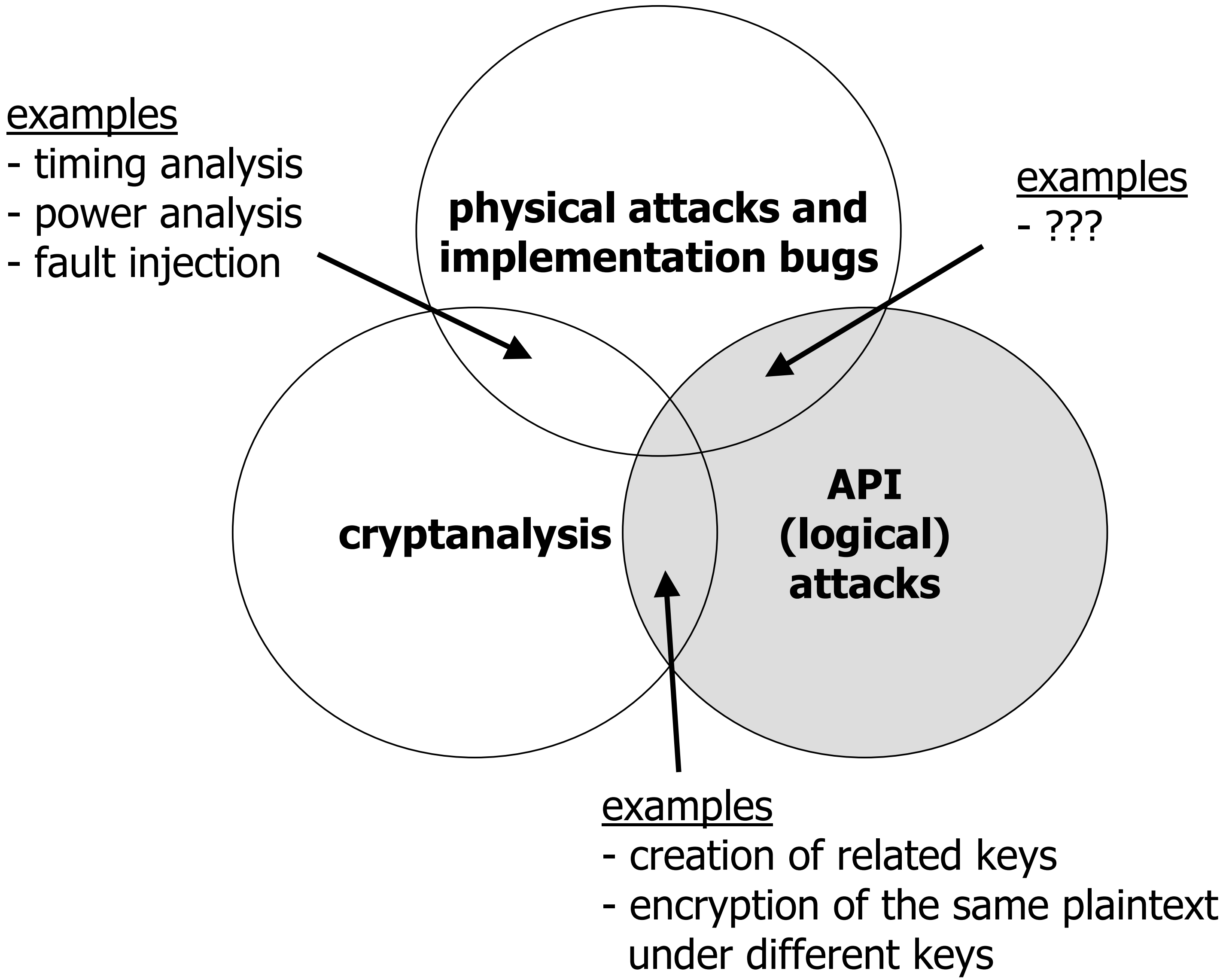
An example API

- key_part_import
  - inputs
    - key part: K’
    - key token: $E_{MK}(K)$
  - outputs
    - updated key token: $E_{MK}(K+K')$

- encrypt
  - inputs
    - key token: $E_{MK}(K)$
    - data: X
  - outputs
    - encrypted data: $E_K(X)$

- …

Attacks on tamper resistant modules



API attacks

- exploit design weaknesses of the API for extracting secrets from the module or increasing the efficiency of cryptanalytical attacks
- simple examples:
  - creating related keys:

    key_part_import (K', E_{MK}(K)) $\rightarrow$ creates K+K'

    key_part_import (K'+Δ, E_{MK}(K)) $\rightarrow$ creates K+K'+Δ
  - key conjuring:

    key_import (R, R') $\rightarrow$ creates an unknown key $D_{DMK(R')}$(R)
  - key separation:

    key_export (E_{MK}(K), E_{MK}(KEK)) $\rightarrow$ returns E_{KEK}(K)

    decrypt (E_{MK}(KEK), E_{KEK}(K)) $\rightarrow$ returns K

Attacking the API of the IBM 4758

- preliminaries
  - keys are stored externally in *key tokens*
  - key tokens are encrypted with a master key or a key wrapping key (exporting key) *modulated with the type* of the key in the token
  - types are encoded in *control vectors*
  - example:
    - let K be an exportable symmetric data encryption key
    - let KEK be a key encryption key
    - it is possible to export K under the protection of KEK in a key token $E_{KEK+CV\_DATA}(K)$

Attacking the API of the IBM 4758

- use key_part_import to create two unknown but related key encryption keys UKEK and UKEK':

key_part_import (K', E_{MK}(K), "KEK")

→ creates UKEK = K + K' : "KEK"

→ outputs E_{MK+CV_KEK}(UKEK)

key_part_import (K' + CV_KEK + CV_DATA, E_{MK}(K), "KEK")

→ creates UKEK' = K + K' + CV_KEK + CV_DATA : "KEK"

→ outputs E_{MK+CV_KEK}(UKEK')

UKEK' = UKEK + CV_KEK + CV_DATA

Attacking the API of the IBM 4758

- use key_import to create two copies of an unknown random key URK with different types:

key_import (R, E_{MK+CV_KEK}(UKEK), “KEK”)
→ creates URK = $D_{UKEK+CV\_KEK}$(R) : “KEK”
→ outputs E_{MK+CV_KEK}(URK)

key_import (R, E_{MK+CV_KEK}(UKEK’), “DATA”)
→ creates URK’ = $D_{UKEK’+CV\_DATA}$(R) : “DATA”
→ outputs E_{MK+CV_DATA}(URK’)

$$
\begin{aligned}
URK' &= D_{UKEK'+CV\_DATA}(R) \\
&= D_{UKEK+CV\_KEK+CV\_DATA+CV\_DATA}(R) \\
&= D_{UKEK+CV\_KEK}(R) \\
&= URK
\end{aligned}
$$

Attacking the API of the IBM 4758

- export URK:“DATA” under URK:“KEK”:

  key_export (E_{MK+CV_DATA}(URK), E_{MK+CV_KEK}(URK), “DATA”)
  $\rightarrow$ outputs E_{URK+CV_DATA}(URK) = E_{URK}(URK)
  because CV_DATA = 0

- decrypt E_{URK}(URK) with URK:”DATA”:

  decrypt (E_{MK+CV_DATA}(URK), E_{URK}(URK))
  $\rightarrow$ returns URK

- export any target key K_{target} under URK:”KEK”:

  key_export (E_{MK+CV_ANY}(K_{target}), E_{MK+CV_KEK}(URK), ANY)
  $\rightarrow$ returns E_{URK+CV_ANY}(K_{target})

PKCS #11

- Cryptographic Token Interface (cryptoki) Standard
- supported by many products including Mozilla and various SSL hardware accelerators
- among many others, cryptoki includes a key management interface:
  - C_GenerateKey
  - C_GenerateKeyPair
  - C_WrapKey
  - C_UnwrapKey
  - C_DeriveKey
  - …
- secret key objects have a control vector that specifies the intended usage
  - encrypt / decrypt
  - sign / verify (MAC)
  - wrap / unwrap

Attacking PKCS #11

- key separation attack
  - control vector elements can be independently set
  - one may insert a key with type “wrap” and “decrypt”
  - this key can be used to export and decrypt any exportable key
- weaker key / algorithm attack
  - it is possible to wrap a private key with a weak symmetric key or using a weak algorithm
- small public exponent with no padding
  - symmetric keys can be wrapped with public keys using no padding (i.e., textbook RSA)
  - resulting key token is $T = k^e \bmod n$
  - if $k^e < n$ ($e < \log n / \log k$), then $k = T^{1/e}$
  - condition satisfied if $e = 3$, $\log n = 1024$, $\log k = 128$

Attacking PKCS #11

- Trojan public key
  - public keys are not authenticated
  - one can export a target key under a supplied public key for which she knows the corresponding private key
- Trojan wrapped key
  - no authentication for wrapped keys
  - one can wrap any key with a known public key and import it into the device
- private key modification
  - private key token contains (n, e, d, p, q, d mod (p-1), d mod (q-1), q-1 mod p) encoded as a byte string and encrypted in CBC mode
  - one modified block in the ciphertext affects only the corresponding block and the next block in the plaintext
  - parameters can be modified
  - may be used in fault injection attacks

Lessons learnt

- no matter how secure the device is physically if it leaks secrets due to API attacks
- most tamper resistant devices are vulnerable to some form of API attacks
- careful design and analysis of the API is indeed very important with respect to overall security

Security analysis of APIs

- API attacks can be very subtle and hard to discover by informal analysis
- the problem of API analysis seems to be very similar to that of analyzing authentication and key exchange protocols
  - the attacker interacts with the device using a well defined set of “messages”
  - the goal is to obtain some secret or bring the device in a “bad” state
- formal analysis techniques developed for key exchange protocols may be amenable to the analysis of crypto APIs

References

- Mike Bond. **Attacks on Cryptoprocessor Transaction Sets.** CHES 2001.
- Jolyon Clulow. **The design and analysis of cryptographic application programming interfaces for security devices.** Master's thesis, University of Natal, Durban, 2003.
- Jolyon Clulow. **On the Security of PKCS #11.** CHES 2003.
- Mike Bond, Jolyon Clulow. **Extending Security Protocol Analysis : New Challenges.** ARSPA 2004.