PA193 - Secure coding principles and practices

Automata-based programming
Designing good and secure API

Petr Švenda *svenda@fi.muni.cz* *@rngsec*
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on Cryptography and Security

Link for comments (thank you!)
https://drive.google.com/file/d/1txIBFe5zi40GIMJsas_6e9OW0koDQK8y/view?usp=sharing

Phase II

- Configure Github Actions to run tests automatically
- Start digitally signing your commits
- Start the implementation
  - You can use only standard library
  - By the end of this phase, you should have:
    - Basic encoding/decoding functionality passing test vectors.
    - No need to provide user interface yet.
- Prepare 3-4 A4 report
  - Project design
  - Current progress
  - Encountered obstacles
- Deadline Monday 14. 3. 2022 16:00

Phase III

- Finalize the implementation
  - Try to identify any vulnerabilities with analysis tools
  - Release the final binary build with a digital signature (GPG)
- Prepare and record a presentation of your project (5-7 minutes)
  - Structure of the project
  - Encountered obstacles and solutions
  - Used analysis tools
  - How can the tool be used
    - (Quick guide for the other team in Phase IV)
- Discussion of the presentation
- Assignment of other team projects for the next phase
- Deadline Monday 11. 4. 2022 16:00

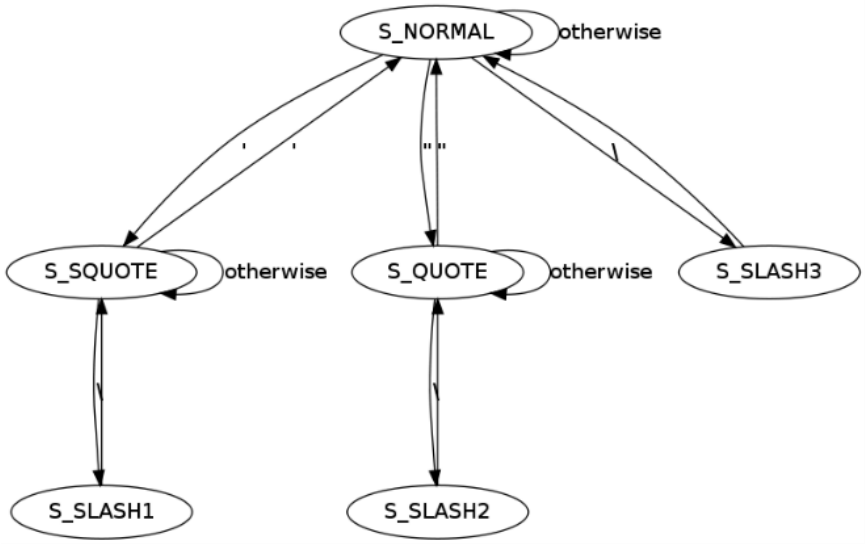
Overview

- Lecture:
  - How to write good & secure API
  - automata-based programming
- Labs
  - Write small security API
  - Model inner state with automata-based design

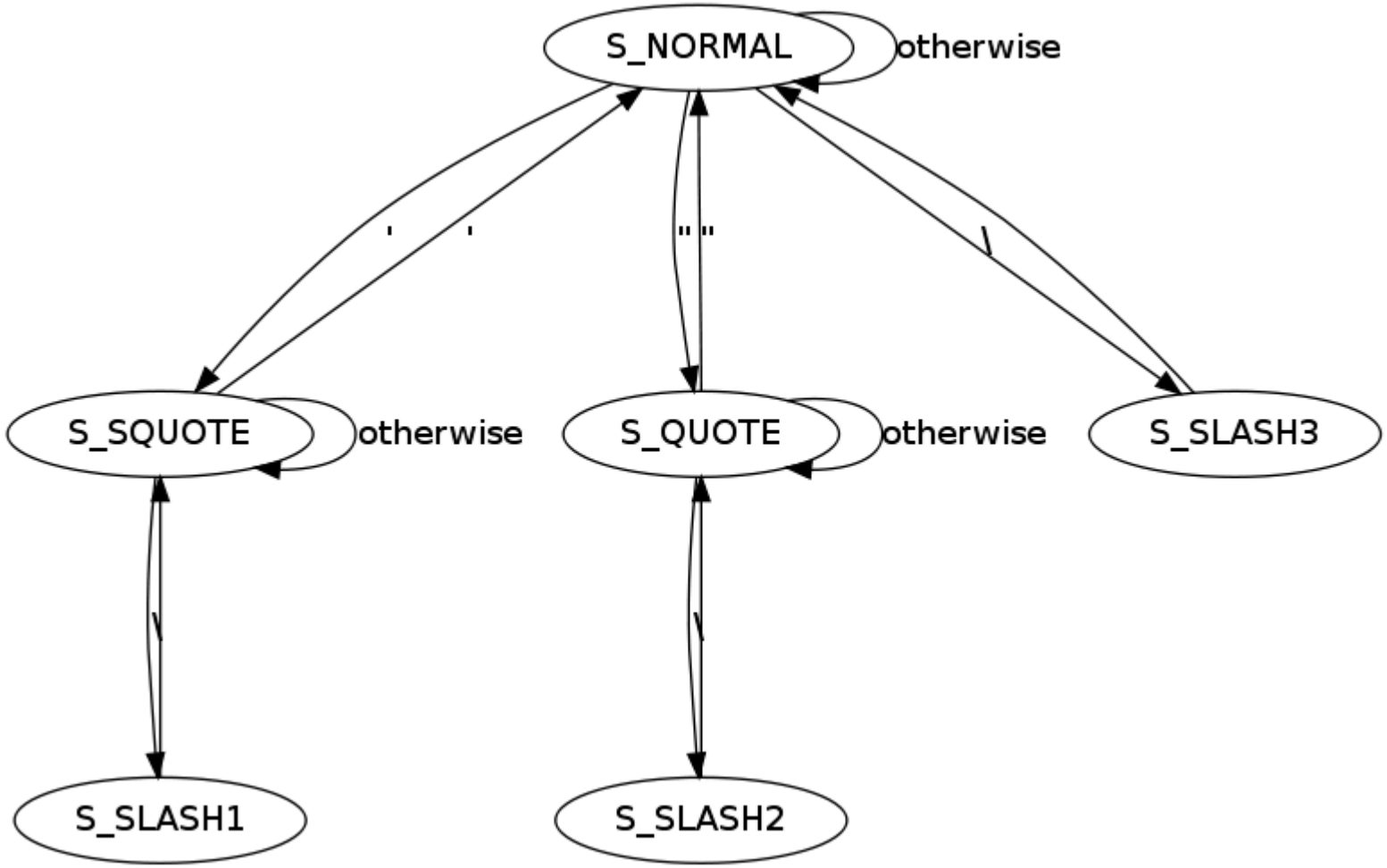
AUTOMATA-BASED PROGRAMMING

Automata-based style program

- Program (or its part) is though of as a model of finite state machine (FSM)
- Basic principles
  - Automata state (explicit designation of FSM state)
  - Automata step (transition between FSM states)
  - 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: detect correct “ in input string



```c
int main(void) {
    t_states state = S_NORMAL;
    char symbol;

    init_data(&data, "\";\" \\\"&  Test 'string  &\\' really ' & ; ", 1024);
    printf("Test string: %s\n", data.input_string);

    symbol = get_next_char();

    while (symbol != EOF) {
        printf("[state:] %02d [symbol:] %c [output:] ", state, symbol);

        switch (state) {

            case S_NORMAL:
                switch (symbol) {
                    case ';':
                        put_next_char('\\');
                        break;
                    case '&':
                        put_next_char('\\');
                        break;
                    case '"':
                        state = S_QUOTE;
                        break;
                    case '\'':
                        state = S_SQUOTE;
                        break;
                    case '\\':
                        state = S_SLASH3;
                        break;
                }
                break;


...
```

```c
            case S_SQUOTE:
                switch (symbol) {
                    case '\'':
                        state = S_NORMAL;
                        break;
                    case '\\':
                        state = S_SLASH1;
                        break;
                }
                break;

            case S_QUOTE:
                switch (symbol) {
                    case '"':
                        state = S_NORMAL;
                        break;
                    case '\\':
                        state = S_SLASH2;
                        break;
                }
                break;

            case S_SLASH1:
                state = S_SQUOTE;
                break;
            case S_SLASH2:
                state = S_QUOTE;
                break;
            case S_SLASH3:
                state = S_NORMAL;
                break;
            case S_SPACE:
                break;
        }

        /* There is no 'output symbol suppression'
           needed in our example */
        put_next_char(symbol);
        printf("\n");
        /* .. and prepare the next char waiting */
        symbol = get_next_char();
    } /* End while */
```

Example: SimpleSign applet

- Simple smart card applet for digital signature
  - Op1: user must verify UserPIN before private key usage for signature is allowed
  - Op2: unblock of user pin allowed only after successful AdminPIN verification
- Imperative solution:
  - sensitive operation (Sign()) wrapped into condition testing successful PIN verification
  - more conditions may be required (PIN and less than 5 signatures)
  - same signature operation may be called from different contexts (SignHash(), ComputeHashAndSign())

REQUIREMENT:
USER MUST VERIFY PIN BEFORE SIGNATURE OPERATION (USING PRIVATE KEY) IS PERFORMED

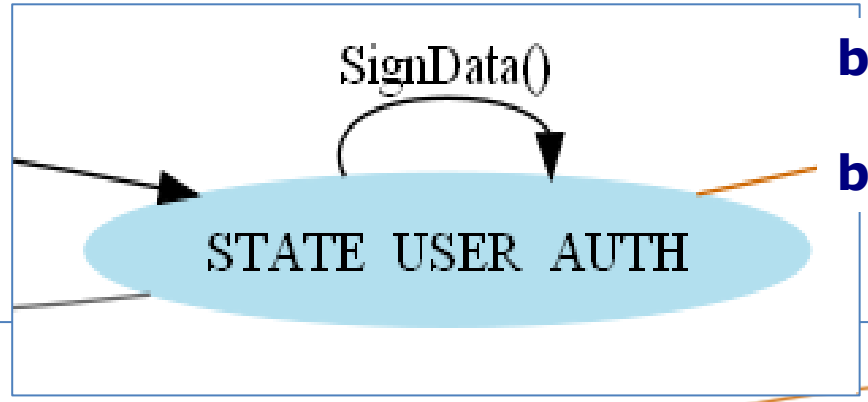
SimpleSign – imperative solution

```
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);

    // …
}
```

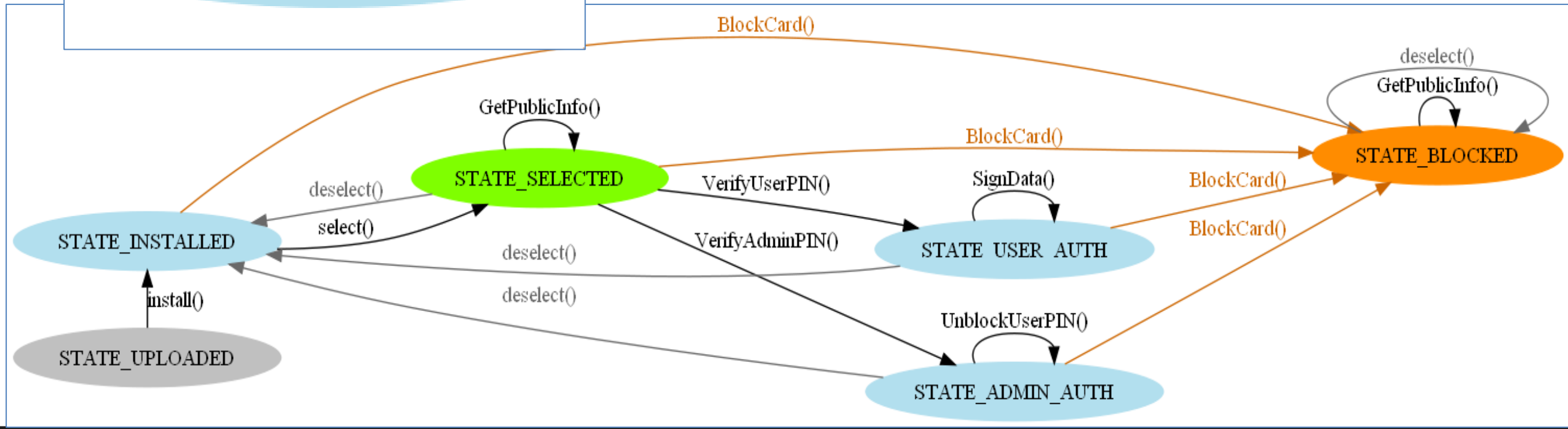
Test of required condition(s)

Execution of sensitive operation

SimpleSign – automata-based programming

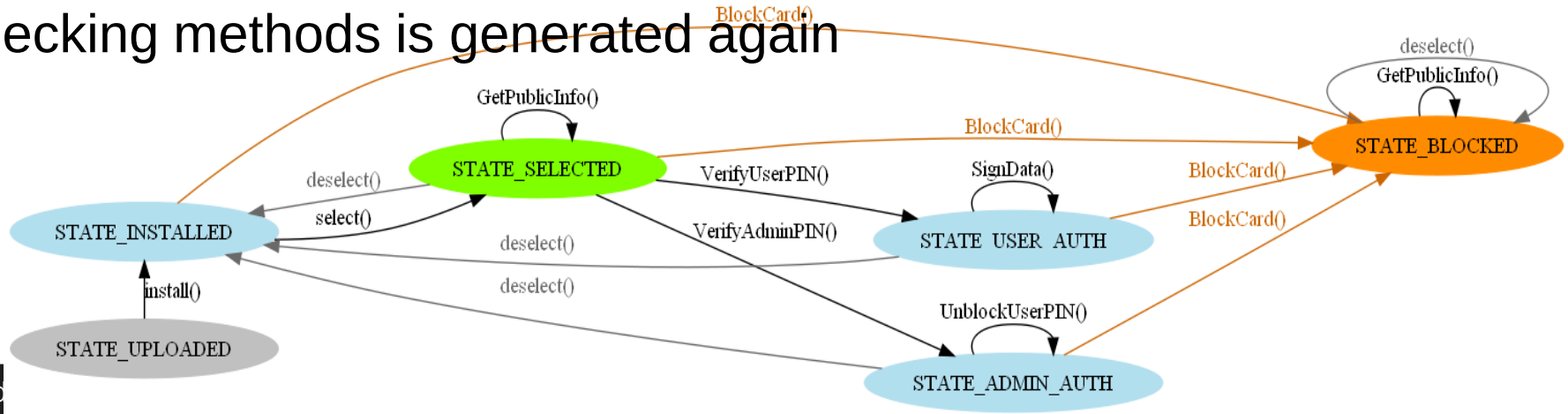



```
bool isFunctionAllowed(int FNC, int CURRENT_STATE);

bool isTransitionAllowed(int CURRENT_STATE, int NEW_STATE);
```



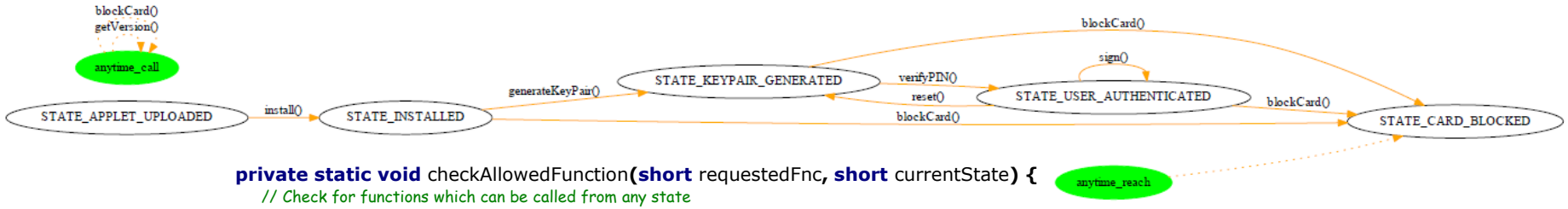
SimpleSign – automata-based solution

1. Mental model → human readable format (yaml, xml, dot…)
   - Graphviz visualization (allows visual inspection)
   - Source code generated for state check and transition check
   - Possible input for formal verification (state reachability)
2. More robust against programming mistakes and omissions
3. Easy to extend by new states
   - source code for checking methods is generated again



```yaml
# GENERAL CONFIGURATION  #
config:
  package_name: yourpackage

# SPECIAL STATES AND FNCS #
states_special:
  anytime_call: [getVersion, blockCard]
  anytime_reach: [STATE_CARD_BLOCKED]

# STATE TRANSITIONS #
states_transitions:

  STATE_APPLET_UPLOADED:
    install: STATE_INSTALLED

  STATE_INSTALLED:
    generateKeyPair: STATE_KEYPAIR_GENERATED
    blockCard: STATE_CARD_BLOCKED

  STATE_KEYPAIR_GENERATED:
    verifyPIN: STATE_USER_AUTHENTICATED
    blockCard: STATE_CARD_BLOCKED

  STATE_USER_AUTHENTICATED:
    sign: STATE_USER_AUTHENTICATED
    reset: STATE_KEYPAIR_GENERATED
    blockCard: STATE_CARD_BLOCKED

# SECONDARY STATE CHECKING #
secondary_state_check:

  SECURE_CHANNEL_ESTABLISHED: [generateKeyPair, verifyPIN, sign]

  CHANNEL_NONE: [install, blockCard, reset]
```

Source yaml file: examples\simple_state_model.yml




```java
private static void checkAllowedFunction(short requestedFnc, short currentState) {
   // Check for functions which can be called from any state
   switch (requestedFnc) {
      // case FNC_someFunction:  return;   // enable if FNC_someFunction can be called from any state (typical for cleaning instructions)
      case FNC_blockCard:  return;
      case FNC_getVersion:  return;
   }

   // Check if function can be called from current state
   switch (currentState) {
      case STATE_APPLET_UPLOADED:
         if (requestedFnc == FNC_install) return;
         ISOException.throwIt(SW_FUNCTINNOTALLOWED); // if reached, function is not allowed in given state
         break;
      case STATE_INSTALLED:
         if (requestedFnc == FNC_blockCard) return;
         if (requestedFnc == FNC_generateKeyPair) return;
         ISOException.throwIt(SW_FUNCTINNOTALLOWED); // if reached, function is not allowed in given state
         break;
      case STATE_KEYPAIR_GENERATED:
         if (requestedFnc == FNC_blockCard) return;
         if (requestedFnc == FNC_verifyPIN) return;
         ISOException.throwIt(SW_FUNCTINNOTALLOWED); // if reached, function is not allowed in given state
         break;
      case STATE_USER_AUTHENTICATED:
         if (requestedFnc == FNC_blockCard) return;
         if (requestedFnc == FNC_reset) return;
         if (requestedFnc == FNC_sign) return;
         ISOException.throwIt(SW_FUNCTINNOTALLOWED); // if reached, function is not allowed in given state
         break;
      default:
         ISOException.throwIt(SW_UNKNOWNSTATE);
         break;
   }
}
```

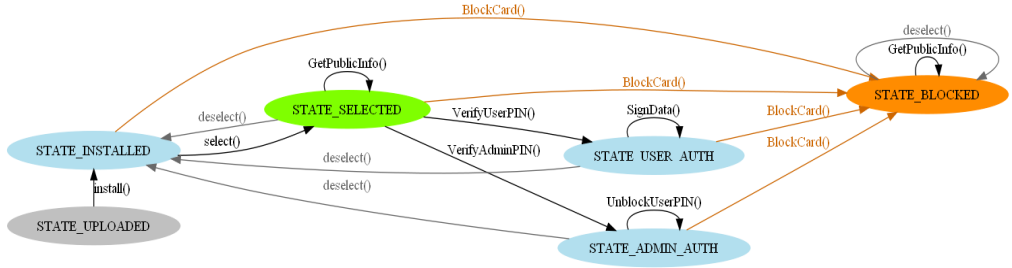
anytime_reach

Examples from https://github.com/petrs/state_enforcer

```
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_UPLOADED to STATE_ADMIN_AUTH?



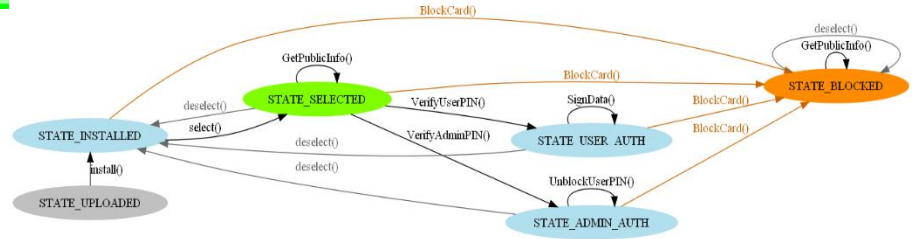

```
private void SetStateTransition(short newState) throws Exception {
  // CHECK IF TRANSITION IS ALLOWED
  switch (m_currentState) {
    case STATE_UPLOADED: {
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
        throw new Exception();
    }
    case STATE_INSTALLED: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

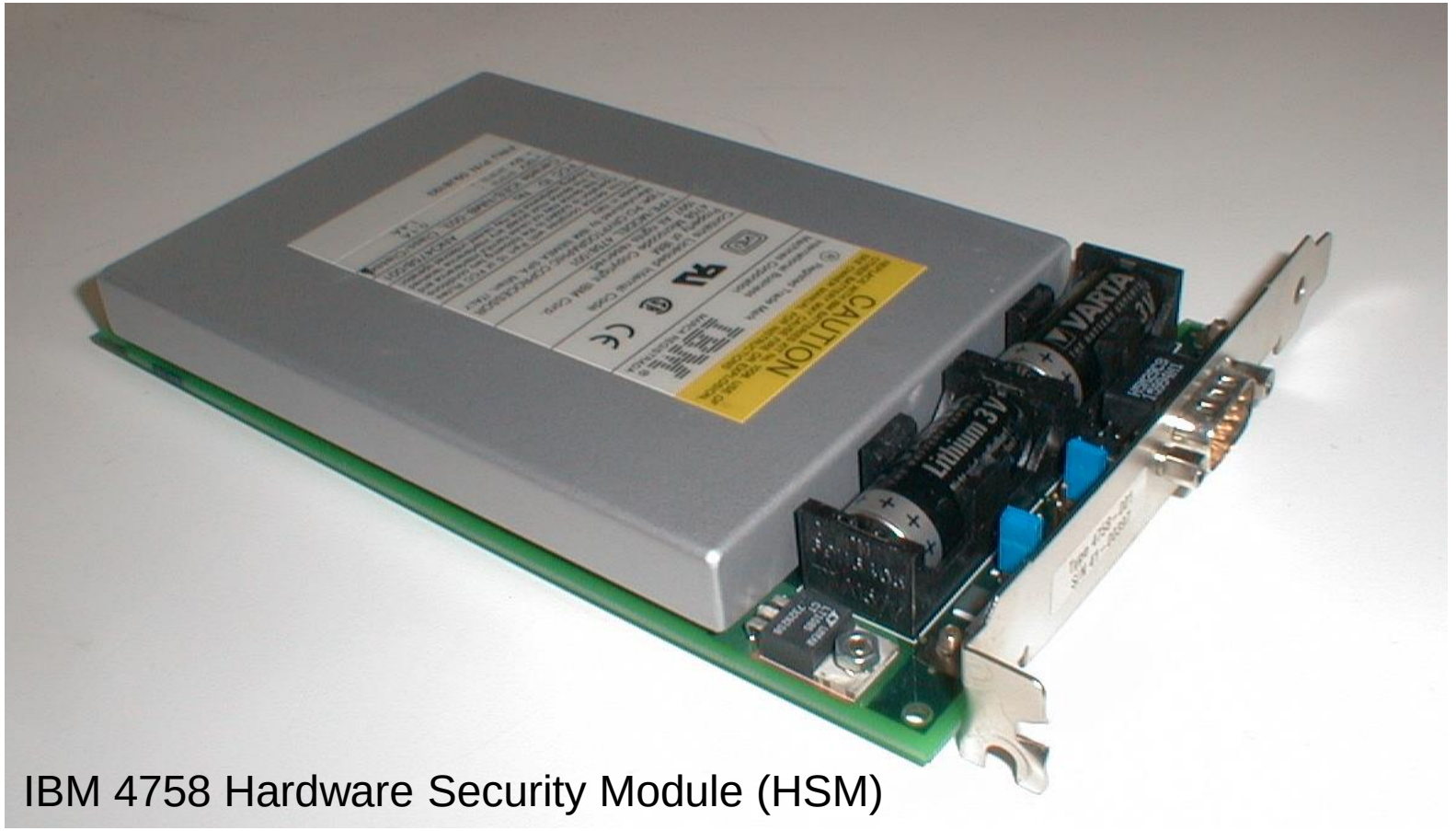
- Depends on application
  - Create error log entry
  - Throw exception
  - Terminate process
  - ...
- Advantage: reaction is handled at two places only (checking methods)
  - Systematically same for all cases
- (Remainder: Error message should not reveal too much)
  - Leads to side-channel attack based on error content
  - Leads to padding oracle attacks…

SimpleSign – new requirement, additional functionality

- Programs almost always change in time
- If new functionality is required:
  - E.g., 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 forget to include conditions to all places
  - 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.

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
    - 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 exposes its functionality via API
  - E.g., encrypt supplied data with key generated inside HSM
- HSM is trusted, accessed by not-so-trusted applications
  - HSM's API serves as wall between different levels of trust
  - Intentionally limits visibility and access

API

Security API

APPLICATION PROGRAMMING INTERFACE

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 used by others, 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, Web REST API...
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

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, structure of json for webapi...
- 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 (little/big endian…)
- API != ABI, but both are necessary

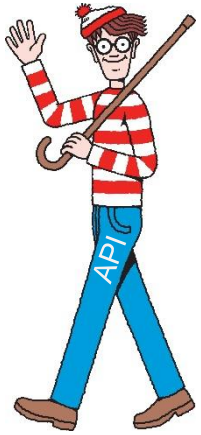
Language (in)dependent API

- Language dependent API
  - API available only for one particular language (C printf(), Java System.out.println())
  - ABI is relevant (calling convention, memory layout…)
- Language independent API
  - Not restricted to particular languages (=> cross-language compatibility)
  - E.g., Web API based on HTTP/REST/JSON
- Language bindings
  - Bridge between particular language and library/OS API
  - E.g., library implemented in C, but called from Python
  - Additional API in target language with small proxy code
  - E.g., python ctypes package: https://realpython.com/python-bindings-overview/

Web API

- Web API = API used to invoke method on web server
  - Usually via HTTP(S) with REST
  - Language independent API
- E.g., Red Hat CVE API
  - *GET https://access.redhat.com/hydra/rest/securitydata/cve/CVE-2016-3706.json*
- Application programming interface key (API key)
  - Code supplied by program calling an API
  - Identifies program, developer, user…
  - Can be used to control usage (e.g., limit requests…)

Quiz – where is API stored?



- Language: C
  - API: Functions listed in header files (*.h)
- Language: C++
  - API: public methods of class
  - API: public methods of abstract class
- Language: Java
  - API: public methods of class
  - API: methods of interface
- Web API
  - API: HTTP/REST requests, response in JSON format

PRINCIPLES OF GOOD API

Credits: Joshua Bloch

- *Joshua Bloch, How to Design a Good API and Why it Matters (Google)*
  - *https://research.google/pubs/pub32713/*
  - *video: http://www.infoq.com/presentations/effective-api-design*
- *Reading/watching is highly recommended*
- *Many ideas taken from his presentation*
  - *demonstrated on cryptographic libraries by myself*

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

- *https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf*

Process of API design (Joshua Bloch)

1. Gather requirements
2. Start with short specification (1 page)
3. Write API early and often
4. Test and use your API
   - especially when designing SPI (Service Providers Interface)
   - write more plugins (one – NOK, two – difficult, three - OK)
5. Prepare for evolution and mistakes
   - displease everyone equally

What is Service Provider Interface?

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 implemer 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");
```

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

Process of API design (Arnab Biswas)

1. Understand the requirements
   - gather from “customer”, write use cases
2. Think and discuss with others
   - write down preliminary list, update
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

http://www.javacodegeeks.com/2013/05/fundamentals-of-good-api-designing.html

Process of API design (Arnab Biswas)

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
9. Test it
10. Use it
    - use internally before wider release, update

General principles - encapsulation

- API should do one thing and do it well
- As small as possible, but not smaller
  - if in doubt, leave function out (you can (usually) add, but not remove)
- Implementation details should not leak into API
  - try to hide as much as possible from user
- 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!");
```

Which one you like more? Why?

mbedTLS

```
/**
 * \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);
```

General principles – behave as expected

- 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
    - (mistake in use based on the argument types)
  - same behavior for same (position of) arguments (C memcpy() vs. Java arraycopy())
- 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 as strings (do not force programmer to parse strings)

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 */;
```

Sensitive data (keys) in memory

- What is the difference?

```
int set_key(Key_t key, pin_t seal_pin);
                    vs.
int set_key(Key_t* key, pin_t* seal_pin);
```

- Try to limit copies of sensitive data in memory
  - potential unintended disclosure (memory, swap…)
  - Pass by value requires more memory erases
- Look for platform's existing security API to handle sensitive data
  - E.g., Android Keystore API (https://developer.android.com/guide/topics/security/cryptography)

General principles - documentation

- 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, Java annotations…

```
/**
 * \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* 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);
```

General principles

- Consider “never remove, only add” strategy
  - backward compatibility
- Flexibility and learning curve / usability
  - Continuous trade-off

General principles - performance

- 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
    - pass arguments by reference or pointer
    - use copy free functions
  - usage of mutable objects instead of immutable
    - use **`const`** everywhere possible
  - need for frequent re-coding (byte[] -> string -> byte[])

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 – static factory

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

```
import javacardx.crypto.*;

// CREATE AES KEY OBJECT
m_aesKey = (AESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
          KeyBuilder.LENGTH_AES_256, false);
// CREATE OBJECTS FOR CBC CIPHERING
m_encryptCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD, false);
m_decryptCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD, false);
// CREATE RANDOM DATA GENERATOR
m_secureRandom = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
// CREATE SHA256 ENGINE
m_sha256 = MessageDigest.getInstance(MessageDigest.ALG_SHA_256, false);
```

General principles – static factory

- Advantages of static factory over constructors
  - provides named "constructors“ (getInstance, buildKey)
  - 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 – arguments types

- 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)
    - but also more user-friendly - tradeoff
  - no floating point for monetary values (approximation)
  - use **`double`** instead of **`float`** (precision gain, only low performance impact)

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
);
```

Which one you like more?
Why?

QT API

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

General principles - parameters

- Avoid long parameter lists
  - three or fewer 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
- Use consistent parameter ordering (src vs. desc)

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

Bad example from C stdlib

Security API

- *"A security API allows untrusted code to access sensitive resources in a secure way." Graham 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 (entities) 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 API – problems in time

- 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 possible when trusted component talks to less trusted
  - interface often leaks more information than anticipated by designer of trusted component (error codes, timing…)

Security API - typical problems

- Unexpected command sequences
  - methods called in different order than expected
  - use method call fuzzer to test
  - use automata-based programing to verify proper state
- Unexpected input data
  - 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 can be obtained from *.lib
    - grep with replacement can generate necessary the code
    - some libraries provides functions for querying available interface (COM objects)

Security API - 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

Security API - 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/

Security API: best practices

- 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 (including state)
- Supply methods for secure erase of sensitive data
  - Do not force developer to figure out how to make this right

Security API: best practices

- Use TLS when secure channel is required
  - or another 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 getting better
  - Very helpful as bug hunting tool
- Many interesting practical results
  - real attacks against PKCS#11 devices
  - 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/

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

References: API design for REST

- https://blog.apigee.com/detail/api_design_ruminating_over_rest
- http://www.stormpath.com/blog/how-secure-api-tips-rest-json-developers
- https://www.stormpath.com/blog/secure-your-rest-api-right-way

CODE ANNOTATIONS

Motivation for making annotations

- More semantics of code available for checker
  - Capture otherwise missed bugs
- More explicit documentation of code/API
  - Ideally automatically testable
  - Problems captured in compile time
- Compliancy requirements
  - Driver signature by Microsoft, mandatory for 64b Windows
- …

Microsoft SDL C/C++ static checker

- Problems not found by PREfast checker by default
- Achievable via source-code annotation language (SAL)
  - check of return value
  - argument must be not NULL
  - string must be terminated
  - length of data read / written into buffer
- Additional requirements are added to declaration of function, structure... via (non-standard) keywords
- Validity of such requirements are checked by PREfast
  - in pre-state (before fnc call) & in post-state (after fnc call)

SAL – examples (in and out buffer)

```
// read from buffer with size equal to length
int readData( void *buffer, int length );
int readData(_In_reads_(length) void *buffer, int length );

// writes specified amount (length) of data into buffer
int fillData( void *buffer, int *length );
int fillData(_Out_writes_all_(length) void *buffer, const int length );
```

Source-code annotation language (SAL)

- Microsoft's annotation language
- Improves code checking done by MS static checker PREfast
- MS Annotations
  - http://msdn.microsoft.com/en-us/library/ms182032.aspx
  - http://msdn.microsoft.com/en-us/library/hh916382.aspx
  - http://msdn.microsoft.com/en-us/security/gg675036
- Additional requirements are added to declaration of function, structure... via (non-standard) keywords
- Validity of such requirements are checked by PREfast
  - in pre-state (before fnc call) & in post-state (after fnc call)

SAL – basic terms

- Element is valid if contains explicitly assigned value
  - Item of allocated array with unassigned value is invalid
- Valid in *pre-condition* (before function is called)
  - Annotation typically starts with In_xxx
- Valid in *post-condition* (when function ends)
  - Annotation typically starts with Out_xxx
- Number of specified *bytes* vs. *items*
  - Default is number of items, bytes if _bytes_ added
  - Number of elements valid

SAL functions basics

http://msdn.microsoft.com/en-us/library/hh916383.aspx

Category	Parameter Annotation	Description
Input to called function	**_In_**	Data is passed to the called function, and is treated as read-only.
Input to called function, and output to caller	**_Inout_**	Usable data is passed into the function and potentially is modified.
Output to caller	**_Out_**	The caller only provides space for the called function to write to. The called function writes data into that space.
Output of pointer to caller	**_Outptr_**	Like **Output to caller**. The value that's returned by the called function is a pointer.

- Optional version of arguments
  - argument might be NULL
  - **`_In_opt_, _Out_opt_...`**
  - function must perform check before use
    - Otherwise PREFast will report error

SAL functions basics II.

- Pointer type annotations
- `_Outptr_`
  - should not be NULL
  - should be initialized
- _Outptr_opt_
  - can be NULL, must be checked

```
void salDemo(_In_ int* pInArray, _Outptr_ int** ppArray) {
}

int main(int argc, char* argv[]) {
    int* pArray = NULL;
    int* pArray2 = NULL;
    if (strcmp(argv[1], "alloc") == 0) {pArray = new int[5];}
    salDemo(pArray, &pArray2);

    return 0;
}
```

- PREfast analysis

```
test.cpp(34): warning : C6101: Returning uninitialized memory '*ppArray'. A successful
   path through the function does not set the named _Out_ parameter.
test.cpp(49): warning : C6387: 'pArray' could be '0':  this does not adhere to the
   specification for the function 'salDemo'.
test.cpp(49): warning : C6001: Using uninitialized memory '*pArray'.
```

SAL annotations – much more

- Annotations of functions
  - http://msdn.microsoft.com/en-us/library/hh916382.aspx
- Structs and classes can be also annotated
  - http://msdn.microsoft.com/en-us/library/jj159528.aspx
- Locking behavior for concurrency can be annotated
  - http://msdn.microsoft.com/en-us/library/hh916381.aspx
- Whole function can be annotated
  - http://msdn.microsoft.com/en-us/library/jj159529.aspx
  - `_Must_inspect_result_`
- Best practices
  - http://msdn.microsoft.com/en-us/library/jj159525.aspx

SAL – examples (in and out buffer)

```
// read from buffer with size equal to length
int readData( void *buffer, int length );
int readData(_In_reads_(length) void *buffer, int length );

// writes specified amount (length) of data into buffer
int fillData( void *buffer, int *length );
int fillData(_Out_writes_all_(length) void *buffer, const int length );

// writes into buffer maxLength at max, but possibly less and modifies also length argument
int fillData( void *buffer, const int maxLength, int *length );
// Check if no more then maxLength and *length is written, also check range of length
int fillData( __Out_writes_to_( maxLength, *length ) void *buffer,
        const int maxLength, _Out_range_(0, maxLength-1) int *length );

// read AND write from buffer
int readWriteData( void *buffer, int length );
int readWriteData(_Inout_updates_(length) void *buffer, int length );
```

SAL – examples (pointers, strings)

```
// pass argument by value foo pointer
int getInfo( struct thing *thingPtr );
// value is used as input and output => _Inout_
int getInfo( _Inout_ struct thing *thingPtr );

// pass C null-terminated strings
int writeString( const char *string );
// must be null terminated string > _In_z_
int writeString( _In_z_ const char *string );
```

Using SAL with / without PREfast

- Cross-platform code
  - Compiled with MSVC for Windows (SAL is supported)
  - Compiled with GCC for Linux (SAL not supported)
- Issue: SAL annotations makes GCC compilation to fail
- Solution
  - Create custom `#define` for most common SAL annotations
  - Define as empty if not compiled with MSVC
  - Can be also tuned when SAL annotation changes itself
- Peter Gutmann' Experiences with SAL/PREfast
  - http://www.cs.auckland.ac.nz/~pgut001/pubs/sal.html

Wrapping defines for SAL

```
#if defined( _MSC_VER ) && defined( _PREFAST_ )
  #define IN_BUFSIZE          _In_reads
  #define IN_BUFSIZE_OPT      _In_reads_opt
  #define OUT_BUFSIZE         _Out_writes
  #define OUT_BUFSIZE_OPT     _Out_writes_opt
  #define OUT_PTR             _Outptr_
  #define OUT_PTR_OPT         _Outptr_opt_
#else
  #define IN_BUFSIZE( size )
  #define IN_BUFSIZE_OPT( size )
  #define OUT_BUFSIZE( max, size )
  #define OUT_BUFSIZE_OPT( max, size )
  #define OUT_PTR
  #define OUT_PTR_OPT
#endif /* VC++ with source analysis enabled */
```

Modification of http://www.cs.auckland.ac.nz/~pgut001/pubs/sal.html

Annotations for GCC/LLVM

- Deputy
  - Not active any more ☹, last update 2006?
  - http://www.stanford.edu/class/cs295/asgns/asgn5/www/
- Clang Static Analyzer
  - http://clang-analyzer.llvm.org/annotations.html
  - Only few annotations

Splint (is simple to use?)

- **SAL version**

```
void strcpy( _Out_z char *s1, _In_z const char *s2 );
```

- **Splint version**

```
void /*@alt char * @*/ strcpy(
  /*@unique@*/ /*@out@*/ /*@returned@*/ char *s1, char *s2 )
  /*@modifies *s1@*/ /*@requires maxSet( s1 ) >= maxRead( s2 ) @*/
  /*@ensures maxRead( s1 ) == maxRead( s2 ) @*/;
```

SUMMARY

Summary

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

Questions ?