VORMETRIC APPLICATION ENCRYPTION ARCHITECTURE

Vormetric, Inc.
2860 Junction Ave, San Jose, CA 95134
United States: 888.267.3732
United Kingdom: +44.118.949.7711
Singapore: +65 6829 2266
info@vormetric.com
www.vormetric.com

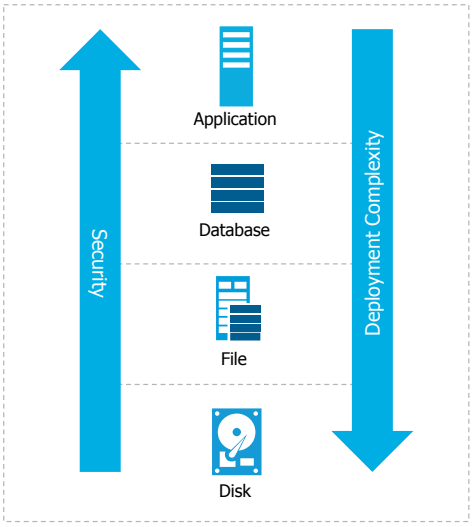
TABLE OF CONTENTS:

INTRODUCTION

Organizations and individuals are more connected digitally than ever before. As a consequence, data is accessible to more people than ever before. While digitization accelerates information sharing, it exacerbates the threat of sensitive data falling into the wrong hands. To combat this threat, enterprises turn to encryption.

Encryption is the process of encoding sensitive data so that only authorized parties can read it. Encryption is typically employed in one of these four levels of the technology stack: full-disk (or media-based), file, database, or application. Vormetric provides solutions for encrypting at the file, database, and application layers. The company also offers key management for many full-disk encryption solutions.

When deploying encryption solutions, the lower in the stack the encryption/decryption occurs, the less likely these processes are to interfere with the operations of other layers. For example, if the encryption occurs at the disk level, there is very little risk of the encryption affecting the other layers. The file, database, and application layers will be accessing data in the clear and will function the same as before. On the other hand, if the encryption occurs in the application layer, the disk, file, and database layers will be accessing encrypted data, which may have an impact on operations in these layers. However, because data is encrypted across multiple layers, this approach can reduce the number of potential attack vectors and so increase security.



Security increases when encryption is implemented higher in the stack, but it is more complex to deploy

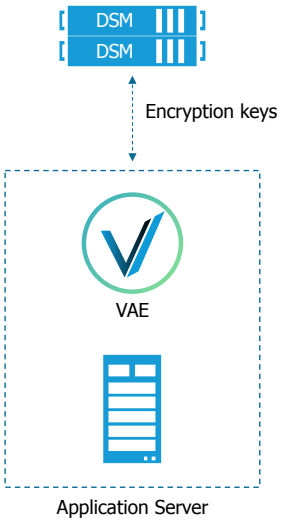
In spite of these tradeoffs, an organization that wants to establish the highest level of security would be well served by implementing application encryption. Application encryption provides the highest level of security with the most granular control of data. However, encryption and key management can be a challenge for many application development organizations. This is why Vormetric developed Vormetric Application Encryption. With this product, developers can easily implement application encryption, data access controls, and key management—without having to become cryptography experts or compromise the security of cryptographic keys.

This paper provides a technical overview of Vormetric Application Encryption. The following sections describe the product's architecture and security model and reveal how it helps organizations comply with security policies and industry standards. The paper also provides an example of how an application can use Vormetric Application Encryption to encrypt data.

ARCHITECTURE

The Vormetric Application Encryption solution contains two components:

- **Vormetric Data Security Manager (DSM).** The DSM is the central component of the Vormetric Data Security Platform. The DSM offers capabilities for managing policies and keys in a centralized fashion. The DSM stores and manages host encryption keys, data access policies, administrative domains, and administrator profiles.
- **Vormetric Application Encryption Library.** Vormetric Application Encryption features a library that implements a subset of PKCS#11 APIs. This is run as a dynamically loaded library (.dll) on Windows or as a shared object (.so) on Linux and Unix. The Vormetric Application Encryption library communicates over a secure channel to the DSM.



The DSM centrally manages policies and keys for the Vormetric Application Encryption Libraries running on servers

PROGRAMMING LANGUAGES SUPPORTED BY VORMETRIC APPLICATION ENCRYPTION

The Vormetric Application Encryption library supports the following languages:

- **C.** The library is written in C, so C-based functions are exported as part of the library and can be called directly by C-based applications.

- **Java.** The C library can be called from a Java library with the help of a standard JDK package named "sun.security.pkcs11.wrapper". This is a lightweight wrapper provided by the Oracle JDK to allow Java to interface with PKCS#11 C APIs.

- **.NET.** Vormetric Application Encryption ships with an additional DLL for .NET applications. Named "PKCS11InterOp.dll", this DLL converts data types and functions from managed objects in C# to unmanaged C objects in the PKCS#11 library.

Sample code for all three languages is provided with the product.

STANDARDS AND APIS

Vormetric is an active voting member of the Oasis PKCS#11 (Public Key Cryptography Standards) open standards committee. Vormetric Application Encryption implements PKCS#11 APIs. The PKCS#11 standard defines a platform-independent API to integrate with cryptographic tokens, smart cards, and hardware security modules (HSMs). The APIs define the most commonly used cryptographic types and operations. The Vormetric application library implements a subset of PKCS#11 APIs to enable the following functions:

- Create a key
- Search for a key by name
- Destroy a key
- Export a key (wrapped with another key) from DSM
- Import a key into DSM
- Encrypt
- Decrypt
- Sign
- Verify

SECURITY MODEL

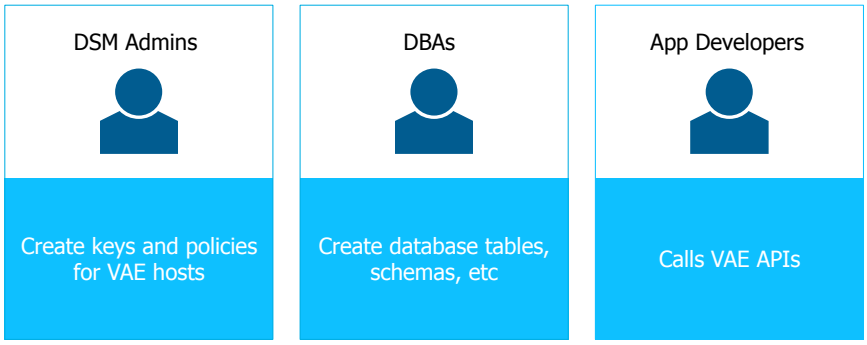
Vormetric Application Encryption is designed from the ground up to be secure. The security model inherits the authentication model from the PKCS#11 standard, including the use of personal identification numbers (PIN). The security model also offers capabilities for separation of duties and strong authentication, which enables enterprises to restrict privileged users from accessing data. By providing centralized key management with the DSM, the solution can help administrators easily and securely create, manage, and back up all cryptographic keys.

SEPARATION OF DUTIES

Separation of duties is a proven security method for mitigating the risk of a data breach. Through this approach, organizations can restrict the amount of power held by any one administrator and help prevent any single individual from committing data theft, sabotaging keys, or performing other malicious activities. With the Vormetric solution, controls can be divided among three groups:

- DSM administrators
- Database administrators (DBAs)
- Application developers

The following diagram illustrates the privileges and responsibilities of each group of users. When implemented properly, no single user can gain complete access to sensitive data without the cooperation of other groups.



DSM Administrators

DSM administrators are responsible for the day-to-day administration of the DSM platform. They can configure the list of hosts that are allowed to register to a particular DSM or DSM cluster and they create security administrator accounts. Security administrators are the only administrators who have the authority to create and manage encryption keys and policies. DSM and security administrators do not have administrative access to Windows or Linux hosts, nor do they have credentials to administer the database. Consequently, even though DSM and security administrators have access to the encryption keys, they will not be able to gain access to the encrypted data.

Database Administrators (DBAs)

DBAs are responsible for installing, configuring, maintaining, and monitoring databases in an organization. When organizations employ application encryption and separation of duties, they can enable DBAs to control the operational aspects of the database, but without having the privileges needed to access the sensitive data stored within the database.

Application Developers

Application developers work at the top of the encryption stack. They develop applications that use Vormetric Application Encryption APIs to encrypt and decrypt data in databases.

To limit their power and privileges, security guidelines often mandate that application developers write and test code using only simulated data. For example, requirement 6.4 of the Payment Card Industry Data Security Standard (PCI DSS) mandates that "The development/test environments are separate from the production environment, with access control in place to enforce the separation." By enforcing this separation of development and production environments, organizations can ensure that application developers won't be able to decrypt and steal sensitive data.

To learn more about the comprehensive PCI DSS coverage that Vormetric provides, please visit: www.vormetric.com/compliance/pci-dss.

VORMETRIC APPLICATION ENCRYPTION AND THE SECURITY PIN

The Vormetric Application Encryption PIN provides an additional layer of security. The Vormetric Application Encryption library supports host-based authentication. The first step to running the Vormetric Application Encryption library is to register it with the DSM. During registration, certificates are exchanged and a Vormetric Application Encryption password (PIN) is created. The certificates are tied to the host to ensure than even if a rogue application developer steals the password from Vormetric Application Encryption, he or she cannot write applications on their own to access encryption keys from the DSM.

In order to execute the encryption and key management APIs of Vormetric Application Encryption, the user of the application must provide the PIN. The PIN is controlled by the machine administrator, not the application developer. The PIN is used to encrypt the private key certificate file used in creating a secure communication channel with the DSM. Every time the Vormetric Application Encryption application starts up, the PIN must be provided to unlock the private key file needed to establish two-way TLS communications with the DSM. If the PIN is lost, the root administrator must re-register the Vormetric Application Encryption host with the DSM and create a new PIN.

Vormetric Application Encryption also has an optional, but highly recommended, feature of enabling hardware association. This ties the certificate to the various hardware attributes of the host it is installed on, including the MAC address. With this hardware association, even if a rogue developer steals certificates and the PIN from the installed host, he or she will not be able to simply copy the certificates and connect to the DSM with a rogue machine.

ENCRYPTION

The Vormetric Application Encryption library offers capabilities for AES Encryption, supporting 128 and 256 bit keys in CBC mode, and format-preserving encryption (FPE).

AES is an encryption standard established by the National Institute of Standards and Technology (NIST) in 2001. Since then, AES has become the industry standard for encryption. The 256-bit key size is considered cryptographically most secure.

FPE was approved by NIST in early 2016 and it leverages AES algorithms and functionality. Unlike with most other encryption algorithms, with FPE the resulting ciphertext isn't any longer than the original value of the plaintext that was encrypted. This feature can be useful when the contents of fixed-size database columns, like those containing credit card numbers or Social Security numbers, need to be encrypted, without modifying the schema of the database.

CIPHER BLOCK CHAINING (CBC) MODE

Vormetric solutions support CBC mode because it is cryptographically more secure than the older Electronic Code Book (ECB) mode. With the ECB mode, each form of plaintext is encrypted into identical cipher text, which can potentially leak dangerous information to attackers.

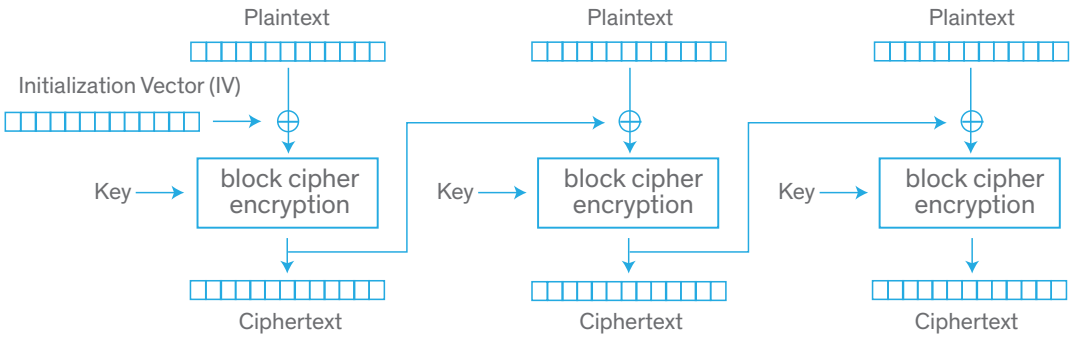
For example, consider uncompressed bitmap images. If an uncompressed bitmap image is encrypted using ECB mode, each 16-byte block of data is replaced by an unpredictable 16-byte block of cipher text, but identical input blocks result in identical cipher text blocks. Thus, if the image shows an object in front of an all-white background, the shape of the object would still be discernable in the encrypted bitmap image.

CBC enhances security because the output of this mode of encryption depends on the outcome of the previous block. The very first block to be encrypted depends on an initialization vector that is passed in as a parameter to the encryption operation. This initialization vector is a "pseudo-block" that is used to provide input to the first block of data to be encrypted. Supplying a different initialization vector to each piece of data to be encrypted ensures that the cipher text will always be unique and will not leak information to an attacker.

For example, consider two inputs of 32 bytes each: "00000000000000000123456789abcdef" and "0123456789abcdef0123456789abcdef". The first block (16 bytes) of each of these strings is different. The first block of the first input is "00000000000000000" and the first block of the second input is "123456789abcdef". The second block of these inputs are identical.

If we were to use EBC mode, the second block of these strings would encrypt to the exact same value. "0123456789abcdef" would encrypt to the same value no matter where that pattern occurs in the file. This leaks potentially sensitive information to the attacker that they can use to further decrypt the data.

CBC mode, by contrast, takes into account the output of the previous block as input. So even though the plaintext blocks are identical, the resulting ciphertext will be different. This ensures that no useful information is leaked to an attacker.



Cipher Block Chaining (CBC) mode encryption

CENTRALIZED KEY MANAGEMENT

Enterprises must not only protect against data theft, but they must also protect their encryption keys from theft, misplacement, and accidental destruction. To facilitate these safeguards, Vormetric Application Encryption supports enterprise key management through the DSM. The DSM is a highly secure, tamper-resistant hardware appliance that is FIPS 140-2 level 2 or level 3 certified. Like other Vormetric products, all keys that are created and used by the Vormetric Application Encryption library reside in the DSM. This gives Vormetric customers a unified, centralized platform for encryption and key management.

As mentioned earlier, there are some implementation factors to consider when deploying Vormetric Application Encryption. If the data resides in a relational database and the AES-CBC encryption algorithm is employed, encryption will necessitate schema changes. This is due to three factors: accommodating changes in data size associated with encryption, keeping track of the initialization vector, and enabling future support of key rotation. However, at least the first factor can be avoided by employing FPE.

Following is more information on each of these factors.

ACCOMMODATING DATA SIZE CHANGES WHEN AES-CBC IS USED

The first consideration is the change in data type and data size, which occurs when AES-128 or AES-256 encryption in CBC mode is used. With this particular algorithm, the data type and the data size changes. The data type changes from the original format (integer, varchar) to binary and the data size expands, comprising the original data size plus the block size (16 bytes). Database schemas must be changed to accommodate these differences. If changing the database schema isn't an option, then consider using FPE. However, since FPE uses multiple AES operations internally, the encryption performance of FPE may be lower than when AES-CBC is employed.

KEEPING TRACK OF THE AES-CBC INITIALIZATION VECTOR OR THE FPE TWEAK

To establish the strongest security, a unique initialization vector (for AES-CBC) or so-called tweak (for FPE) should be supplied for each row of data in the database. A unique initialization vector or tweak for each row will ensure that when identical data appears in multiple rows in the same column, encrypted values will be different each time, ensuring no information is leaked to an attacker. Each initialization vector or tweak needs to be stored in the row alongside the data because the same initialization vector or tweak needs to be supplied to support the decryption function.

KEY ROTATION

This is not strictly necessary, but the recommended third change in the database schema is to plan for the support of key rotation. Many security mandates, including PCI DSS, require an encryption key to be rotated every 12 to 24 months. To support key rotation, the user should add another column to the database schema to keep track of the current key name. This will allow the enterprise to implement key rotation as the data is accessed, rather than having to take production systems down to do the key rotation on the entire database at one time.

Schema before encryption

First Name (varchar)	Last Name (varchar)	National ID Number (INT)
John	Doe	245125
James	Smith	125626
Susan	Jones	324613
Bob	Chan	466262

Schema after encryption

First Name (binary)	Last Name (binary)	National ID Number (binary)	IV	Key
1y2t212	14qfagas	1w3g152	123415251524142	Key1
46twwh	3rabw53	12gawe4	463414352361434	Key1
4636rgw	124aha3	2gat3412	346675232361434	Key1
13g2ae3	asefb235	q3ag3521	234426213225353	Key1

PERFORMANCE CONSIDERATIONS

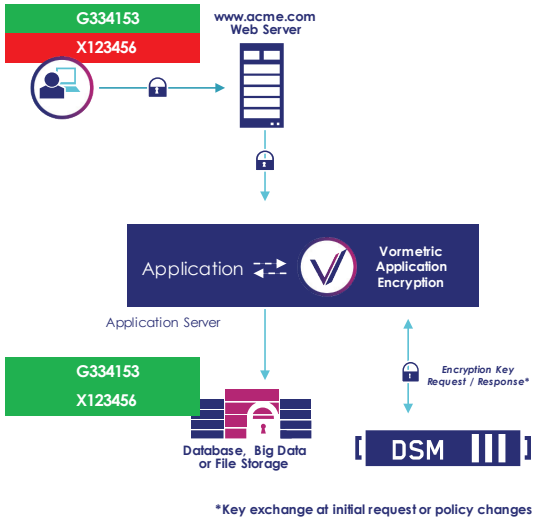
To maintain the highest level of security, Vormetric Application Encryption can be configured to ensure encryption keys never leave the DSM. In this scenario, the data that must be encrypted or decrypted is sent to the DSM. However, this mode should be used only when small data objects, such as certificates or passwords, must be encrypted or decrypted. For organizations that have to support the encryption of large data sets and data in performance-sensitive, highly transactional operations, Vormetric introduced a local key cache in Vormetric Application Encryption, version 5.2.1. Keys are securely exported from the DSM and then stored locally within the Vormetric Application Encryption library. Subsequent cryptographic operations are performed locally without going to the DSM, which offers significant performance advantages. Using this approach, organizations can perform 50,000 transactions per second, per thread.

Note: At the time of this paper's publication, when FPE is employed, keys must be cached locally.

EXAMPLE APPLICATION ENCRYPTION WORKFLOW

Here's a common workflow in an e-commerce scenario:

1. The user submits personal information to purchase items.
2. The Web server sends personal information to the application server.
3. The application calls into Vormetric Application Encryption to encrypt sensitive data.
4. The Vormetric Application Encryption library returns encrypted values back to the application.
5. The application then stores the encrypted value.



Vormetric Application Encryption with format preserving encryption applied to protect PII

OPERATIONAL ASPECTS

UPGRADES AND PATCHING

Newer versions of the DSM support older versions of the Vormetric Application Encryption agent, but not vice versa. Consequently, the recommended workflow is for customers to upgrade their DSMs first, and then upgrade their existing Vormetric Application Encryption agents.

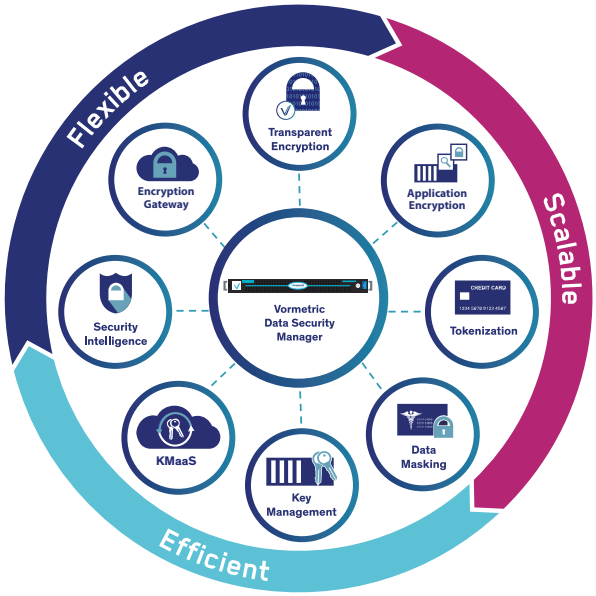
Vormetric Application Encryption libraries can be easily upgraded to newer versions without having to re-register with the DSM, which helps ensure business continuity.

COMPATIBILITY WITH DSM

The respective release notes for Vormetric Application Encryption and for the DSM describe the current compatibility matrix.

CONCLUSION

Vormetric Application Encryption is one of the products that is featured in the Vormetric Data Security Platform. The Vormetric Data Security Platform makes it efficient to manage data-at-rest security across your entire organization. Built on an extensible infrastructure, the Vormetric Data Security Platform features several products that can be deployed individually, while offering efficient, centralized key management. These products deliver capabilities for transparent file-level encryption, application-layer encryption, tokenization, cloud encryption gateway, integrated key management, and security intelligence logs. Through the platform's centralized key management and flexible implementation, you can address security policies and compliance mandates across databases, files, and big data environments—whether assets are located in the cloud, virtualized systems, or traditional infrastructures. With this platform's comprehensive, unified capabilities, you can efficiently scale to address your expanding security and compliance requirements, while significantly reducing total cost of ownership (TCO).



Vormetric Data Security Platform

Application-layer encryption is the most secure form of encryption because it runs at the top of the technology stack. Vormetric Application Encryption is based on the PKCS#11 standard, which has been widely used and proven over the last 20 years. With the solution's APIs, enterprises can easily integrate application encryption into their existing libraries. Together with the DSM, Vormetric Application Encryption removes the complexity and risk of implementing an in-house encryption and key management solution.

REFERENCES

https://en.wikipedia.org/wiki/PKCS_11 www.oasis-open.org/committees/pkcs11

www.oasis-open.org/committees/pkcs11

www.oracle.com/technetwork/database/options/advanced-security/index-099011.html

APPENDIX A. EXAMPLE APPLICATION USING VORMETRIC APPLICATION ENCRYPTION LIBRARY

The following sample code illustrates how to encrypt and decrypt a string using the Vormetric Application Encryption library in C:

```
/*
*********************************************************************
*     Function: encryptAndDecrypt
*     This function first encrypts a block of data using a symmetric key. Then
*     decrypts the ciphertext with the same key to make sure the plain text and
*     decrypted text matches.
*     The caller is responsible for creating a buffer for the output that is
*     of sufficient size.
*********************************************************************
*     Parameters: none
*     Returns: CK_RV
*********************************************************************
*/
static CK_RV encryptAndDecrypt ()
{
/*

************************************
* Supply an IV to use
************************************
*/
```

```
        CK_BYTE iv[] = "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\ x0D\
        x0E\x0F\x10\x00";
/*
*****************************
* Plaintext to encrypt
*****************************
*/
        CK_BYTE    plainText[] = "\xef\xbb\xbf\x53\x65\x63\x74\x69\x6f\x6e\ x0d\
        x0a\x45\x6e\x64\x47\x6c\x6f\x62\x61\x6C\x0D\x0A";
/*
*****************************
*     Specify the mechanism for Encryption. We specify AES CBC mode *with padding,
*     the IV and the size of the plaintext. The other parameters specify a place
*     to hold the resulting ciphertext and the ciphertext length
*****************************
*/
        CK_MECHANISM mechEncryptionPad = { CKM_AES_CBC_PAD, iv, 16 }; CK_BYTE*
        cipherText = NULL_PTR;
        CK_ULONG   cipherTextLen = 0;
/*
*****************************
*     We will reuse the encryption mechanism to decrypt. No need to
*     specify again. The other two parameters holds the decrypted text
*     and decrypted text length.
*****************************
*/
        CK_BYTE* decryptedText = NULL_PTR;
        CK_ULONG   decryptedTextLen = 0;
        CK_RV rc = CKR_OK;
        int status;
/*
*************************
*     C_EncryptInit is called to initialize the encryption operation.
*     Parameters: hSession — The current session handle
      mechEncrytion Pad — the mechanism which which to encrypt hGenKey — the
      handle of the encryption key
*************************
*/
        rc = FunctionListFuncPtr->C_EncryptInit(
        hSession,&mechEncryptionPad,
        hGenKey
        );
        if (rc != CKR_OK)
```

```
        {
        fprintf(stderr, "FAIL: call to C_EncryptInit() failed.\n"); return rc;
        }
        printf("Plain Text: \n"); dumpHexArray(plainText, sizeof(plainText));
      /*
*****************************
*     C_Encrypt is called twice. First call C_Encrypt with NULL as the ciphertext
buffer. C_Encrypt will then return the required buffer size.
*     The user then allocates the memory for the ciphertext buffer and calls * C_
Encrypt again with the buffer with the correct size
*********************************
*/
/* Getthe buffer size required for the ciphertext */
        rc = FunctionListFuncPtr->C_Encrypt(
        hSession,
        plainText,
        sizeof(plainText),
        NULL, &cipherTextLen
        );
       if (rc != CKR_OK)
{
      fprintf (stderr, "FAIL: 1st call to C_Encrypt() failed.\n"); return rc;
}
else
{
      printf ("1st call to C_Encrypt() succeed.\n");
      cipherText = (CK_BYTE *)calloc( 1, sizeof(CK_BYTE) * cipherTextLen );
}
/*then call C_Encrypt to get actual cipherText */ rc = FunctionListFuncPtr->C_
Encrypt(
      hSession,
      plainText,
      sizeof(plainText),
      cipherText,
      &cipherTextLen
      );
if (rc != CKR_OK)
{
      fprintf (stderr, "FAIL: 2nd call to C_Encrypt() failed\n")
      return rc;
}
else
{
```

```
      printf ("2nd call to C_Encrypt() succeed. Ecrypted Text:\n");
      dumpHexArray( cipherText, (int)cipherTextLen );
}
/*C_Decrypt*/
rc = FunctionListFuncPtr->C_DecryptInit(
      hSession,
      &mechEncryptionPad,
      hGenKey
      );
if (rc != CKR_OK)
{
      fprintf (stderr, "FAIL: Call to C_DecryptInit() failed\n"); return rc;
}
/*
*************************************
*     Same workflow as with Encryption.
*     The first time we pass in NULL for the plaintext buffer pass in NULL,
*     to get the decrypted buffer size. Then the buffer of the correct size
*     is allocated, and C_Decrypt is called again to perform decryption.
*************************************
*/

rc = FunctionListFuncPtr->C_Decrypt( hSession,
      cipherText, cipherTextLen,
      NULL, &decryptedTextLen
      );
if (rc != CKR_OK)
{
      fprintf (stderr, "FAIL : 1st call to C_Decrypt() failed.\n"); return rc;
}
else
{
      printf ("1st call to C_Decrypt() succeed.\n");
      decryptedText = (CK_BYTE *)calloc( 1, sizeof(CK_BYTE) *
decryptedTextLen
      );
      if ( NULL == decryptedText)
      return CKR_HOST_MEMORY;
}
/* Second call to C_Decyrpt -- pass in the buffer, to get the decrypted text and
real decrypted size */
rc = FunctionListFuncPtr->C_Decrypt(
```

```
      hSession,
      cipherText,
      cipherTextLen,
      decryptedText,
      &decryptedTextLen
      );
if (rc != CKR_OK)
{
      fprintf (stderr, "FAIL: 2nd call to C_Decrypt() failed\n"); return rc;
}
else
{
      printf ("2nd call to C_Decrypt() succeed. Decrypted Text:\n");
dumpHexArray(decryptedText, (int)decryptedTextLen);
}

/* compare the plaintext and decrypted text */
      status = memcmp( plainText, decryptedText, decryptedTextLen );
      if (status == 0)
{
      printf ("Success! Plain Text and Decrypted Text match! \n");
}
else
{
printf ("Failure!, Plain Text and Decrypted Text do NOT match!! \n");
}
/* cleanup and free memory */
      if (cipherText)
            free (cipherText);
      if (decryptedText)
            free (decryptedText);
      return rc;
}
```

ABOUT VORMETRIC, A THALES COMPANY

Vormetric (@Vormetric) is the industry leader in data security solutions that span physical, virtual and cloud environments. Data is the new currency and Vormetric helps over 1,500 customers, including 17 of the Fortune 30 and many of the world's most security conscious government organizations, to meet compliance requirements and protect what matters—their sensitive data—from both internal and external threats. The company's scalable Vormetric Data Security Platform protects any file, any database and any application—anywhere it resides—with a high performance, market-leading data security platform that incorporates application transparent encryption, privileged user access controls, automation and security intelligence.

For more information, please visit: **www.vormetric.com** and find us on **Twitter @Vormetric**.

GLOBAL HEADQUARTERS

2860 Junction Ave, San Jose, CA 95134
Tel: +1.888.267.3732
Fax: +1.408.844.8638

EMEA HEADQUARTERS

200 Brook Drive
Green Park, Reading, RG2 6UB
United Kingdom
Tel: +44.118.949.7711
Fax: +44.118.949.7001

APAC HEADQUARTERS

Level 42 Suntec Tower Three
8 Temasek Boulevard
Singapore 038988
Tel: +65 6829 2266

082216