Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

ASSG 4 - Public Key Cryptography Standards (PKCS #11)

RSA KEY MANAGEMENT + ENCRYPTION

Prepared For:
The University of Washington Bothell

CSS527 - CRYPTOGRAPHY

Winter, Q2 - 2020

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

Index:

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 0 - PREFACE

SECTION 0.0 - Assignment Description

Assignment Description
From Canvas: https://canvas.uw.edu/courses/1368253/assignments/5205153
(Abbreviations appended and altered to reflect local terminology)

*Design and build a Proof of Concept (**PoC**) illustrating the possibility of interfacing with the (virtual Hardware Security Module (**vHSM**) module you built in A(ssignment)3 using PKCS #11.*

Please refer to primary document at: **[LINK]**
Also refer to the errata at: **[LINK]**

*Select an Application Programming Interface (**API**) wrapper based on the programming language of your choice. Implement a PoC app(lication) illustrating simple integration with the vHSM module.*

Deliverables:
1) A report introducing PKCS #11 and its applications.
2) A summary of the PoC design.
3) Source code and screenshots of your working program.

Terminology:
Proof of Concept - Evidence, typically derived from an experiment or pilot project, which demonstrates that a design concept, business proposal, etc., is feasible.
API Wrapper - An entity that encapsulates another item, in this case an Application Programming Interface (API). Wrappers are used for one or both of two primary purposes - to convert data to a compatible format or to hide the complexity of the underlying entity using abstraction.
Integration - The combination of elements with others so as that they operate as a whole.

SECTION 0.1 - PKCS / PKCS #11 Description

PKCS
From Wikipedia: https://en.wikipedia.org/wiki/PKCS

"PKCS stands for "Public Key Cryptography Standards". These are a group of public-key cryptography standards devised and published by RSA Security LLC, starting in the early 1990s. The company published the standards to promote the use of the cryptography techniques to which they had patents, such as the RSA algorithm, the Schnorr signature algorithm and several others."

PKCS #11
From Wikipedia: https://en.wikipedia.org/wiki/PKCS_11

"The PKCS #11 standard defines a platform-independent API to cryptographic tokens, such as hardware security modules (HSM) and smart cards, and names the API itself "Cryptoki" (from "cryptographic token interface" and pronounced as "crypto-key" - but "PKCS #11" is often used to refer to the API as well as the standard that defines it).

The API defines most commonly used cryptographic object types (RSA keys, X.509 Certificates, DES/Triple DES keys, etc.) and all the functions needed to use, create/generate, modify and delete those objects."

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 1 - PKCS #11 - AN INTRODUCTION

SECTION 1.0 - Background and History (as of 2020.03.15)

The **Public Key Cryptography Standard (PKCS)** is a set of standards documents released by RSA Security LLC beginning the early 1990s as a set of instructions for how to implement a variety of cryptographic technologies to which RSA held patents. **PKCS#11** is the 11th such standard, first released as version (v) 1.0 in 1994.

In 2012.12, at the v2.20 release, RSA Security transitioned development and maintenance of the PKCS #11 standard to the **OASIS PKCS 11 Technical Committee** [LINK]. Since that time, work has continued; the latest approved version (**v2.40**) having been released on 2016.05 [LINK]. An upcoming update (v3.0) is currently under review. The OASIS committee has published, as of 2019.12, its latest PKCS#11 specification [LINK]. This timeline is briefly summarized as follows:

Timeline

```
1994.01 - PKCS#11 project initialized
1995.04 - v1.0
1996    -
1997.12 - v2.01
1998    -
1999.12 - v2.10
2000    -
2001.01 - v2.11
2002    -
2003    -
2004.06 - v2.20
2005.12 - Amendments 1+2 - One-time Password Tokens, Cryptographic Token Key Initialization Protocol
2006    -
2007.01 - Amendment 3 - Additional mechanisms
2008    -
2009.09 - v2.30 draft (not published)
2010    -
2011    -
2012.12 - Transition to OASIS announced
2013.03 - Transition to OASIS complete - Technical committee inaugural meeting
2014    -
2015.04 - v2.40
2016.05 - v2.40 Errata 01
2017    -
2018    -
2019    -
2020    - v3.0 in progress (present)
```

Fig. 1.0.1 - Timeline of PKCS #11 Standard Development

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

To provide context to PKCS#11, The full range of PKCS standards are briefly described as follows:

STANDARD	NAME	DESCRIPTION
PKCS#1	RSA Cryptography Standard	Defines the mathematical properties and format of RSA public and private keys, and the basic algorithms and encoding/padding schemes for performing RSA encryption, decryption, and producing and verifying signatures.
PKCS#2	(withdrawn / abandoned)	- N/A -
PKCS#3	Diffie–Hellman Key Agreement Standard	A cryptographic protocol that allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel.
PKCS#4	(withdrawn / abandoned)	- N/A -
PKCS#5	Password-based Encryption Standard	Key derivation functions with a sliding computational cost, used to reduce vulnerabilities to brute force attacks.
PKCS#6	Extended-Certificate Syntax Standard	Defines extensions to the old v1 X.509 certificate specification.
PKCS#7	Cryptographic Message Syntax Standard	Used to sign and/or encrypt messages under a PKI. Used also for certificate dissemination (for instance as a response to a PKCS #10 message). Formed the basis for S/MIME. Often used for single sign-on.
PKCS#8	Private-Key Information Syntax Standard	Used to carry private certificate keypairs (encrypted or unencrypted).
PKCS#9	Selected Attribute Types	Defines selected attribute types for use in PKCS #6 extended certificates, PKCS #7 digitally signed messages, PKCS #8 private-key information, and PKCS #10 certificate-signing requests.
PKCS#10	Certification Request Standard	Format of messages sent to a certification authority to request certification of a public key.
PKCS#11	**Cryptographic Token Interface**	**An API defining a generic interface to cryptographic tokens. Often used in single sign-on, public-key cryptography and disk encryption systems.**
PKCS#12	Personal Information Exchange Syntax Standard	Defines a file format commonly used to store private keys with accompanying public key certificates, protected with a password-based symmetric key. PFX is a predecessor to PKCS #12.
PKCS#13	(withdrawn / abandoned)	- N/A -
PKCS#14	(withdrawn / abandoned)	- N/A -
PKCS#15	Cryptographic Token Information Format Standard	Defines a standard allowing users of cryptographic tokens to identify themselves to applications, independent of the application's Cryptoki implementation (PKCS #11) or other API.

Fig. 1.0.2 - PKCS Standards by Number and Name.

SECTION 1.1 - PKCS#11 Structural Overview

The PKCS#11 standard describes a standardized interoperability layer between an arbitrary application and a PKCS#11 compliant HSM. As such, application calls are made to a PKCS#11 driver, which in turn converts these application calls to standard-compliant queries. The PKCS#11 queries are then passed to the hardware module.

By standardizing the Driver-HSM interface, the hardware module may be freely replaced with alternate HSM implementations, and the application developers require no special knowledge of HSM operation to perform useful work with it.

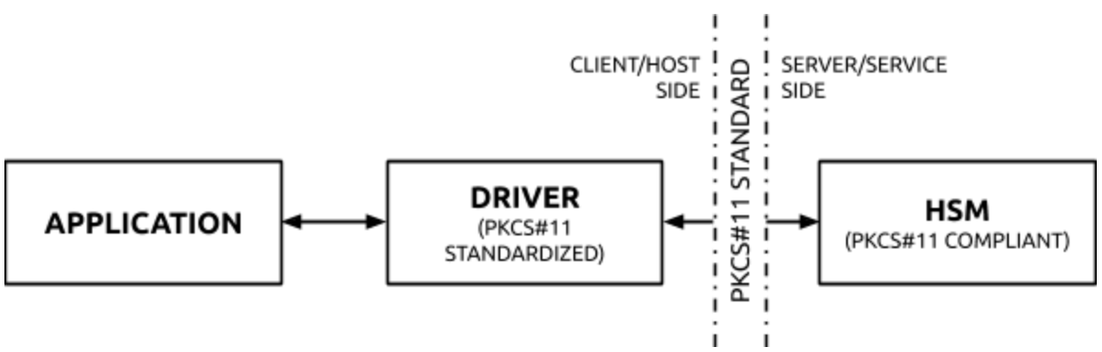


Fig. 1.1.0 - Schematic diagram of the relationship between application, driver, the HSM, and the PKCS#11 standard.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 1.2 - PKCS#11 Interface (the Cryptoki Library)

A compliant PKCS#11 implementation requires three layers (hereafter referred to as the 'Application', 'Driver', and 'vHSM'), all of which must communicate with one another to accomplish the goal of moving an Application request to the HSM, then - if necessary - moving the HSM product of that request back to the Application.

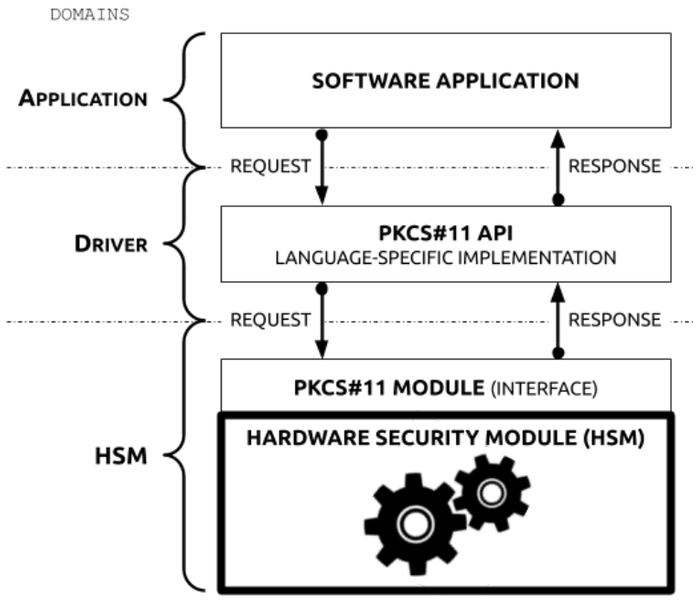


Fig. 1.2.0 - Diagram of the PKCS#11 stack from Application to vHSM and back. Cryptographic operation within the HSM is ideally entirely opaque even to the interfacing module.

For the modules to communicate, each pair of layers must maintain certain communication guarantees between the sender and receiver. These terms detail how and in what format information is to be passed and constitute the API between elements.

An HSM vendor will typically provide a driver for the operating system - this driver conforms to the Cryptoki API and will take the form of a shared library (DLL for Windows, SO for Linux). The vendor-provided driver will communicate with the HSM using a proprietary protocol, but since the API is strictly defined by the Cryptoki standard, any client application can make use of the HSM.

Programming languages are typically provided with libraries that allow for control and data exchange between an application and the Cryptoki API. For example, for the Java language, Oracle provides an abstraction layer (the “Java Cryptography Architecture” **[LINK]**) and interface (the “Sun PKCS#11 Provider” **[LINK]**). With this interface, a Java application may make use of an HSM (or other device/service that provides a PKCS#11 driver) without needing the vendor-specific operational details of a particular HSM implementation.

The proper invocation and management of DLLs is beyond the scope of this document. For the purposes of a proof-of-concept, some means of communicating the requisite call elements must be provided between the Driver and vHSM. For a Java application such as ours, the typical process for developing an application that communicates with the HSM is as follows:

1. Create a C or C++ shared library (DLL or SO) that can securely communicate with the HSM and exercise its functions
2. Create a C or C++ API that matches the Cryptoki standard, and “wire” this API to the portion that communicates with the HSM
3. Create a Java Native Interface (“JNI”) that maps Java classes and methods to the Cryptoki API, OR use Java Native Access (“JNA”) and C compatibility objects provided by JNA to call the Cryptoki API directly
4. Invoke methods provided by the JNI or JNA layer in your Java application

For a Java application to use the Cryptoki API natively is a large project in and of itself - our decision was to simulate calls to the API rather than develop (and troubleshoot!) these multiple layers for this assignment.

However, appropriate naming conventions are used and may be found in Appendix 0.4 (Variable Prefixes) and Appendix 0.5 (Function Name-Number Mappings).

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 1.3 - PKCS#11 HSM

Per Section 1.2 (PKCS#11 Interface), it may be noted that the PKCS#11 standard does not apply to the internal operation of the vHSM itself. The vHSM should ideally provide an opaque set of cryptographic services while a PKCS#11 compliant module converts standardized incoming requests into vHSM-local function requests. Conversely, the module should also translate internal products into outward facing signals (described as 'handles'). Additionally, full PKCS#11 compliance should be considered as a separate issue from vHSM vulnerability, since a security module may exercise fully compliant communication to the outside world while still failing to protect its contents from attack.

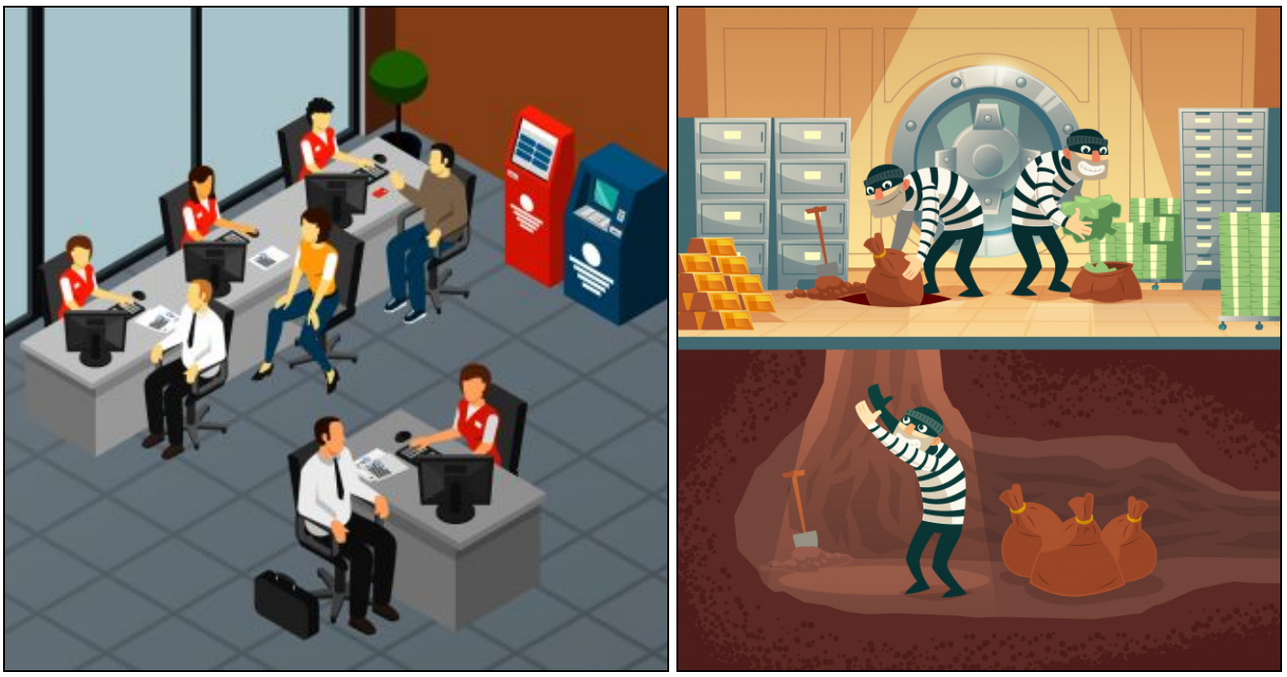


Fig. 1.3.0 - Illustrating the difference between a compliant PKCS#11 interface (left) and vHSM vulnerability (right). Compliance describes a standardized set of forward-facing rules which abstracts the back-of-house operation from hosts. While these rules are designed to facilitate or improve security, they are insufficient to guarantee it.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 2 - PROOF-OF-CONCEPT DESIGN SUMMARY

SECTION 2.0 - Proof-Of-Concept: Structural Overview

The **Proof-of-Concept (PoC)** is comprised of three operational layers and two interfaces:

1) The **Application** - A computer program calling the Cryptoki interface

1.5) An Application Programming Interface between the Application and the Driver - Describing a message standard for requests and responses between the Application and Driver.

2) The **Driver** - Representing the Interface defined in the PKCS#11 standard, a module that maps Application requests to PKCS#11 compliant messages for consumption by the Hardware Security Module.

2.5) A set of PKCS#11 compliant message and responses passed between the Driver and vHSM.

3) The (virtual) hardware security module (**vHSM**) - Formally termed the 'Cryptographic Device'. Consumes messages (cryptographic requests) from the Driver and, at its own discretion, conducts the cryptographic operation. These operations may or may not generate a return.

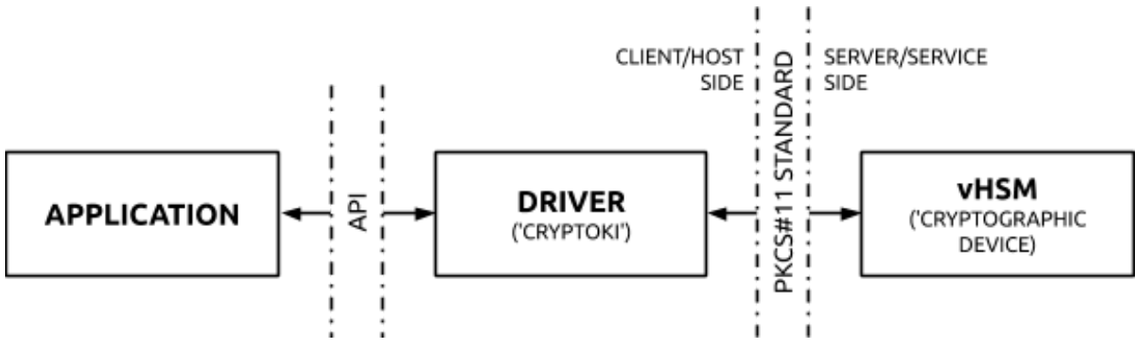


Fig. 2.0.0 - Schematic diagram of the Proof-of-Concept architecture. Note that it is identical in structure to the PKCS#11 architectural overview details in Fig. 1.1.0.

For PoC purposes, the three modules shall be run in the same operating environment as three discrete program instances, termed the **Application**, the **Driver**, and the **vHSM**.

Fig. 2.0.1 - Three program instances, driven by three discrete blocks of code provide the necessary logic to generate a request, produce a cryptographic product (token), and pass a handle to that token back to the requester.

Communication between the Application ⇔ Driver and Driver ⇔ vHSM shall be accomplished using a shared file system. In actuality, this service would likely be supported by network communications, transport protocols, or shared memory, though implementation of secure signal transport is outside the scope of this exercise.

Fig. 2.0.2 - Schematic representation of the use of text files for the transfer of messages.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 2.1 - Proof-Of-Concept: Request Chain

2.1.1 - Specification for Application WRITE to Driver:

An application may request the creation of an RSA keypair (**Keypair**). This action:

1) Accepts a **Key Password** from the User
2) Hashes the Key Password to create a **Key Hashword** as follows:

$$(Key\ Hashword) = (\ SHA\text{-}256(\ Key\ Password\ )\ )$$

3) Binds the Key Hashword to the **User's Identity** and action to create an **Application Request** as follows:

$$(Application\ Request) = (User\ Identity) + (Key\ Generation\ Request) + (Key\ Password)$$

> ***Note:***
> While hashing the password at the application level prior to transit would be preferable, the vHSM implementation at present provides no means of generating an RSA keypair based on password hash. This functionality likely exists, but is outside the scope of the PKCS#11 standard and is included as an item of future development.

4) The Application Request is then forwarded to the Driver by means of writing to file.

Fig. 2.1.1.0 - Request authorship from the Application to the Driver.

2.1.2 - Specification for Driver READ from Application, WRITE to vHSM:

The Driver may ingest and process an Application Request message as follows:

1) The Request is read in from file
2) The relevant fields are parsed
3) The Driver performs a table lookup to map the Application request name to a C Function Name.

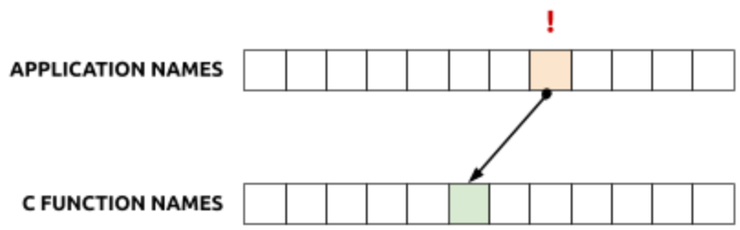


Fig. 2.1.2.1 - Mapping of Application name conventions to C Function compliant naming conventions.

> ***Note:***
> While the mapping herein is bijective, in actuality this may not be the case. An arbitrary number of Application function calls may map to single C Function Names and, inversely, not all C Function Names may have Application equivalents.

4) The Driver may then author one of two PKCS#11 Compliant Request **(Compliant Request)** to the vHSM:

$$(Driver\ Request) = (C\ Function\ Name) + (Function\ Number) + (0..*\ Optional\ Arguments)$$

Fig. 2.1.2.2 - Compliant Request authorship from the Driver to the vHSM.

> ***Note:***
> Under normal protocol operation, a vHSM may perform additional steps such as initialization or validation to prepare memory space and inform the Driver of the vHSM status. These steps, however, constitute additional interface operations and are not implemented as a component of this exercise.
>
> The user and their permission to perform the operation is presumed to have already been validated by the vHSM with the user identity serving as the secure token granting this permission.

2.1.3 - Specification for vHSM READ from Driver:
The vHSM, in turn, may ingest and process compliant requests from the Driver as follows:

1) The Compliant Request is read in from file

Fig. 2.1.3.1 - Compliant Request ingest from Driver by the vHSM

2) The request fields are parsed and held by the vHSM pending processing (Section 2.2)
3) The vHSM may elect to perform a legality check of the provided fields

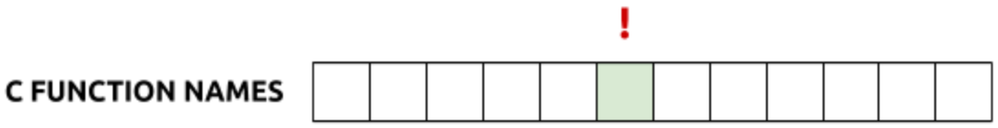


Fig. 2.1.3.2 - The C Function Name may be validated by way of dictionary lookup in the vHSM.

This concludes the description of the Request Chain from Application to vHSM.

SECTION 2.2 - Proof-Of-Concept: vHSM Endpoint Request Processing

Once a valid request has been received by the vHSM, it may elect to process the request. For the PoC, support is implemented for the C_GenerateKeyPair operation. Provided, then, that the received C Function Name matches this and other relevant data is provided, the vHSM will proceed to generate an RSA keypair based on the logic from Assignment 3 - vHSM.

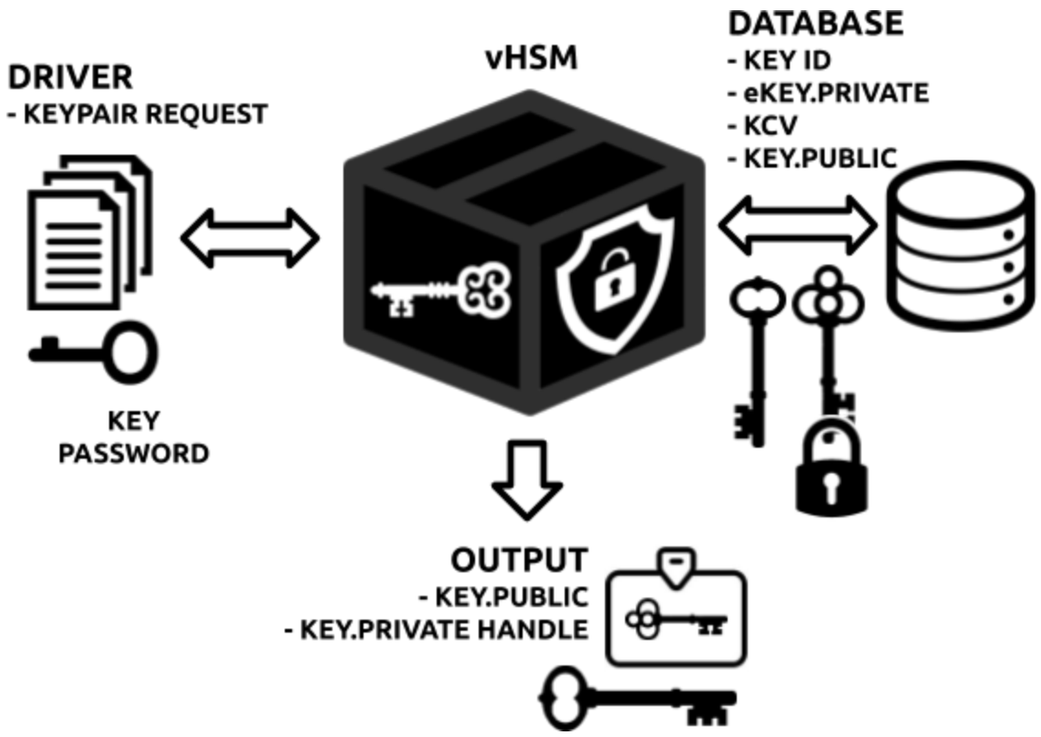


Fig. 2.2.1 - A schematic diagram of the keypair creation process

The notable alterations from the prior vHSM version include:
- Identity and key creation information is **provided via message parsing**, rather than user login and inputs
- The **Public Key** is held for insertion to a response message rather than printed to console
- The keyID is used as the vHSM's **Handle** to the private key

This concludes the description of the Endpoint Request Processing by the vHSM.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 2.3 - Proof-Of-Concept: Response Chain

The response chain, being almost precisely the inverse of the request chain, is herein described in an abbreviated fashion.

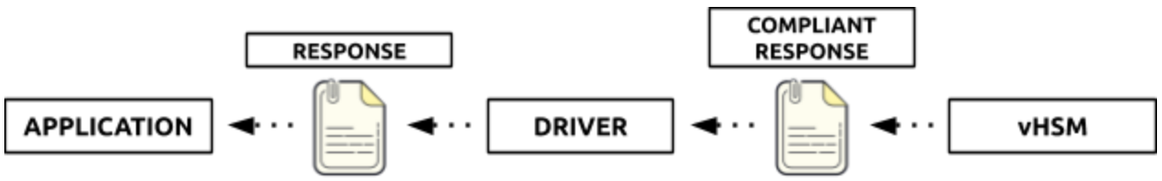


Fig. 2.3.0 - Schematic representation of the response chain

1) The vHSM prepares a **Compliant Response** using the products (a handle and public key) in format:

$$(Driver\ Response) = (C\ Function\ Name) + (Function\ Number) + (Handle) + (Key.Public)$$

2) The Compliant Response is passed to the Driver.
3) The Driver maps the Compliant Response's C Function Names back to the **Application Function Names**

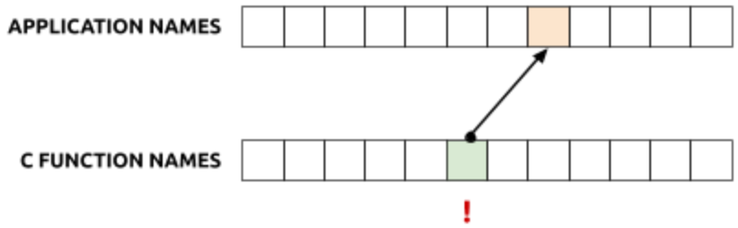


Fig. 2.3.1 - Driver mapping of C Function Name back to Application Name.

4) The Driver generates an **Application-formatted response** and passes it to the Application in format:

$$(Application\ Response) = (Application\ Function\ Name) + (Handle) + (Key.Public)$$

5) The Application ingests the response, receiving the Public Key and Handle products.

This concludes the description of the Endpoint Response Chain from the vHSM to the Application.

> ***Note:***
> The following section ('Source Code and Screenshots') shall follow the above sequencing.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3 - SOURCE CODE AND SCREENSHOTS

The interface function chosen for implementation was **RSA Keypair Generation** ('C_GenerateKeyPair'). To accomplish this, the Application must submit an appropriate response into the Request Chain and a relevant Public Key and Key Handle returned on the Response Chain. At the terminus, the response must be interpreted by the Application and the Response contents displayed to the User.

SECTION 3.0 - Application Request Generation

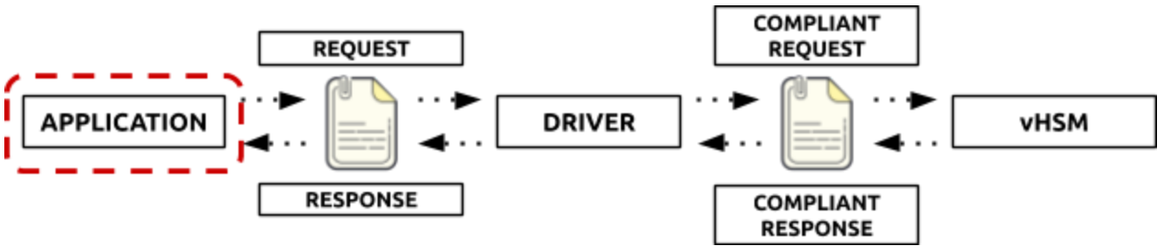


Fig. 3.0.0 - Process roadmap with current location in red.

The Application begins in a logged-in state, user credentials and authentication at this level being superfluous to the PKCS#11 protocol.

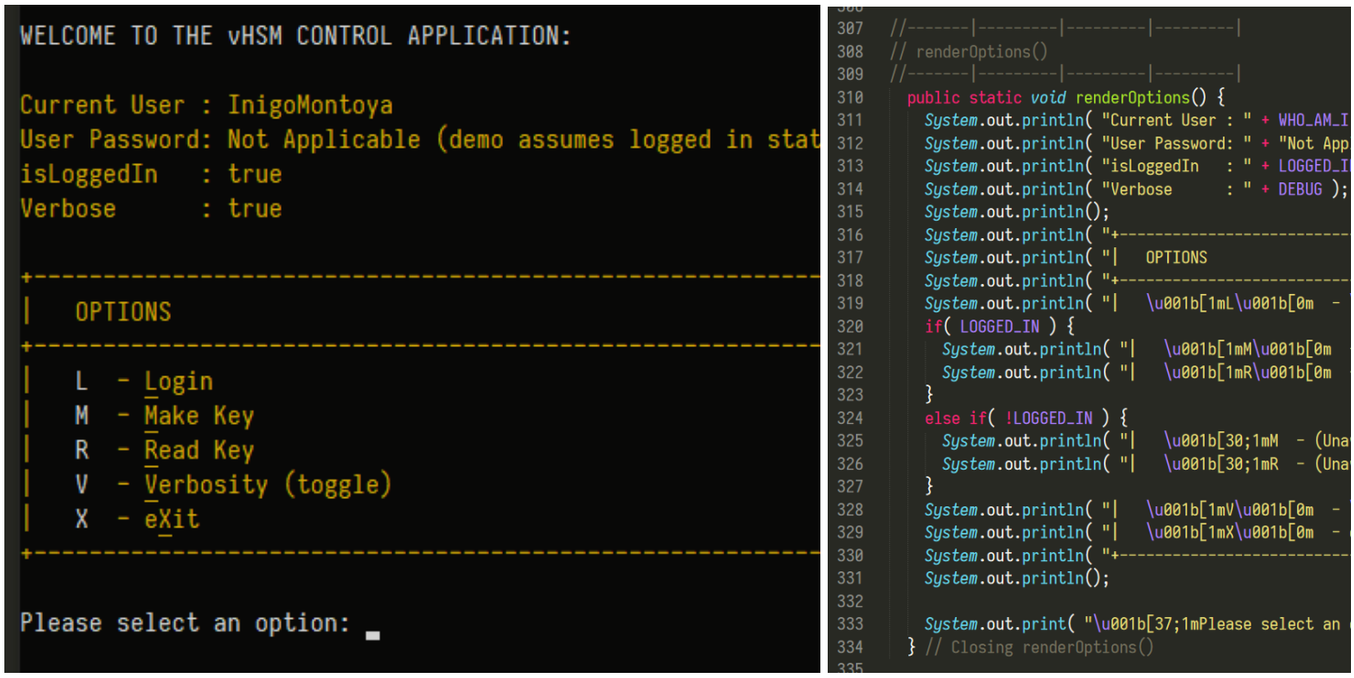


Fig. 3.0.1 - User interface of the Application and associated code block driving output.

The Application then may generate a keypair request using the user's name (**Username**) and their desired key password (**Keypass**).

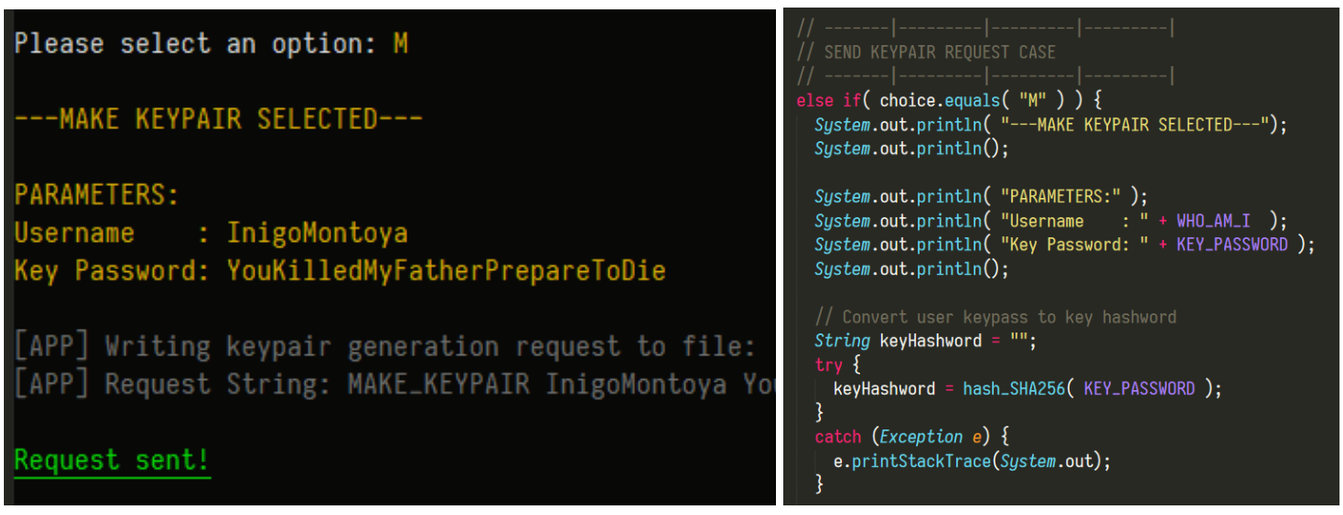


Fig. 3.0.2 - User selection of 'Make Keypair' from the Application interface and associated code block.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.1 - Application-Driver Request Send

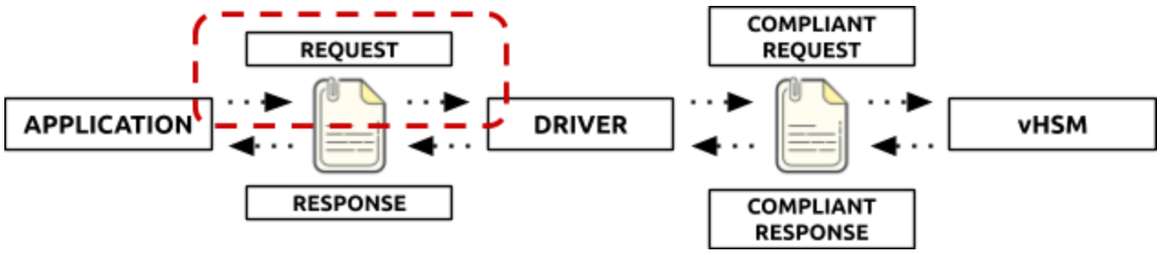


Fig. 3.1.0 - Process roadmap with current location in red.

The Application Request is written to file and awaits receipt by the Driver.

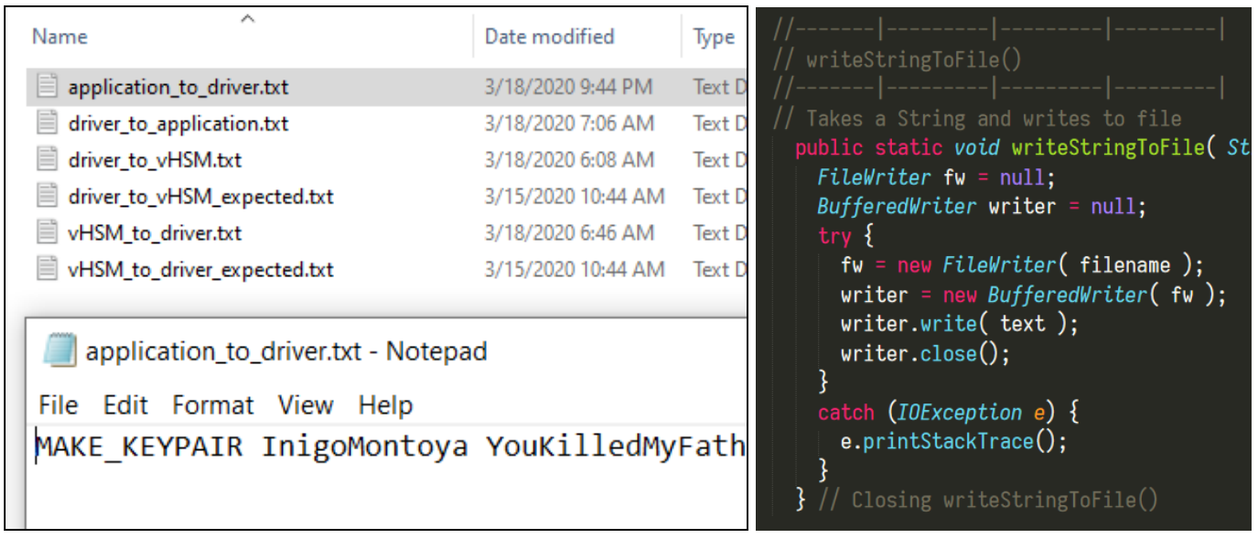


Fig. 3.1.1 - Application request written to file awaiting read and associated code block.

SECTION 3.2 - Driver Request Ingest

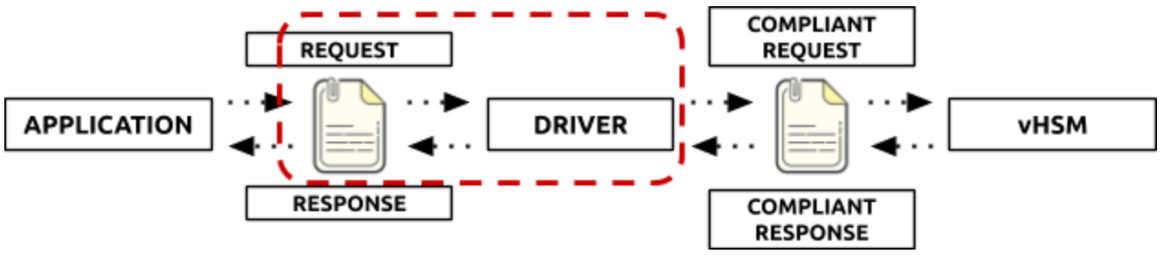


Fig. 3.2.0 - Process roadmap with current location in red.

The Driver's initial configuration action comprises accessing known Application to C Function Names (stored as a text file) and C Function Names to Function Numbers (also stored as a text file).

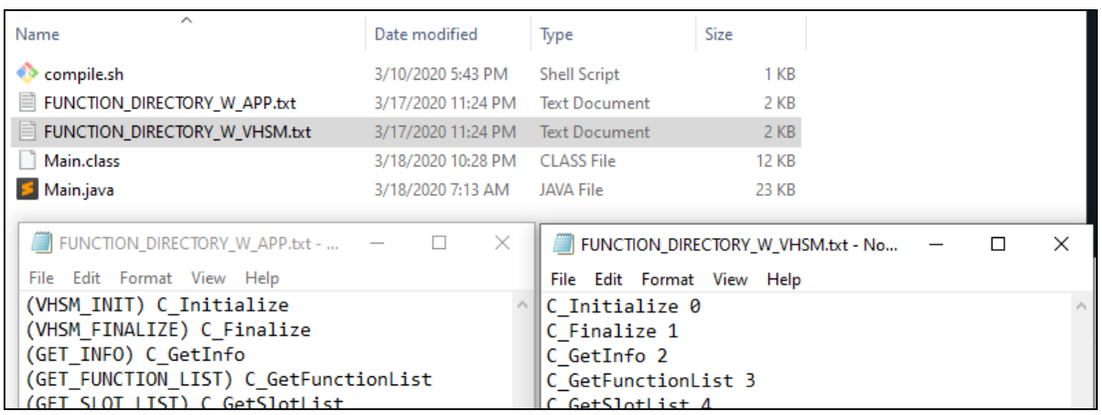


Fig. 3.2.1 - Function name mappings stored as text files.

Searchable databases (**Maps**) are generated from these values.

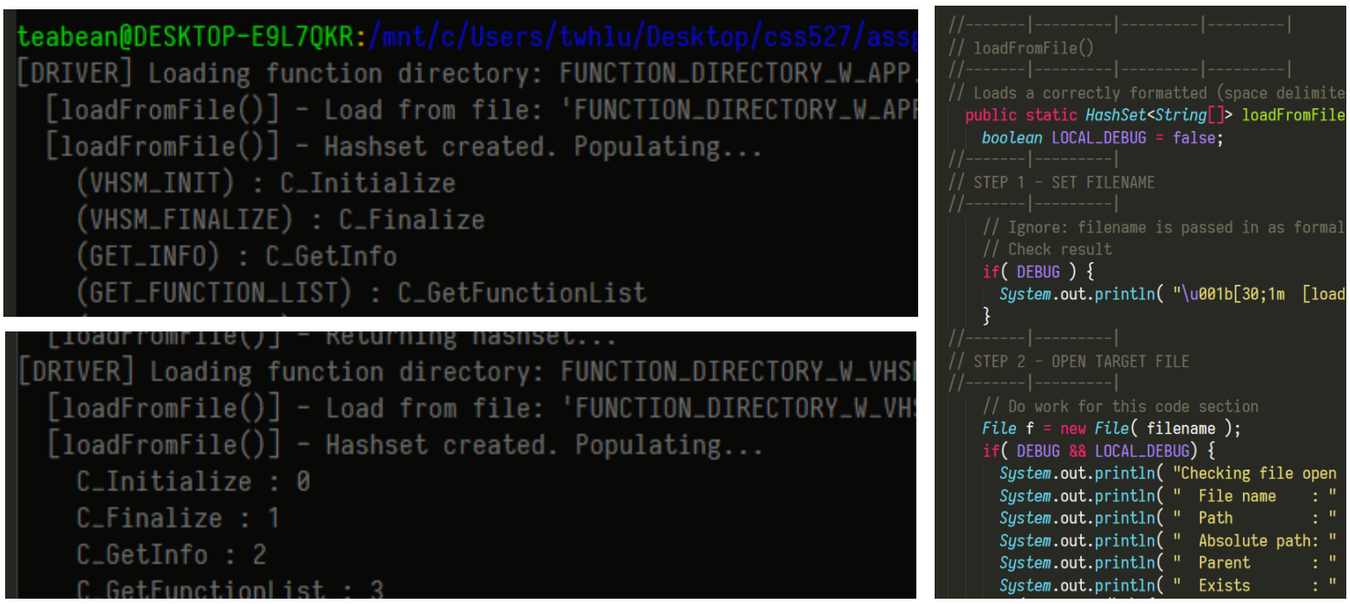


Fig. 3.2.1 - Function map generation and associated code block

The Driver does not maintain a concept of users and authentication, only acting as a message shunt between the Application and vHSM. As such, it begins in a ready state.

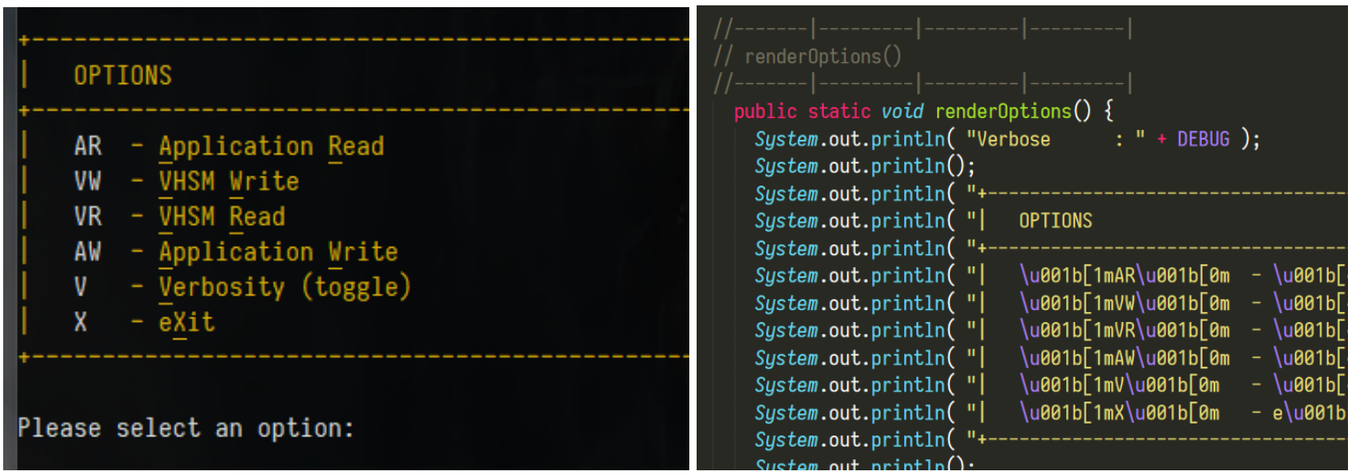


Fig. 3.2.2 - User interface of the Driver and associated code block driving output.

> **Note:**
> In a real implementation, operations by the driver and vHSM would be performed automatically and without need of user intervention. For PoC demonstration purposes, however, command inputs are required to illustrate each process step.

Selection of the 'Application Read' (AR) option directs the Driver to ingest a Request from an Application.

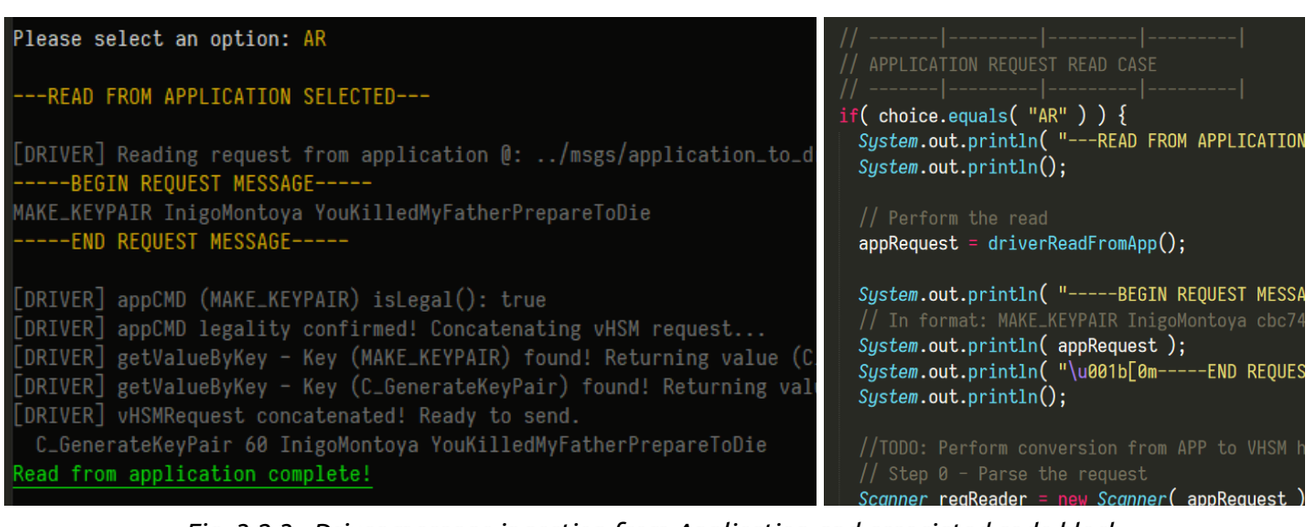


Fig. 3.2.3 - Driver message ingestion from Application and associated code block.

Upon ingestion, a PKCS#11 compliant message is created by mapping the application function call to the known C Function Call expected by the vHSM.

```
[DRIVER] vHSMRequest concatenated! Ready to send.
  C_GenerateKeyPair 60 InigoMontoya YouKilledMyFatherPrepareToDie
Read from application complete!
```

Fig. 3.2.4 - PKCS#11 message prepared and ready for send.

SECTION 3.3 - Driver-vHSM Request Send

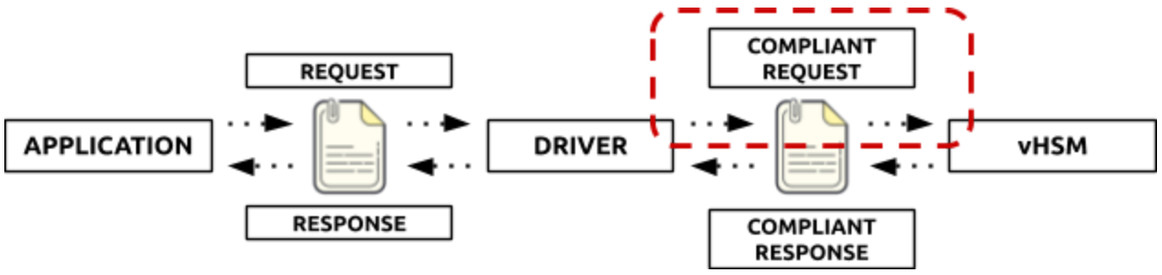


Fig. 3.3.0 - Process roadmap with current location in red.

The Driver may then validate the request as necessary and, at its discretion, forward the PKCS#11 compliant message to the vHSM. In this implementation, selection of the vHSM Write ('VW') option prompts this action.

```
Please select an option: VW

---WRITE TO VHSM SELECTED---

[DRIVER] Writing request message to vHSM...
  C_GenerateKeyPair 60 InigoMontoya YouKilledMyFatherP
[DRIVER] Writing request from application to: ../msgs/
Request sent to vHSM!
```

```
// -------|---------|---------|---------|
// vHSM REQUEST WRITE CASE
// -------|---------|---------|---------|
else if( choice.equals( "VW" ) ) {
  System.out.println( "---WRITE TO VHSM SELECTED---");
  System.out.println();

  if( DEBUG) {
    System.out.println( "\u001b[30;1m[DRIVER] Writing r
    System.out.println( "\u001b[30;1m  " + vHSMRequest
  }

  driverWriteTovHSM( vHSMRequest );
```

Fig. 3.3.1 - Driver-to-vHSM message send and associated code block.

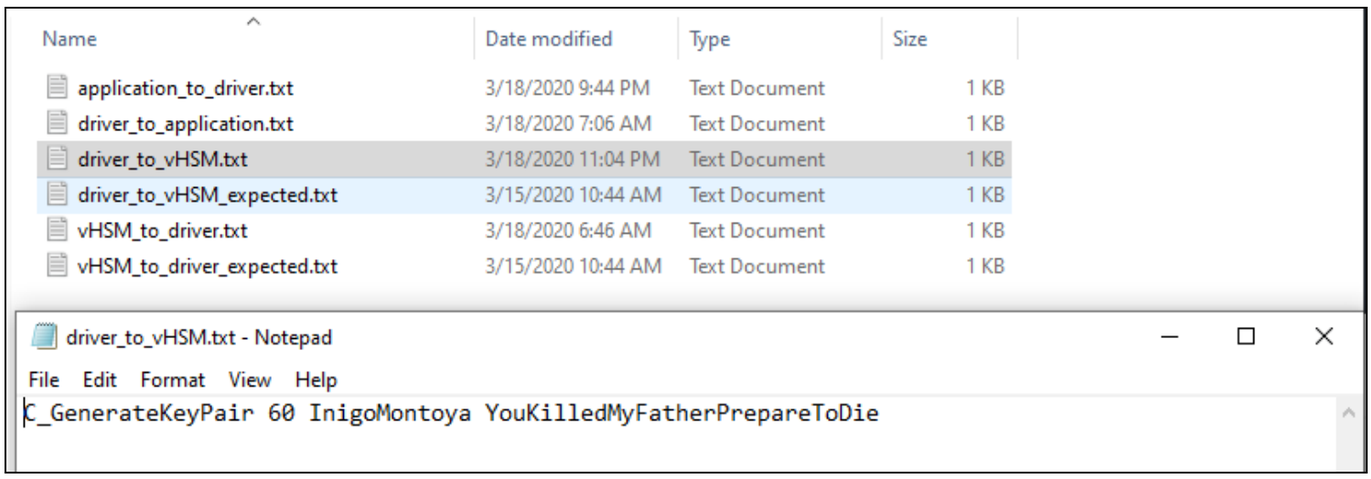


Fig. 3.3.1 - PKCS#11 compliant message awaiting ingest from vHSM.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.4 - vHSM Request Ingestion

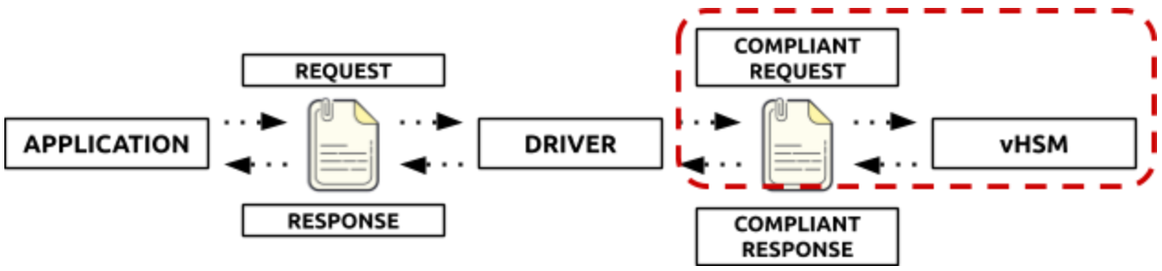


Fig. 3.4.0 - Process roadmap with current location in red.

While most aspects of the vHSM remain unaltered from the previous assignment, initialization in this instance required mapping C Function Names to Function Numbers. Upon startup, the vHSM consults a flat file of known associations and generates a HashSet of these Key:Value pairs.

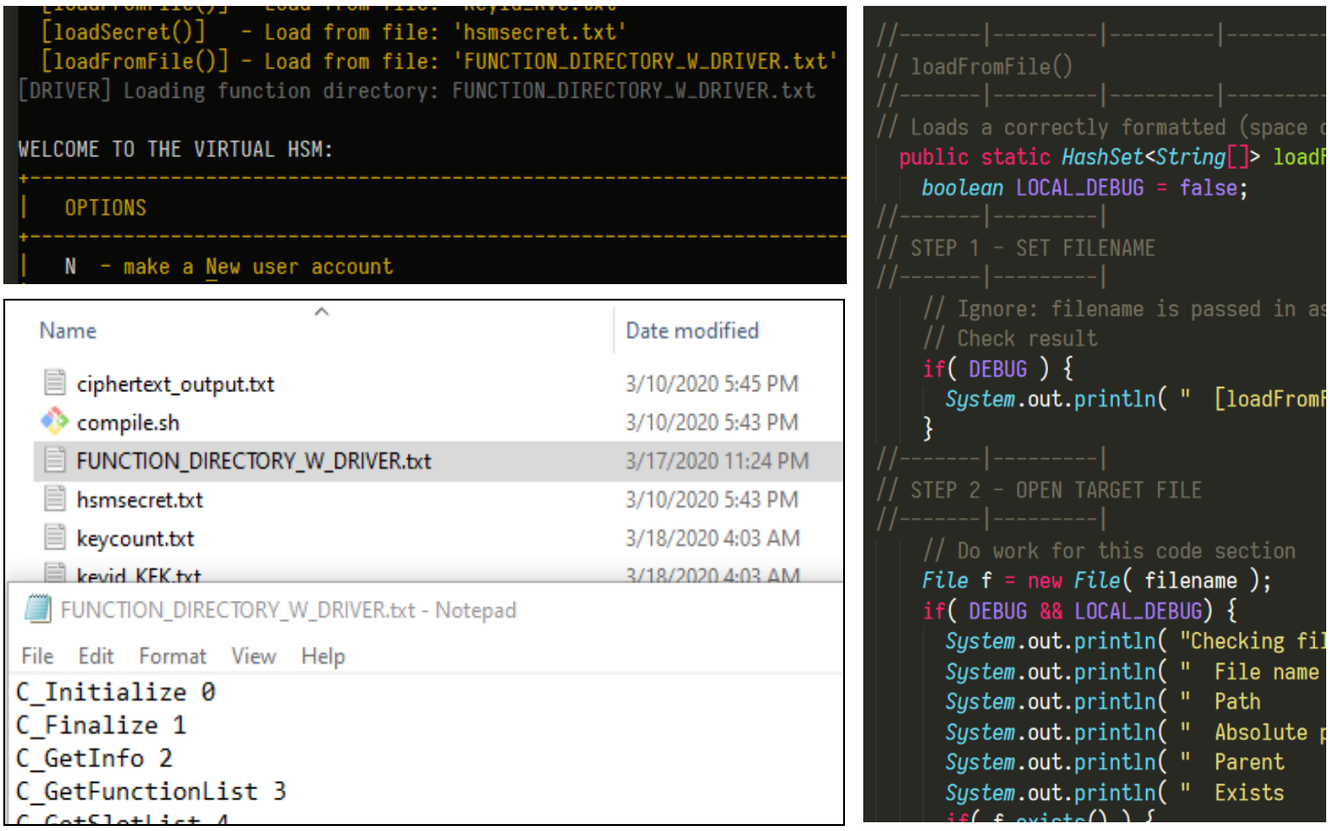


Fig. 3.4.1 - vHSM initial load of known C Function Name to Function Number maps and associated code control.

Once initialized, the vHSM may then elect to ingest awaiting messages.

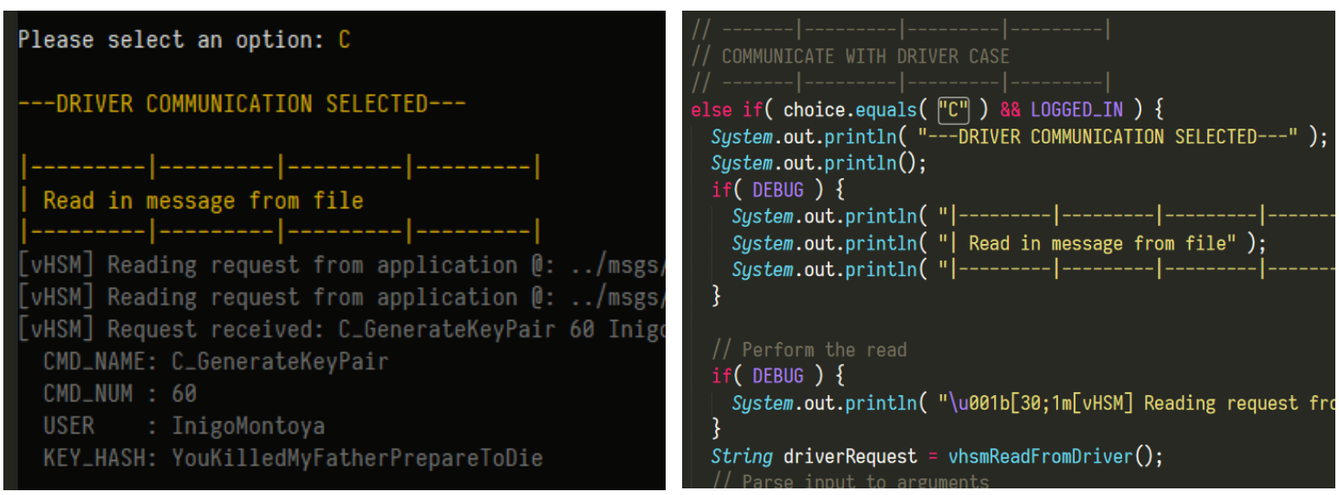


Fig. 3.4.2 - vHSM message ingestion selected and associated code block.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.5 - vHSM Endpoint Processing

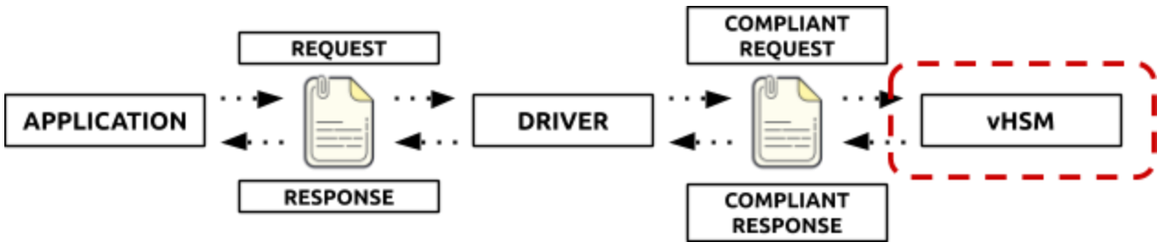


Fig. 3.5.0 - Process roadmap with current location in red.

During the message ingestion process, if the message is identified as both validly formatted and legal to perform (the complexity of this check may be extended arbitrarily), the vHSM will use the parsed fields to perform the requested cryptographic operation. In this case, 'C_GenerateKeyPair', which corresponds to the creation of an RSA public/private keypair.

```
// Note: No validation at this step, 'login' not implemented for assg4
boolean userIsLegal = isLegalUser( userDB, "InigoMontoya" );
boolean cmdIsLegal = isLegalCMD( funcDirectoryDriver, "C_GenerateKeyPair", "60" );

if( !(userIsLegal && cmdIsLegal) ) {
  System.out.println( "Illegal command request. Aborting." );
  // TODO: Write error message to return file
  String response = "UNABLE TO COMPLY";
  vhsmWriteToDriver( response );
  continue;
}
```

Fig. 3.5.1 - Request legality checks must determine the validity of the request as well as the permissions of the entities specified therein.

For full details on the creation, handling, and maintenance of the RSA Private Key, please see the previous assignment. For the purposes of this exercise, a public key and associated Handle is stored to the vHSM database.

```
|---------|---------|---------|----
| Secure encKey.Private in the DB
|---------|---------|---------|----
Storing the encKey.Private to the d
  Key ID        : InigoMontoya:0
  encKey.Private: E+ZtePUV5ZEHMMkxR
  [doesContainKeyValPair()] - No ke
  [addPair()] - Key (InigoMontoya:0
(KeyID : encKey.Private) stored to
[vHSM] CURR_HANDLE : InigoMontoya:0

RSA keypair generated!
```

```
OWNER       : KEY ID
InigoMontoya : InigoMontoya:0

KEY ID      : KEY VERIFICATION CODE (KVC)
InigoMontoya:0 : wvRQlBgxVs/MNxFJBLqZqw==

KEY ID      : PUBLIC KEY
InigoMontoya:0 : MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AM

KEY ID      : ENCRYPTED PRIVATE KEY
InigoMontoya:0 : eBeeQPsDa2ijygvTKvVCcqw8MhWFhLvmo
```

Fig. 3.5.2 - vHSM RSA keypair creation in response to ingested message (left) with resulting database state (right) reflecting storage of a keypair and associated key identifier.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.6 - vHSM-Driver Response Send

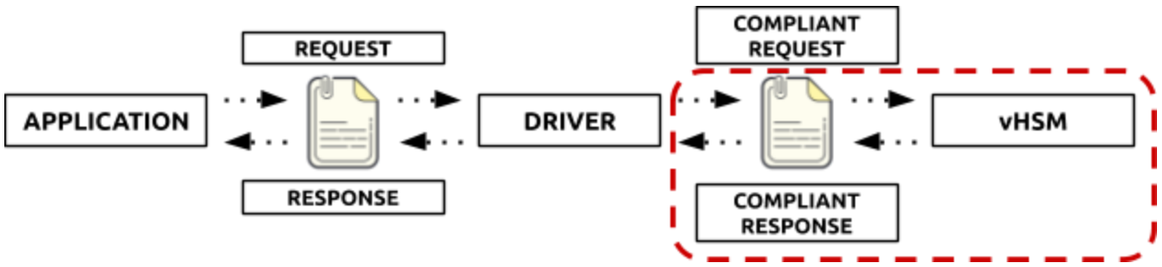


Fig. 3.6.0 - Process roadmap with current location in red.

After preparing a response to a cryptographic request (in this case, the generation of RSA keypairs), the vHSM may then push the product back up the response chain.

```
Please select an option: S

---SEND PRODUCT SELECTED---
```

```
C_GenerateKeyPair 60 InigoMontoya:0 MIIBIjANBgk
Gm3XlpLCBgcYAR3jrva2xt8ZSOOasc4ybwI25v1ay41TOP1
GpqY2FMLlgAQnK4suaHwq+ujgvc8g0naLNgsOjZ2H9dvsQH
0ATfi5ZnOEC8uoh2m4HAyAFM/nBETF3O2k0Dufxln7gjhS0
AQAB
[vHSM] Writing request from application to: ../
```

```
// -------|---------|---------|---------|
// SEND RESPONSE BACK CASE
// -------|---------|---------|---------|
else if( choice.equals( "S" ) && LOGGED_IN ) {
  System.out.println( "---SEND PRODUCT SELECTED---");
  System.out.println( );

  if( DEBUG ) {
    System.out.println( "|---------|---------|---------|
    System.out.println( "| Perform Key Lookup by Current
    System.out.println( "|---------|---------|---------|
  }
```

Fig. 3.6.1 - Invocation to Send a prepared Cryptographic product back and controlling code block

For PoC purposes, the response is written to file awaiting Driver re-ingest. The response in this instance contains the C Function Name being served, the Function Number, a unique product handle, and the Public Key.

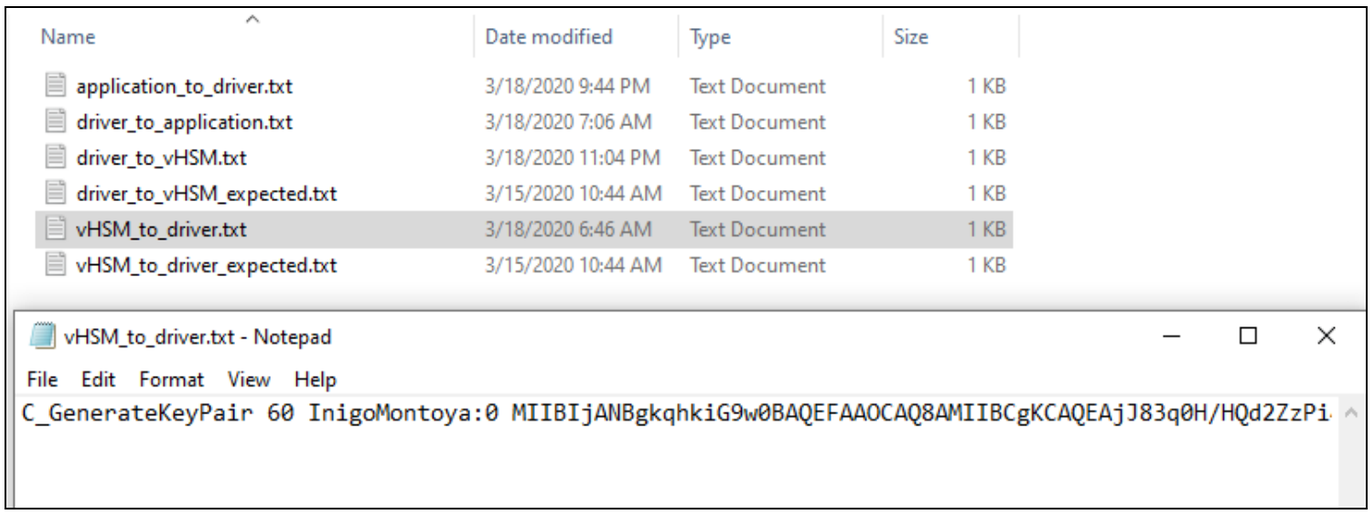


Fig. 3.6.2 - Response file written and awaiting read by the Driver.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.7 - Driver Response Ingest and Processing

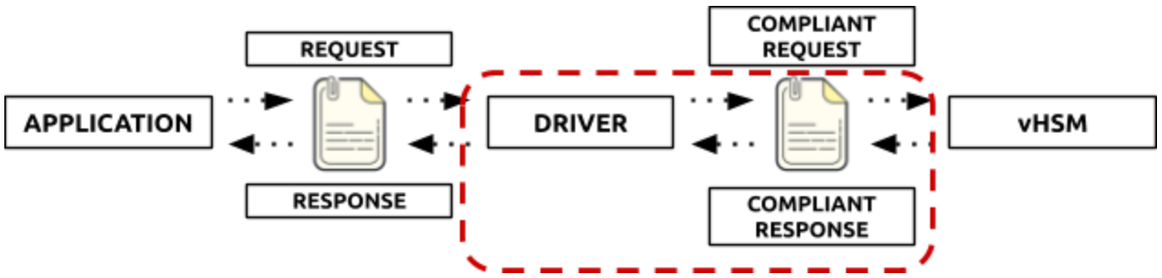
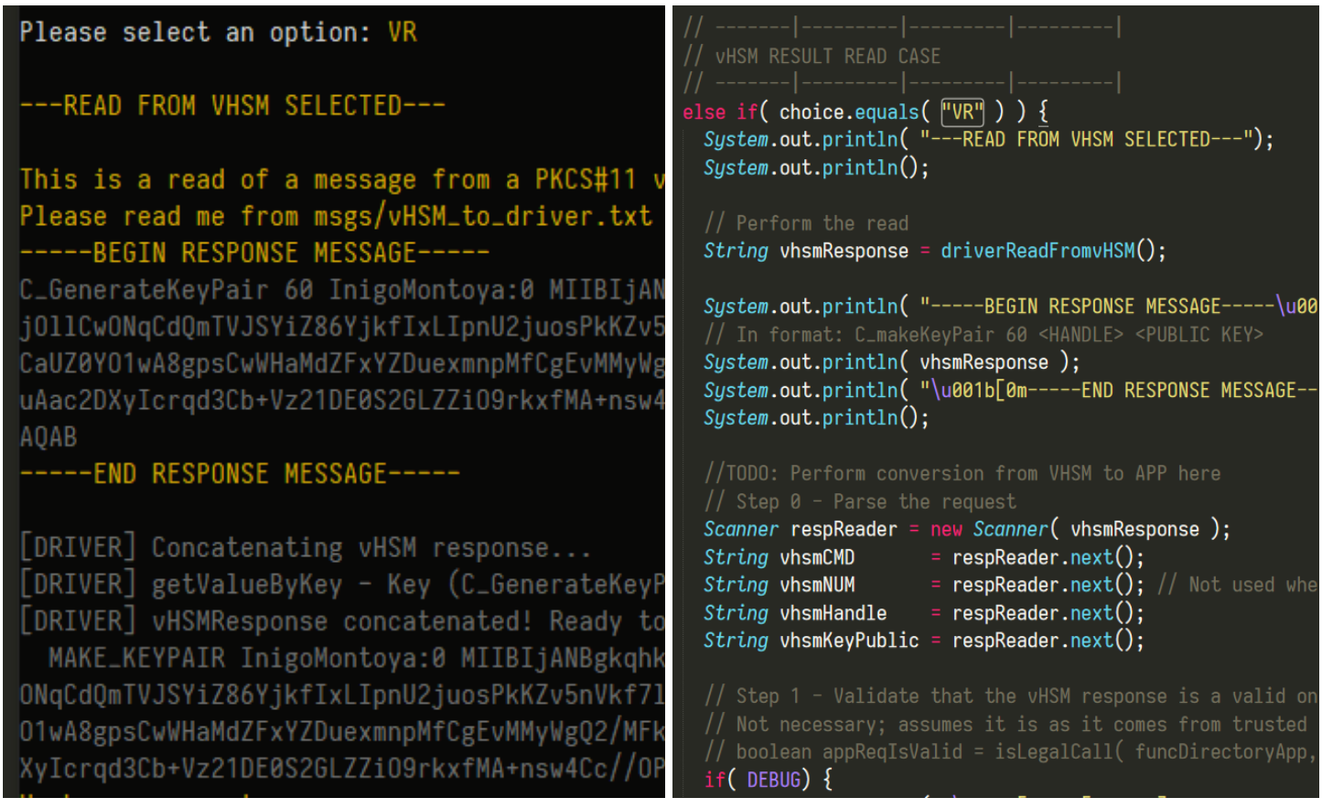


Fig. 3.7.0 - Process roadmap with current location in red.

For Driver may then elect to ingest the PKCS#11 Compliant Response, mapping its contents to an Application-Specific format in the inverse fashion to the Application Function Name to C Function Name mapping used in the request chain.



```
Please select an option: VR

---READ FROM VHSM SELECTED---

This is a read of a message from a PKCS#11 v
Please read me from msgs/vHSM_to_driver.txt
-----BEGIN RESPONSE MESSAGE-----
C_GenerateKeyPair 60 InigoMontoya:0 MIIBIjAN
jOllCwONqCdQmTVJSYiZ86YjkfIxLIpnU2juosPkKZv5
CaUZ0YO1wA8gpsCwWHaMdZFxYZDuexmnpMfCgEvMMyWg
uAac2DXyIcrqd3Cb+Vz21DE0S2GLZZiO9rkxfMA+nsw4
AQAB
-----END RESPONSE MESSAGE-----

[DRIVER] Concatenating vHSM response...
[DRIVER] getValueByKey - Key (C_GenerateKeyP
[DRIVER] vHSMResponse concatenated! Ready to
  MAKE_KEYPAIR InigoMontoya:0 MIIBIjANBgkqhk
ONqCdQmTVJSYiZ86YjkfIxLIpnU2juosPkKZv5nVkf7l
O1wA8gpsCwWHaMdZFxYZDuexmnpMfCgEvMMyWgQ2/MFk
XyIcrqd3Cb+Vz21DE0S2GLZZiO9rkxfMA+nsw4Cc//OP
```

```
// -------|---------|---------|---------|
// vHSM RESULT READ CASE
// -------|---------|---------|---------|
else if( choice.equals( "VR" ) ) {
  System.out.println( "---READ FROM VHSM SELECTED---");
  System.out.println();

  // Perform the read
  String vhsmResponse = driverReadFromvHSM();

  System.out.println( "-----BEGIN RESPONSE MESSAGE-----\u00
  // In format: C_makeKeyPair 60 <HANDLE> <PUBLIC KEY>
  System.out.println( vhsmResponse );
  System.out.println( "\u001b[0m-----END RESPONSE MESSAGE--
  System.out.println();

  //TODO: Perform conversion from VHSM to APP here
  // Step 0 - Parse the request
  Scanner respReader = new Scanner( vhsmResponse );
  String vhsmCMD       = respReader.next();
  String vhsmNUM       = respReader.next(); // Not used whe
  String vhsmHandle    = respReader.next();
  String vhsmKeyPublic = respReader.next();

  // Step 1 - Validate that the vHSM response is a valid on
  // Not necessary; assumes it is as it comes from trusted
  // boolean appReqIsValid = isLegalCall( funcDirectoryApp,
  if( DEBUG) {
```

Fig. 3.7.1 - vHSM ingestion, processing, and relevant controlling code block.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.8 - Driver-Application Response Send

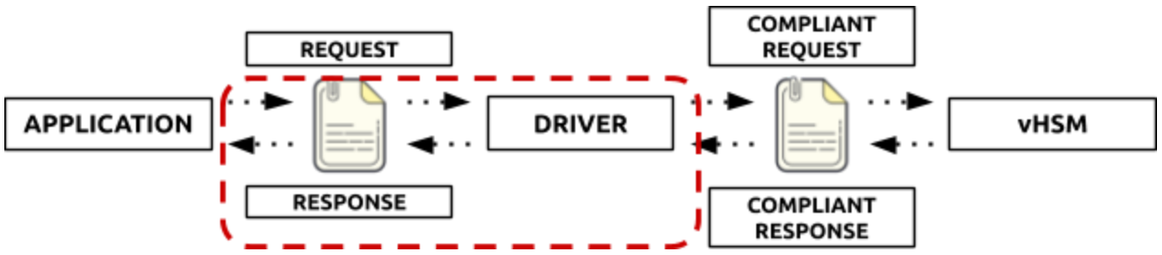


Fig. 3.8.0 - Process roadmap with current location in red.

```
Please select an option: AW

---WRITE TO APPLICATION SELECTED---

Write to Application complete!

[DRIVER] Writing keypair generation response to fi
[DRIVER] Response String: MAKE_KEYPAIR InigoMontoy
HQd2ZzPi4q7xWKKsfwjOllCwONqCdQmTVJSYiZ86YjkfIxLIpr
4PxoRghPnECv9KZgr4CaUZ0YO1wA8gpsCwWHaMdZFxYZDuexmr
dqx4gTIqu2Sqp3vrutuAac2DXyIcrqd3Cb+Vz21DE0S2GLZZiU
Uo9VJs6fD8PKBNyQIDAQAB

This is a write of a message to an application
Please write me to msgs/driver_to_application.txt
Response returned to application!
```

```
// -------|---------|---------|---------|
// APPLICATION RESULT WRITE CASE
// -------|---------|---------|---------|
else if( choice.equals( "AW" ) ) {
  System.out.println( "---WRITE TO APPLICATION SELECT
  System.out.println();

  System.out.println( "\u001b[32;1m\u001b[4mWrite to
  System.out.println();

  if( DEBUG ) {
    System.out.println( "\u001b[30;1m[DRIVER] Writing
    System.out.println( "\u001b[30;1m[DRIVER] Respons
    System.out.println();
  }

  driverWriteToApp( APP_RESPONSE );

  System.out.println( "\u001b[32;1m\u001b[4mResponse
  System.out.println();
}
```

Fig. 3.8.1 - Driver to Application message passing and relevant code control block.

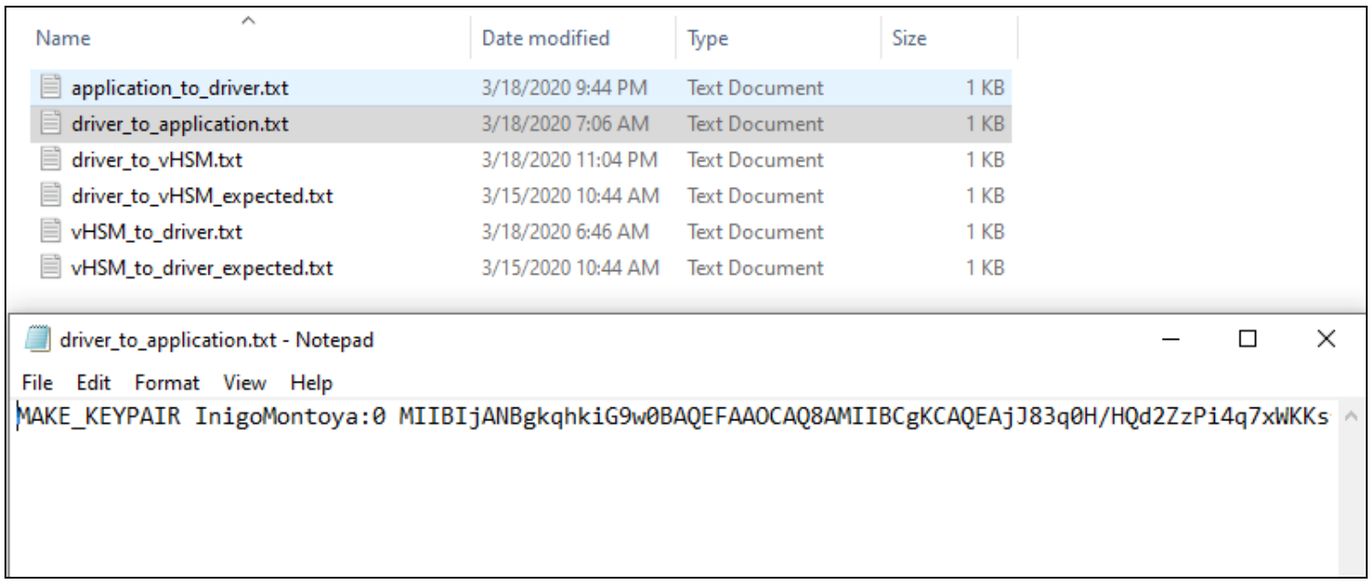


Fig. 3.8.2 - The vHSM product (a Key Handle and Public Key) are written to file and await Application ingest.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

SECTION 3.9 - Application Response Ingestion and Final Processing

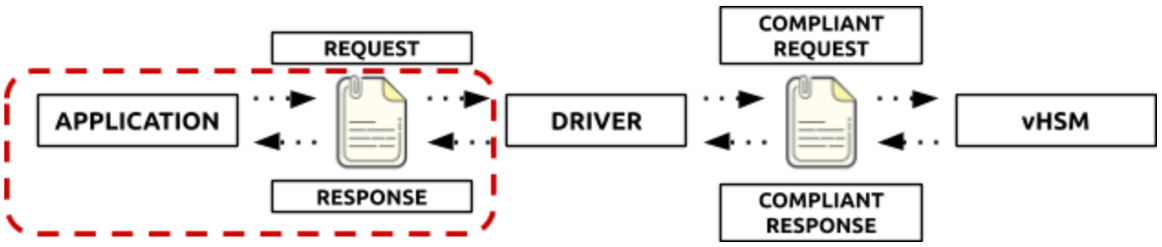


Fig. 3.9.0 - Process roadmap with current location in red.

Finally, the Application may ingest the response.

```
Please select an option: R

---RESULT READ SELECTED---

[APP] Reading keypair generation response from

-----BEGIN RESPONSE MESSAGE-----
MAKE_KEYPAIR InigoMontoya:0 MIIBIjANBgkqhkiG9w
qCdQmTVJSYiZ86YjkfIxLIpnU2juosPkKZv5nVkf7lg6ac
wA8gpsCwWHaMdZFxYZDuexmnpMfCgEvMMyWgQ2/MFkTRzr
Icrqd3Cb+Vz21DE0S2GLZZiO9rkxfMA+nsw4Cc//OPopFs
-----END RESPONSE MESSAGE-----

Read complete!
vHSM Handle    : InigoMontoya:0
vHSM Public Key:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAjJ
osPkKZv5nVkf7lg6acLl2Ju76DaIU1W+tYaa2bLAQFkOts
gEvMMyWgQ2/MFkTRzrOAv+t5yCVTF+pewjczvlD7rJpS2T
fMA+nsw4Cc//OPopFsPBLZycEyVkY5Q57D2GCndSkj9Q2f
```

```
// -------|---------|---------|---------|
// READ KEYPAIR RESPONSE CASE
// -------|---------|---------|---------|
else if( choice.equals( "R" ) ) {
  System.out.println( "---RESULT READ SELECTED---");
  System.out.println();

  if( DEBUG ) {
    System.out.println( "\u001b[30;1m[APP] Reading keypa
    System.out.println();
  }

  // Perform the read
  String readResult = applicationRead();

  System.out.println( "-----BEGIN RESPONSE MESSAGE-----\
  System.out.println( readResult );
  System.out.println( "\u001b[0m-----END RESPONSE MESSAG
  System.out.println();

  // Parse and isolate handle + Public key
  Scanner driverParser = new Scanner( readResult );
  String cmdToDiscard  = driverParser.next();
  String handle        = driverParser.next();
  String keyPublic     = driverParser.next();
```

Fig. 3.9.1 - Application response ingestion and controlling code block.

```
Read complete!
vHSM Handle    : InigoMontoya:0
vHSM Public Key:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAjJ83q0H/HQd2ZzPi4q7xWKKsfwjOllCwONqCdQmTVJSYiZ86YjkfIx
LIpnU2juosPkKZv5nVkf7lg6acLl2Ju76DaIU1W+tYaa2bLAQFkOtsicb1c64PxoRghPnECv9KZgr4CaUZ0YO1wA8gpsCwWHaM
dZFxYZDuexmnpMfCgEvMMyWgQ2/MFkTRzrOAv+t5yCVTF+pewjczvlD7rJpS2TUx0/Ftdqx4gTIqu2Sqp3vrutuAac2DXyIcrq
d3Cb+Vz21DE0S2GLZZiO9rkxfMA+nsw4Cc//OPopFsPBLZycEyVkY5Q57D2GCndSkj9Q2frhzFRgUo9VJs6fD8PKBNyQIDAQAB
```

Fig. 3.9.2 - Application final reporting state displaying the vHSM handle bound to the encrypted Private Key and the corresponding Public Key of the Keypair.

SECTION 4 - RESULTS

Per the assignment specification (Section 0.0), no results are produced for this exercise.

SECTION 5 - ANALYSIS

None for this assignment.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

APPENDIX

APPENDIX 0.0 - Application Source Code

Command Line Interface (CLI) version:
https://github.com/Teabeans/UW527_Crypto_Assg4_PKCS11

APPENDIX 0.1 - Specifications

PKCS Documentation:
http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/errata01/os/pkcs11-base-v2.40-errata01-os.html

APPENDIX 0.2 - Libraries Used in this Project

Java Cryptography: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

APPENDIX 0.3 - Select Terms and Definitions

TERM	DEFINITION
API	Application Programming Interface.
Application	Any computer program that calls the Cryptoki interface.
Certificate	A signed message binding a subject name and a public key, or a subject name and a set of attributes.
CMS	Cryptographic Message Syntax (see RFC 5652)
Cryptographic Device	A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.
Cryptoki	The Cryptographic Token Interface defined in this standard.
Cryptoki library	A library that implements the functions specified in this standard.
IV	Initialization Vector.
MAC	Message Authentication Code.
Mechanism	A process for implementing a cryptographic operation.
Object	An item that is stored on a token. May be data, a certificate, or a key.
PKCS	Public-Key Cryptography Standards.
PRF	Pseudo random function.
Reader	The means by which information is exchanged with a device.
R/O	Read-Only
R/W	Read / Write
Session	A logical connection between an application and a token.
Slot	A logical reader that potentially contains a token.
Token	The logical view of a cryptographic device defined by Cryptoki.
User	The person using an application that interfaces to Cryptoki.
UTF-8	Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.

Tim Lum + Matt Sell
2020.03.19
CSS527 - Cryptography

APPENDIX 0.4 - Variable Prefixes

PREFIX	DESCRIPTION		PREFIX	DESCRIPTION
C_	Function		CKP_	Pseudo-random function
CK_	Data type or general constant		CKS_	Session state
CKA_	Attribute		CKR_	Return value
CKC_	Certificate type		CKU_	User type
CKD_	Key derivation function		CKZ_	Salt/Encoding parameter source
CKF_	Bit flag		h	a handle
CKG_	Mask generation function		ul	a CK_ULONG
CKH_	Hardware feature type		p	a pointer
CKK_	Key type		pb	a pointer to a CK_BYTE
CKM_	Mechanism type		ph	a pointer to a handle
CKN_	Notification		pul	a pointer to a CK_ULONG
CKO_	Object class			

APPENDIX 0.5 - Function Name-Number Mappings

FUNCTION NAMES		FUNCTION NUMBER
APPLICATION	PKCS#11	
(VHSM_INIT)	C_Initialize	0
(VHSM_FINALIZE)	C_Finalize	1
(GET_INFO)	C_GetInfo	2
(GET_FUNCTION_LIST)	C_GetFunctionList	3
(GET_SLOT_LIST)	C_GetSlotList	4
(GET_SLOT_INFO)	C_GetSlotInfo	5
(GET_TOKEN_INFO)	C_GetTokenInfo	6
(WAIT_FOR_SLOT_EVENT)	C_WaitForSlotEvent	7
(GET_MECHANISM_LIST)	C_GetMechanismList	8
(GET_MECHANISM_INFO)	C_GetMechanismInfo	9
(INIT_TOKEN)	C_InitToken	10
(INIT_PIN)	C_InitPIN	11
(SET_PIN)	C_SetPIN	12
(OPEN_SESSION)	C_OpenSession	13
(CLOSE_SESSION)	C_CloseSession	14
(CLOSE_ALL_SESSIONS)	C_CloseAllSessions	15
(GET_SESSION_INFO)	C_GetSessionInfo	16
(GET_OPERATION_SIZE)	C_GetOperationState	17
(SET_OPERATION_STATE)	C_SetOperationState	18
(LOGIN)	C_Login	19
(LOGOUT)	C_Logout	20
(OBJ_CREATE)	C_CreateObject	21
(OBJ_COPY)	C_CopyObject	22
(OBJ_DESTROY)	C_DestroyObject	23
(GET_OBJ_SIZEOF)	C_GetObjectSize	24
(GET_ATTRVAL)	C_GetAttributeValue	25
(SET_ATTRVAL)	C_SetAttributeValue	26
(INIT_FINDOBJ)	C_FindObjectsInit	27
(FINDOBJ)	C_FindObjects	28
(FINDOBJ_FINAL)	C_FindObjectsFinal	29
(INIT_ENC)	C_EncryptInit	30
(ENC)	C_Encrypt	31
(ENC_UPD)	C_EncryptUpdate	32

FUNCTION NAMES		FUNCTION NUMBER
APPLICATION	PKCS#11	
(ENC_FINAL)	C_EncryptFinal	33
(INIT_DEC)	C_DecryptInit	34
(DEC)	C_Decrypt	35
(DEC_UPD)	C_DecryptUpdate	36
(DEC_FINAL)	C_DecryptFinal	37
(INIT_DIGEST)	C_DigestInit	38
(DIGEST)	C_Digest	39
(DIGEST_UPDATE)	C_DigestUpdate	40
(DIGEST_KEY)	C_DigestKey	41
(DIGEST_FINAL)	C_DigestFinal	42
(INIT_SIGN)	C_SignInit	43
(SIGN)	C_Sign	44
(SIGN_UPD)	C_SignUpdate	45
(SIGN_FINAL)	C_SignFinal	46
(INIT_SIGN_RECOVER)	C_SignRecoverInit	47
(SIGN_RECOVER)	C_SignRecover	48
(VERIFY_INIT)	C_VerifyInit	49
(VERIFY)	C_Verify	50
(VERIFY_UPDATE)	C_VerifyUpdate	51
(VERIFY_FINAL)	C_VerifyFinal	52
(INIT_VERIFY_RECOVER)	C_VerifyRecoverInit	53
(VERIFY_RECOVER)	C_VerifyRecover	54
(DIG_ENG_UPD)	C_DigestEncryptUpdate	55
(DEC_DIG_UPD)	C_DecryptDigestUpdate	56
(SIGN_ENC_UPD)	C_SignEncryptUpdate	57
(DEC_VER_UPD)	C_DecryptVerifyUpdate	58
(MAKE_KEY)	C_GenerateKey	59
MAKE_KEYPAIR	**C_GenerateKeyPair**	**60**
(WRAP_KEY)	C_WrapKey	61
(UNWRAP_KEY)	C_UnwrapKey	62
(DERIVE_KEY)	C_DeriveKey	63
(SEED_RANDOM)	C_SeedRandom	64
(GEN_RANDOM)	C_GenerateRandom	65