PA193 - Secure coding principles and practices

Designing good and secure API
Automata-based programming

Petr Švenda svenda@fi.muni.cz

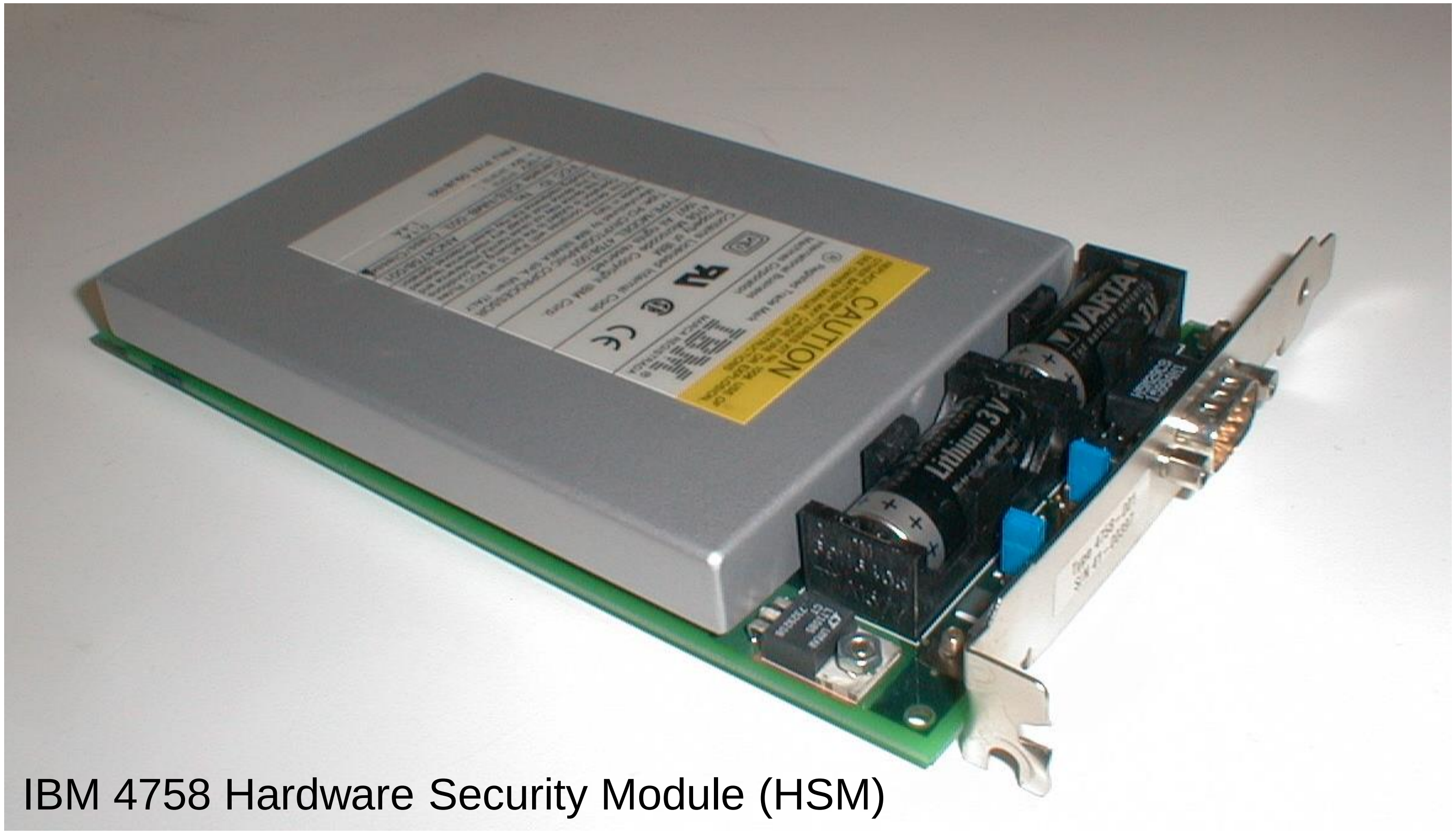
CROCS
Centre for Research on Cryptography and Security

Overview

- Lecture: problems (demo), prevention
  - How to write good & secure API
  - automata-based programming
- Labs
  - Write small security API
  - model inner state with automata-based design

PROBLEM

What is this device for?


IBM 4758 Hardware Security Module (HSM)

Hardware Security Modules (HSM)

- Hardware Security Modules are high-security devices
  - small security computer inside general purpose computer
    - RAM, CPU, storage...
    - resilience against tampering, side-channels...
  - support various cryptographic operations
  - keys are generated, stored and used directly on the device
  - additional restricted code can be uploaded (firmware)
- HSM is trusted component, accessed by not-so-trusted applications
  - HSM’s API serves as wall between different levels of trust

API

What is API and ABI?

- API = Application Programming Interface
  - source code-based specification intended to be used as an interface between software components to communicate
  - classes, interfaces, methods...
- ABI = Application Binary Interface
  - specification of interface on binary level
  - size, binary representation and layout of data types
  - function calling conventions (stdcall, decl...)
  - how to make system calls (functions outside program memory)
  - binary formats of data produced...
- API != ABI, but both are necessary

When we use API?

- All the time ☺
- When using function from standard library
- When using external library
- When calling system (Win32 API, POSIX...)
- When calling methods of our own class
- ...

When we design API?

- Almost all the time ☺
- Function signature is API for this function usage
- List of public methods in interface is its API
- When we create good API?
  - good programming habit is to create reusable modules
  - every module has its own API
  - once module will get reused, API cannot be changed easily

Application programming interface (API)

- Different types of API

1. Non-security API
   - any library API (module/library interface)
   - e.g., C++ STL, Boost library...
2. Cryptographic API
   - set of functions for cryptographic operations
   - e.g., Microsoft CryptoAPI, OpenSSL API...
3. Security API
   - allows untrusted code to access sensitive resources in secure way
   - e.g., PKCS#11 HSM module, suExec, OAuth

Principles of good API (Trolltech)

1. Be minimal
2. Be complete
3. Have clear and simple semantics
4. Be intuitive
5. Be easy to memorize
6. Lead to readable code

- http://doc.qt.digia.com/qq/qq13-apis.html

Principles of good API (Joshua Bloch)

1. Easy to learn
2. Easy to use, even without documentation
3. Hard to misuse
4. Easy to read and maintain code that uses it
5. Sufficiently powerful to satisfy requirements
6. Easy to extend
7. Appropriate to audience

- http://lcsd05.cs.tamu.edu/slides/keynote.pdf

Credits: Joshua Bloch, Arnab Biswas

- Joshua Bloch, How to Design a Good API and Why it Matters (Google)
  - http://lcsd05.cs.tamu.edu/slides/keynote.pdf
  - video: http://www.infoq.com/presentations/effective-api-design
- Arnab Biswas, Fundamentals of good API designing
  - http://www.javacodegeeks.com/2013/05/fundamentals-of-good-api-designing.html
- Reading/watching is highly recommended
- Many ideas taken from his presentation
  - (demonstrated on cryptographic libraries by myself)

Process of API design

1. Understand the requirements
   - gather from “customer”, write use cases
2. Think and discuss with others
   - write down preliminary list, update, 1 page description
3. Not too many, not too less
   - combine functionality, decrease complexity, make more general
4. Name them
   - meaningful, self-explanatory, consistent with existing API
5. Signature
   - methods signature for extensibility (generics/templates, interfaces, enums), no surprise when method is used

Process of API design

6. Prototype it
   - write code prototype using your API and update
7. Implement
   - discard method if unsure about functionality (add later)
   - hide implementation details
8. Document it
   - document every small thing, contracts, limitations, pre&post
   - don’t over explain, do not document implementation details

Process of API design

9. Test it
   - especially when designing SPI (Service Providers Interface)
   - write more plugins (one – NOK, two – difficult, three - OK)
10. Use it
    - use internally before wider release, update
11. Prepare for evolution and mistakes
    - displease everyone equally

General principles (1)

- API should do one thing and do it well
- As small as possible, but not smaller
  - if in doubt, leave function out (you can add, but not remove)
- Implementation details should not leak into API
- Minimalize accessibility (encapsulation)
  - make public what really needs to be
  - no public fields (attributes) except constants
- Make understandable names (self-explanatory, consistent, easy to read when used)

```
if (key.length() < 80)
     generateAlert("NSA can crack!");
```

General principles

```
/**
 * \brief Output = HMAC-SHA-512( hmac key, input buffer )
 *
 * \param key       HMAC secret key
 * \param keylen    length of the HMAC key
 * \param input     buffer holding the  data
 * \param ilen      length of the input data
 * \param output    HMAC-SHA-384/512 result
 * \param is384     0 = use SHA512, 1 = use SHA384
 */
void sha512_hmac(...);
```

- Document rigorously
  - JavaDoc, Doxygen
  - specify how function should be used
  - class: what instance represents
  - Method: contract between method and client
    - preconditions, postconditions, side effects
  - Parameters: who owns (ptr), units, format...
- Specific case of documentation are Annotations
  - e.g., Microsoft SAL, pre&post conditions

```
void* memcpy(void* destination, const void* source, size_t num);
void* memcpy(__out_bcount(num) void* destination,
             __in_bcount(num) const void* source, size_t num);
```

POLARSSL

```
/**
 * \brief          Output = HMAC-SHA-512( hmac key, input buffer )
 *
 * \param key      HMAC secret key
 * \param keylen   length of the HMAC key
 * \param input    buffer holding the  data
 * \param ilen     length of the input data
 * \param output   HMAC-SHA-384/512 result
 * \param is384    0 = use SHA512, 1 = use SHA384
 */
void sha512_hmac( const unsigned char *key, size_t keylen,
                  const unsigned char *input, size_t ilen,
                  unsigned char output[64], int is384 );
```

OPENSSL

```
unsigned char *HMAC(const EVP_MD *evp_md, const void *key, int key_len,
                    const unsigned char *d, size_t n, unsigned char *md,
                    unsigned int *md_len);
```

OpenSSL – HMAC ☹ (hard to understand)

```
//hmac.h
unsigned char *HMAC(const EVP_MD *evp_md, const void *key, int key_len,
                const unsigned char *d, size_t n, unsigned char *md,
                unsigned int *md_len);
```

```
//ossl_typ.h
typedef struct env_md_st EVP_MD;
```

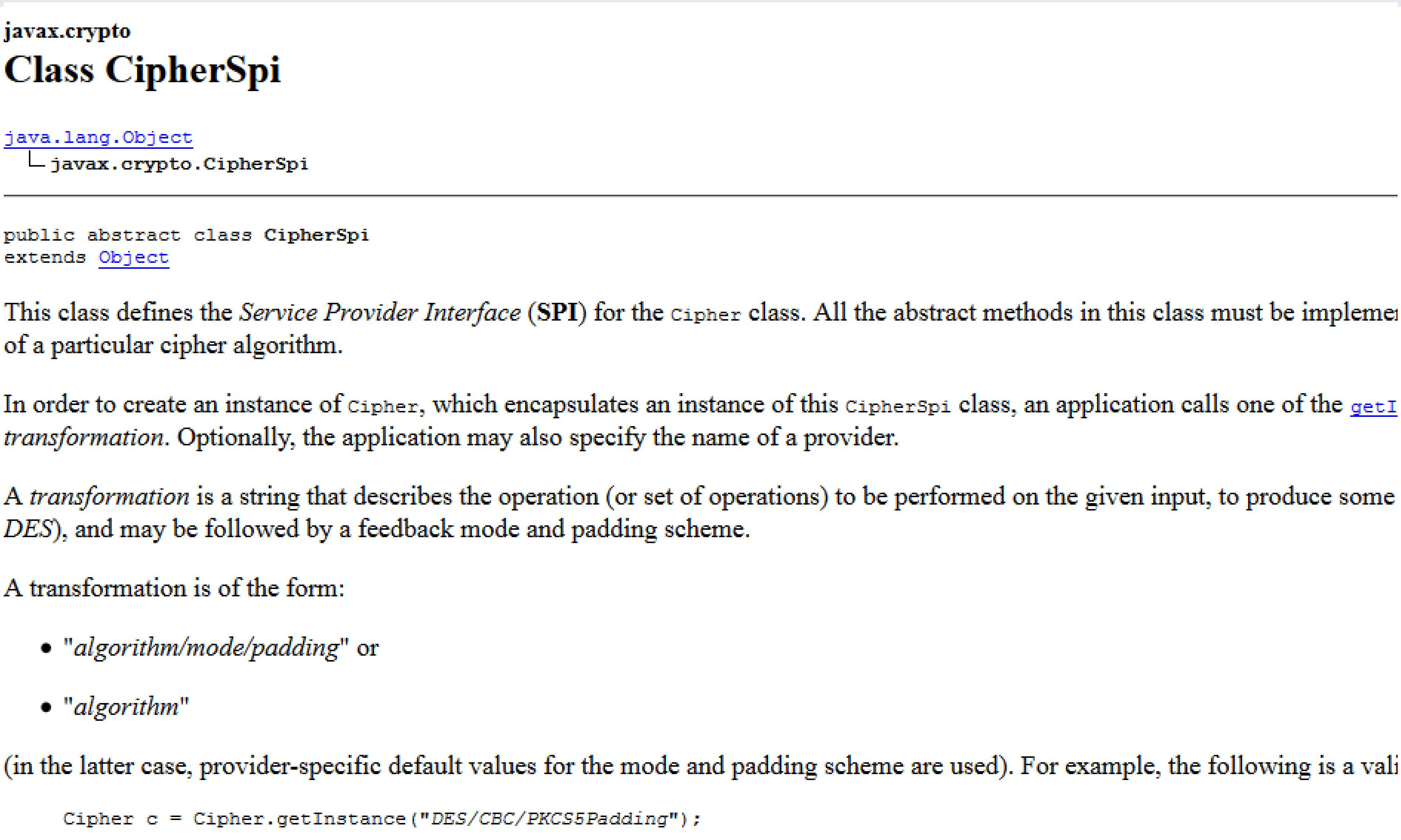
```
//envp.h
struct env_md_st
        {
        int type;
        int pkey_type;
        int md_size;
        unsigned long flags;
        int (*init)(EVP_MD_CTX *ctx);
        int (*update)(EVP_MD_CTX *ctx,const void *data,size_t count);
        int (*final)(EVP_MD_CTX *ctx,unsigned char *md);
        int (*copy)(EVP_MD_CTX *to,const EVP_MD_CTX *from);
        int (*cleanup)(EVP_MD_CTX *ctx);

        /* FIXME: prototype these some day */
        int (*sign)(int type, const unsigned char *m, unsigned int m_length,
                unsigned char *sigret, unsigned int *siglen, void *key);
        int (*verify)(int type, const unsigned char *m, unsigned int m_length,
                 const unsigned char *sigbuf, unsigned int siglen,
....
        } /* EVP_MD */;
```

Service providers interface (SPI)

- Situation:
  - Independent providers of libraries from developers of applications
  - Intermediate entity providing connection (e.g., OS)
  - Library is installed separately into OS
  - Application wants to use provided functionality
- How to support multiple providers of crypto functionality?

1. Providers implements same API (SPI)
   - E.g., MS CNG, JCE…
2. Provider registers with intermediate (e.g., OS)
3. Application can enumerate providers
   - E.g., java.security.Security.getProviders()
4. One provider is selected and used by application



javax.crypto

Class CipherSpi

```
java.lang.Object
  └ javax.crypto.CipherSpi
```

```
public abstract class CipherSpi
extends Object
```

This class defines the *Service Provider Interface* (**SPI**) for the `Cipher` class. All the abstract methods in this class must be impleme
of a particular cipher algorithm.

In order to create an instance of `Cipher`, which encapsulates an instance of this `CipherSpi` class, an application calls one of the `getI`
transformation. Optionally, the application may also specify the name of a provider.

A *transformation* is a string that describes the operation (or set of operations) to be performed on the given input, to produce some
DES), and may be followed by a feedback mode and padding scheme.

A transformation is of the form:

- "*algorithm/mode/padding*" or
- "*algorithm*"

(in the latter case, provider-specific default values for the mode and padding scheme are used). For example, the following is a vali

```
    Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

General principles

- Consider performance impact of API decisions
  - but be not influenced by implementation details
  - underlying performance issues will be fixed eventually, but API warping (for fixing past issue) remains
- Examples of bad performance decisions
  - need for frequent allocations and copy constructors
    - *Instead* pass arguments by reference or pointer
    - *Instead* use copy free functions
  - usage of mutable objects instead of immutable
    - *Instead* use const everywhere possible

Copy-free functions

- API style which minimizes array copy operations
- Frequently used in cryptography
  - we take block, process it and put back
  - can take place inside original memory array
- `int encrypt(byte array[], int startOffset, int length);`
  - encrypt data from *startOffset* to *startOffset* + *length;*
- Wrong(?) example:
  - `int encrypt(byte array[], int length, byte outArray[], int* pOutLength);`
  - note: C/C++ can still use pointers arithmetic
  - note: Java can’t (we need to create new array)

General principles

- Use static factory instead of class constructor
  - e.g., `javacardx.crypto` & `class::getInstance()`

```
import javacardx.crypto.*;

// CREATE DES KEY OBJECT
m_desKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
          KeyBuilder.LENGTH_DES, false);
// CREATE OBJECTS FOR CBC CIPHERING
m_encryptCipher = Cipher.getInstance(Cipher.ALG_DES_CBC_NOPAD, false);
m_decryptCipher = Cipher.getInstance(Cipher.ALG_DES_CBC_NOPAD, false);
// CREATE RANDOM DATA GENERATOR
m_secureRandom = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
// CREATE MD5 ENGINE
m_md5 = MessageDigest.getInstance(MessageDigest.ALG_MD5, false);
```

General principles

- Advantages of static factory over constructors
  - provides named "constructors"
  - can return null, if appropriate
  - can return an instance of a derived class, if appropriate
  - reduce verbosity when instantiating variables of generic/template types (no need to write type twice)

```
Map<String, list<String>>* m = new HashMap<String, List<String>>();
                              vs.
Map<String, list<String>> m = HashMap.newInstance();
```

  - allows immutable classes to use preconstructed instances or to cache instances (speed)
  - http://www.informit.com/articles/article.aspx?p=1216151

General principles

- Principle of last astonishment
  - user should not be surprised of API behavior
- Be careful with overloading
  - use different names for methods when having same number of arguments
  - same behavior for same (position of) arguments
- Fail fast – report error as soon as possible
  - failure in compile time is better
  - during runtime, first method invocation with bad state should fail
- Provide methods to obtain data elements from results provided originally in strings
  - do not force programmer to parse strings

General principles

- Use appropriate types for parameters and return types
  - use interface instead of class in method arguments
    - provides base for extensibility later
  - use most specific possible input parameter type
    - errors will move to compile time
  - don't use strings if better type exists (error prone, slow)
  - no floating point for monetary values (approximation)
  - use **`double`** instead of **`float`** (precision gain, only low performance impact)

General principles

- Use consistent parameter ordering (*src* vs. *desc*)

```
#include <string.h>
char *strcpy (char* dest, char* src);
void bcopy (void* src, void* dst, int n);
```

- Avoid long parameter lists
  - three or few parameters ideal (including default values)
    - mistake in filling arguments might be missed in compile
- When more parameters are required:
  - break method into more methods
  - or encapsulate multiple arguments into single class/struct

Example: Avoid long parameter lists

WIN32 API

```
HWND WINAPI CreateWindow(
  _In_opt_  LPCTSTR lpClassName,
  _In_opt_  LPCTSTR lpWindowName,
  _In_      DWORD dwStyle,
  _In_      int x,
  _In_      int y,
  _In_      int nWidth,
  _In_      int nHeight,
  _In_opt_  HWND hWndParent,
  _In_opt_  HMENU hMenu,
  _In_opt_  HINSTANCE hInstance,
  _In_opt_  LPVOID lpParam
);
```

QT API

```
QWidget window;
window.setWindowTitle("Window title");
window.resize(320, 240);
...
window.show();
```

Limit copy of sensitive values

- int set_key(Key_t key, pin_t seal_pin);
- int set_key(Key_t* key, pin_t* seal_pin);
- Pass by value makes another copy in memory
- Increase probability of information disclosure
- Memory can be assigned to another process
- Memory can be swapped to disk

Security API

- *“A security API allows untrusted code to access sensitive resources in a secure way.” G. Steel*
- Interface between different levels of trust
- Security API is designed to enforce a policy
  - certain predefined security properties should always hold
  - e.g., private key cannot be used before user is authenticated
- Security API is not equal to security protocols
  - but closely related
  - security protocol == short program how principals communicate
  - security API == set of short programs called in any order
- http://www.lsv.ens-cachan.fr/~steel/security_APIs_FAQ.html
- http://www.cl.cam.ac.uk/~rja14/Papers/SEv2-c18.pdf

Security API attack

- API attack is sequence of commands (function calls) which breach security policy of an interface
- “Pure” API attacks – only sequence of commands
  - e.g., key value is directly revealed
- “Augmented” API attacks – additional brute-force computations required
  - e.g., 128bits key value is exported under 56bits key

Security problems

- Interfaces get richer and more complex over time
  - – pressure from customers to support more options
  - – economic pressures towards unsafe defaults
  - – failures tend to arise from complexity, KISS
  - – hard to design secure API, even harder to keep it secure
- Leaks when trusted component talks to less trusted
  - – interface often leaks more information than anticipated by designer of trusted component

List of typical problems

- Unexpected command sequences
  - methods called in different order than expected
  - use method call fuzzer to test (next slide)
  - use automata-based programing to verify proper state
- Unknown commands
  - invalid values as method arguments
  - always make extensive input verification
  - use fuzzer to test

Method ordering fuzzer – the idea

1. Number uniquely every API function you like to test
2. Write **`switch`** construct with **`case X:`** calling function numbered with X
3. Generate (randomly) vector of numbers from range [1...# functions] with length of your choice
4. Wrap **`switch`** construct inside loop over vector's content
   - **`vector[i] == X`** will cause function call numbered with X
5. Prepare required variables for function arguments calls
   - outside loop for persistency of variable value from previous call
6. Log trace information for analysis (errors, argument input and output values, return values, exceptions raised...)

Functions ordering fuzzer - code

```
int main() {
    // Preparation of arguments
    int arg1 = 0;
    MyClass arg2;
    // Generation of function call sequence
    std::vector<int> fncCalls;
    generate(fncCalls.begin(), fncCalls.end(), RandomNumber);
    std::vector<int>::iterator iter;
    // Looping over generated sequence and execution of functions
    for (iter = fncCalls.begin(); iter != fncCalls.end(); iter++)
    // Log what is necessary (errors, input and output values...
        switch (*iter) {
        case fnc1: foo1(arg1, arg2); break;
        case fnc2: foo2(arg1, arg2); break;
        case fnc3: foo3(arg1); break;
        }
    }
    return 0;
}
```

Functions ordering fuzzer – the idea

- What if module is in the form of dynamically linked library?
  - you may link statically (manual labor)
  - you may obtain function pointers in runtime
    - LoadLibrary, GetProcAddress
    - list of available functions are usually documented
    - Full list of available functions can be obtained from *.lib
    - grep with replacement can generate necessary the code
    - some libraries provides functions for querying available interface (COM objects)

List of typical problems

- Commands in a wrong device mode
  - sensitive operation (e.g., Sign()) called without previous authentication
  - use methods order fuzzing to test
  - use automata-based programing to ensure proper state
- Existence of undocumented API
  - debugging API not removed (unintentionally)
  - security by obscurity (be aware of reverse engineering)
  - example: Crysalis Luna module (key extraction)
    - http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-592.pdf

List of typical problems

- Multiple different APIs to single component/storage
  - contracts within one set of API may be broken by second set
  - possible interleaving of function calls from different APIs
  - Example: Ultimaco HSM APIs
    - Microsoft world: CNG, CSP, EKMI
    - JCE, PKCS#11, OpenSSL
    - administration API's: IT monitoring SNMP
  - Example: IBM 4758 HSM APIs
    - IBM CCA, VISA EMV, PKCS#11...
    - IBM proprietary
- Attacks already used in the wild for large scale attacks
  - http://www.wired.com/threatlevel/2009/04/pins/

Best practices for designing security API

- Use API keys, not Username/Password
  - e.g., OAuth instead of Basic Auth
- Don’t use sessions (if possible)
  - build API as RESTful services
  - *“Each request from any client contains all of the information necessary to service the request, and any session state is held in the client.” REST Wikipedia*
  - check client input extensively

Best practices for designing security API

- Always use SSL when secure channel is required
  - or other suitable secure channel, don’t build one yourself
- Look at mature APIs for best practice examples
  - Foursquare, Twitter, and Facebook...
- Don’t use weak cryptographic algorithms
  - MD5, RC4... Old NSA saying: “Cryptanalysis always gets better. It never gets worse.”
- Don’t hardcode particular algorithm into API
  - and be prepared for change (e.g., BlockCipher interface instead of AES)

Formal verification of security API

- Harder than security protocol analysis
  - security API typically consist of tens of functions called in any order
  - security protocol only few messages executed in predefined sequence
- Initially applied only to small APIs, now better
- Now many interesting practical results
  - real attacks against PKCS#11 devices
  - Ubikey token API problem
  - PKCS#11 RSA’s token problem found
- Proofs of security within given model may be given
- http://www.lsv.ens-cachan.fr/~steel/security_APIs_FAQ.html

Formal verification of APIs

- Tookan tool
  - http://secgroup.ext.dsi.unive.it/projects/security-apis/tookan/
  - probe PKCS#11 token with multiple function calls
  - automatically create formal model for token
  - run model checker and find attack
  - try to execute attack against real token
- No single “best” tool (Avispa, Proverif…)
- A Generic API for Key Management
  - http://www.lsv.ens-cachan.fr/~steel/genericapi/
- International Workshop on Analysis of Security APIs
  - http://www.lsv.ens-cachan.fr/~steel/asa/

References: Designing API

- How to Design a Good API and Why it Matters (Google)
  - http://lcsd05.cs.tamu.edu/slides/keynote.pdf
  - video: http://www.infoq.com/presentations/effective-api-design
- Designing Good API & Its Importance
  - http://www.slideshare.net/imyousuf/designing-good-api-its-importance
- What is good API design
  - http://richardminerich.com/2012/08/what-is-good-api-design/
- Fundamentals of good API designing
  - http://www.javacodegeeks.com/2013/05/fundamentals-of-good-api-designing.html

References: Security API

- Mike Bond's Ph.D. thesis (attacks on HSMs)
  - http://www.cl.cam.ac.uk/~mkb23/research/Thesis.pdf
- Graham Steel's Security API FAQ
  - http://www.lsv.ens-cachan.fr/~steel/security_APIs_FAQ.html
- Ross Anderson's API attacks (Security Engineering)
  - http://www.cl.cam.ac.uk/~rja14/Papers/SEv2-c18.pdf

AUTOMATA-BASED PROGRAMMING

Automata-based style program

- Program (or its part) is though of as a model of finite state machine (FSM)
- Basic principles
  1. Automata state (explicit designation of FSM state)
  2. Automation step (transition between FSM states)
  3. Explicit state transition table (not all transitions are allowed)
- Practical implementation
  - imperative implementation (switch over states)
  - object-oriented implementation (encapsulates complexity)
- https://en.wikipedia.org/wiki/Automata-based_programming

Example: SimpleSign applet

- Simple smart card applet for digital signature
  - user must verify UserPIN before private key usage for signature Sign() is allowed
  - unblock of user pin allowed only after successful AdminPIN verification
- Imperative solution:
  - sensitive operation (Sign()) is wrapped into condition testing successful PIN verification
  - more conditions may be required (PIN and < 5 signatures)
  - same signature operation may be called from different contexts (SignHash(), ComputeHashAndSign())

SimpleSign – imperative solution

```
void SignData(APDU apdu) {
    // ...
    // INIT WITH PRIVATE KEY
    if (m_userPIN.isValidated()) {
        // INIT WITH PRIVATE KEY
        m_sign.init(m_privateKey, Signature.MODE_SIGN);

        // SIGN INCOMING BUFFER
        signLen = m_sign.sign(apdubuf, ISO7816.OFFSET_CDATA,
                              (byte) dataLen, m_ramArray, (byte) 0);

        // ... SEND OUTGOING BUFFER
    }
    else ISOException.throwIt(SW_SECURITY_STATUS_NOT_SATISFIED);

    // ...
}
```

Test of required condition

Execution of sensitive operation

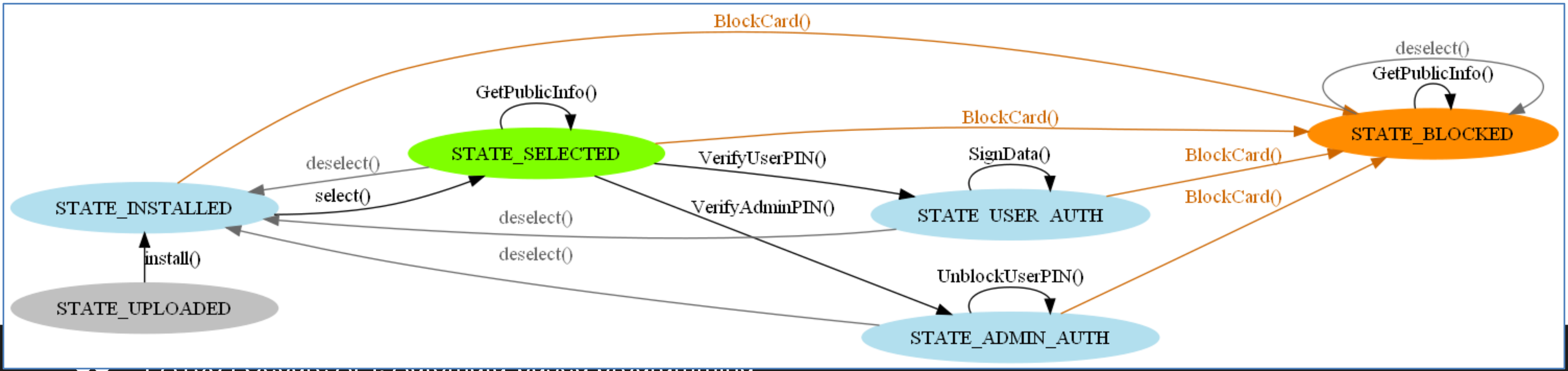
SimpleSign – automata-based solution

- Mental model $\rightarrow$ .dot format (human readable)
  - > Graphviz visualization (visual inspection)
  - > source code generated for state check and transition check
  - > input for formal verification (state reachability)
- Easy to extend by new states
  - source code is generated again
- More robust against programming mistakes and omissions

Example: states for smart card applet

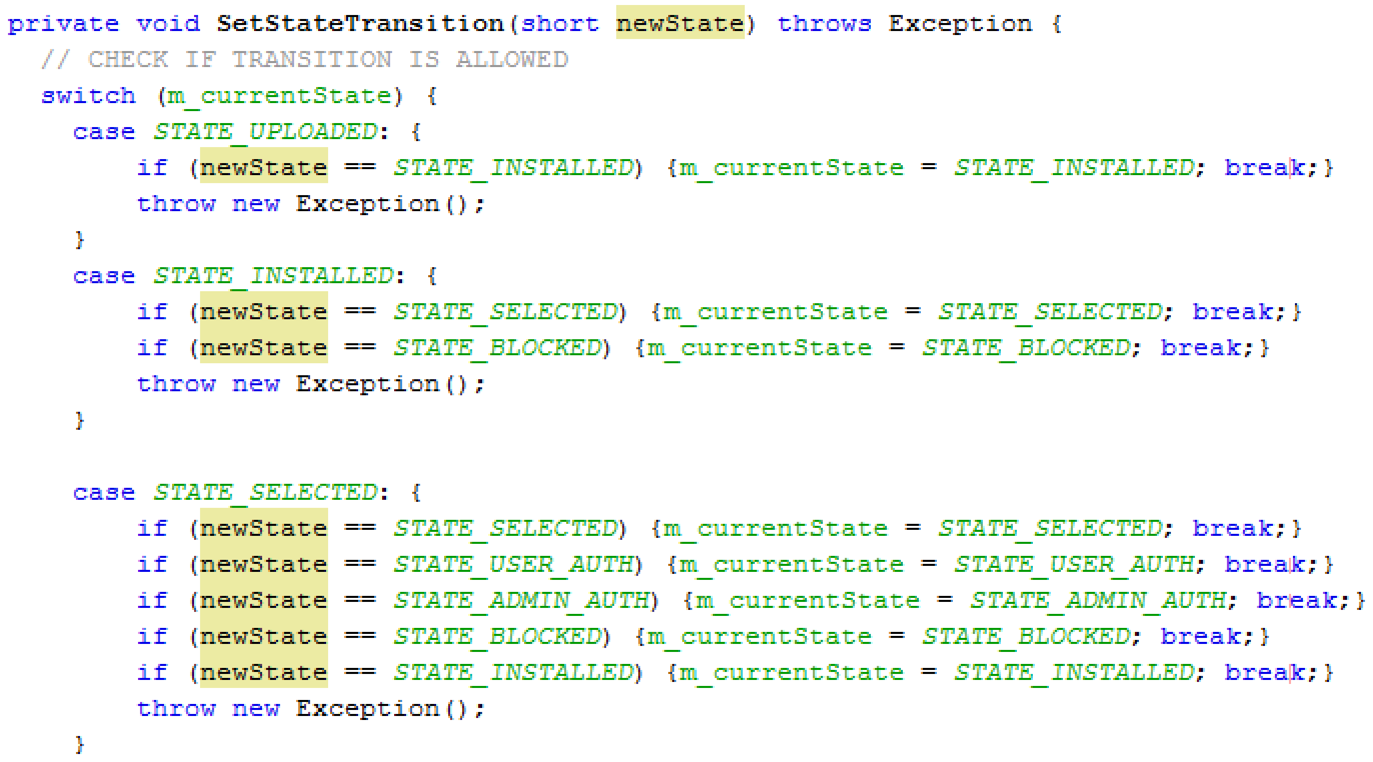
```
digraph StateModel {
rankdir=LR;
size="6,6";
node [shape =ellipse color=green, style=filled];
{ rank=same; "STATE_UPLOADED";"STATE_INSTALLED";}
"STATE_INSTALLED";
"STATE_UPLOADED";
"STATE_UPLOADED" -> "STATE_INSTALLED" [label="install()"];
{ rank=same; "STATE_SELECTED";}
"STATE_SELECTED";
{ rank=same;"STATE_USER_AUTH";"STATE_ADMIN_AUTH";}
"STATE_USER_AUTH" ;
"STATE_ADMIN_AUTH" ;

"STATE_INSTALLED" -> "STATE_SELECTED" [label="select()" color="black"  fontcolor="black"];
"STATE_SELECTED" -> "STATE_USER_AUTH" [label="VerifyUserPIN()" color="black" fontcolor="black"];
"STATE_SELECTED" -> "STATE_ADMIN_AUTH" [label="VerifyAdminPIN()" color="black"  fontcolor="black"];
...
```



Is transition allowed between given states?

- E.g., is allowed to change state directly from STATE_INITIALIZED to STATE_ADMIN_AUTH?



```
private void SetStateTransition(short newState) throws Exception {
  // CHECK IF TRANSITION IS ALLOWED
  switch (m_currentState) {
    case STATE_UPLOADED: {
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
        throw new Exception();
    }
    case STATE_INSTALLED: {
        if (newState == STATE_SELECTED) {m_currentState = STATE_SELECTED; break;}
        if (newState == STATE_BLOCKED) {m_currentState = STATE_BLOCKED; break;}
        throw new Exception();
    }

    case STATE_SELECTED: {
        if (newState == STATE_SELECTED) {m_currentState = STATE_SELECTED; break;}
        if (newState == STATE_USER_AUTH) {m_currentState = STATE_USER_AUTH; break;}
        if (newState == STATE_ADMIN_AUTH) {m_currentState = STATE_ADMIN_AUTH; break;}
        if (newState == STATE_BLOCKED) {m_currentState = STATE_BLOCKED; break;}
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
        throw new Exception();
    }
```

Is function call allowed in present state?

- E.g., do not allow to use private key before UserPIN was verified

```
private void checkAllowedFunction(int requestedFunction) {
    switch (requestedFunction) {
      case FUNCTION_VerifyUserPIN:
        if (m_currentState == STATE_SELECTED) break;
        _OperationException(EXCEPTION_FUNCTIONEXECUTION_DENIED);

      case FUNCTION_SignData:
        if (m_currentState == STATE_USER_AUTH) break;
        _OperationException(EXCEPTION_FUNCTIONEXECUTION_DENIED);
...
```

Sign data only when in STATE_USER_AUTH

How to react on incorrect state transition

- Depends on particular application
  - create error log entry
  - throw exception
  - terminate process
  - ...
- Error message should not reveal too much
  - side-channel attack based on error content
  - information about (non-)existing user name
  - structure of database tables / directories
  - reason for error…

SimpleSign – additional functionality

- New functionality is now required:
  - signature allowed also after verification of AdminPIN
- Changes required in imperative solution:
  - add additional condition before every Sign()
  - when called from multiple places, developer may forgot to include conditions everywhere
  - not easy to realize, what conditions are required from existing code
- Changes required in automata-based solution:
  - add new state transition (STATE_ADMIN_AUTH <-> SignData())
  - generate new transition tables etc.

SUMMARY

Summary

- Designing good API is hard
  – follow best practices, learn from well-established APIs
- Designing security API is even harder
- Automata-based programming
  – make more robust state and transition validation
  – good to combine with visualization and automatic code generation

Questions