RSA LABORATORIES

PKCS #11 v2.20 Amendment 1
PKCS #11 Mechanisms for One-Time Password Tokens

RSA Laboratories

December 27, 2005

TABLE OF CONTENTS

1 Introduction

1.1 Scope

This document is an amendment to PKCS #11 v.20 [1] and describes general PKCS #11 objects, procedures and mechanisms that can be used to retrieve and verify one-time passwords (OTPs) generated by OTP tokens.

1.2 Background

A One-Time Password (OTP) token may be a handheld hardware device, a hardware device connected to a personal computer through an electronic interface such as USB, or a software module resident on a personal computer, which generates one-time passwords that may be used to authenticate a user towards some service. Increasingly, these tokens work in a connected fashion, enabling programmatic retrieval of their OTP values. To meet the needs of applications wishing to access these connected OTP tokens in an interoperable manner, this document extends PKCS #11 [1] to better support these tokens, easing the task for vendors of OTP-consuming applications, and enabling a better user experience.

This document adds basic support of One-Time Password (OTP) tokens to PKCS #11 by defining a common OTP key type with an extensible set of attributes and by describing how PKCS #11 functions can be used to retrieve and verify OTP values generated by an OTP token. It also describes an OTP key generation mechanism that may be used to execute on-token key generation.

Building on the OTP framework, the document specifies the PKCS #11 RSA SecurID™ OTP mechanisms[1], the OATH HOTP mechanisms[2], and the ActivIdentity ACTI mechanisms. Additional mechanisms may be defined separately to support other types of OTP tokens.

A Cryptoki library supporting OTP tokens and the PKCS #11 v2.20 extensions defined herein may also support existing PKCS #11 cryptographic tokens. It is also envisioned that certain tokens will offer both OTP functionality and traditional cryptographic token functionalities such as encryption, decryption, etc.

1.3 Document organization

The organization of this document is as follows:

- Section 1 is an introduction.
- Section 2 provides an overview description of the support for OTP tokens in PKCS #11 defined herein.
- Section 3 defines the new OTP key object type and its attributes.
- Section 4 defines a new OTP-related notification.
- Section 5 defines specific OTP mechanisms.
- Appendix A collects defined PKCS #11 constants.
- Appendix B provides example usages of the OTP mechanisms.
- Appendices C, D, and E cover intellectual property issues, give references to other publications and standards, and provide general information about the One-Time Password Specifications.

2 Usage overview

OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases can be categorized according to the type of sought functionality.

[1] RSA SecurID® two-factor authentication is a symmetric authentication method which is patented by RSA Security. A user authenticates by submitting a one-time password (OTP), or PASSCODE value generated by an RSA SecurID token. The RSA SecurID token may be a handheld hardware device, a hardware device connected to a personal computer through an electronic interface such as USB, or a software module resident on the personal computer.

[2] The HOTP algorithm is work in progress, currently defined in the IETF draft http://www.ietf.org/internet-drafts/draft-mraihi-oath-hmac-otp-04.txt developed by the Open Authentication initiative (http://www.openauthentication.org).

2.1 Case 1: Generation of OTP values

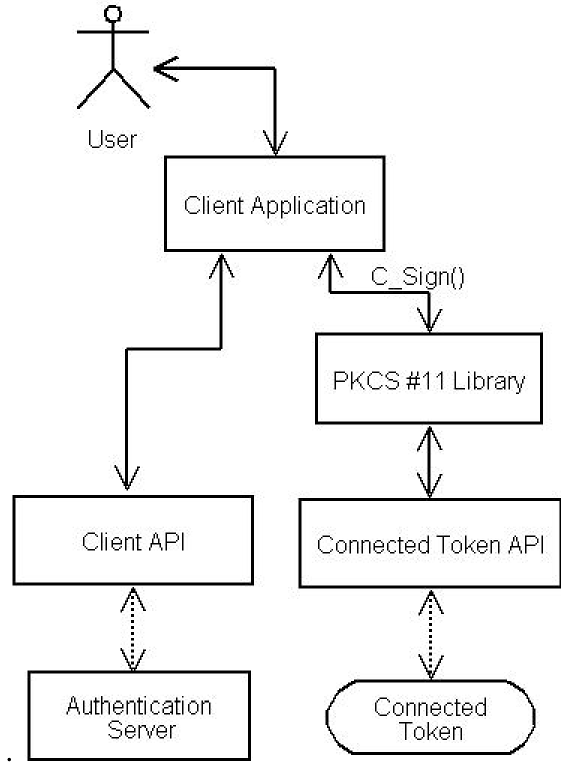

Figure 1: Retrieving OTP values through C_Sign

Figure 1 shows an integration of PKCS #11 into an application that needs to authenticate users holding OTP tokens. In this particular example, a connected hardware token is used, but a software token is equally possible. The application invokes **C_Sign** to retrieve the OTP value from the token. In the example, the application then passes the retrieved OTP value to a client API that sends it via the network to an authentication server. The client API may implement a standard authentication protocol such as RADIUS [2] or EAP [3], or a proprietary protocol such as that used by RSA Security's ACE/Agent® software.

2.2 Case 2: Verification of provided OTP values

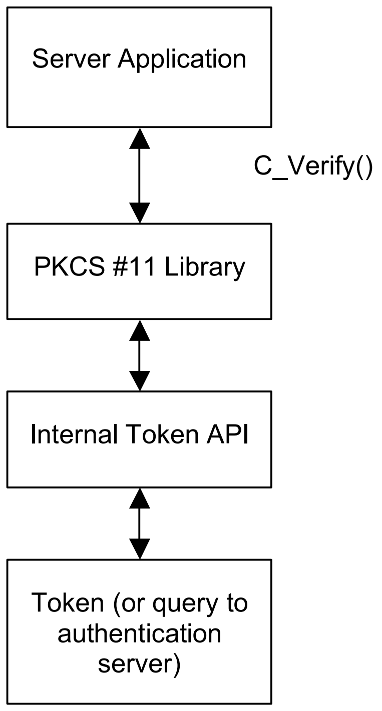

Figure 2: Server-side verification of OTP values

Figure 2 illustrates the server-side equivalent of the scenario depicted in Figure 1. In this case, a server application invokes **C_Verify** with the received OTP value as the signature value to be verified.

2.3 Case 3: Generation of OTP keys

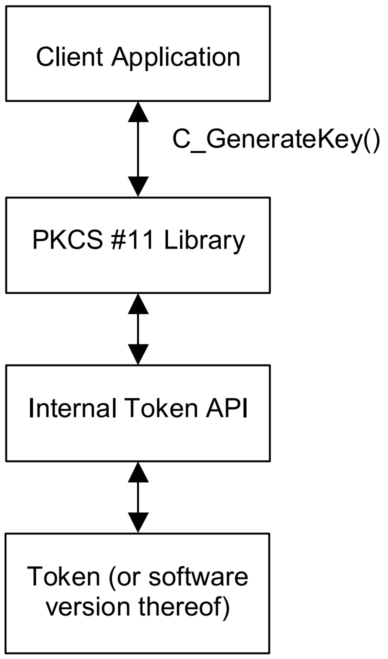

Figure 3: Generation of an OTP key

Figure 3 shows an integration of PKCS #11 into an application that generates OTP keys. The application invokes **C_GenerateKey** to generate an OTP key of a particular type on the token. The key may subsequently be used as a basis to generate OTP values.

3 OTP objects

3.1 Key objects

OTP key objects (object class **CKO_OTP_KEY**) hold secret keys used by OTP tokens. The following table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys, all of which are inherited by this class:

Table 1: Common OTP key attributes

Attribute	Data type	Meaning

CKA_OTP_FORMAT	CK_ULONG	Format of OTP values produced with this key:
		CK_OTP_FORMAT_DECIMAL = Decimal (default) (UTF8-encoded)
		CK_OTP_FORMAT_HEXADECIMAL = Hexadecimal (UTF8-encoded)
		CK_OTP_FORMAT_ALPHANUMERIC = Alphanumeric (UTF8-encoded)
		CK_OTP_FORMAT_BINARY = Only binary values.

CKA_OTP_LENGTH[9]	CK_ULONG	Default length of OTP values (in the CKA_OTP_FORMAT) produced with this key.
CKA_OTP_USER_FRIENDLY_MODE[9]	CK_BBOOL	Set to CK_TRUE when the token is capable of returning OTPs suitable for human consumption. See the description of CKF_USER_FRIENDLY_OTP below.
CKA_OTP _CHALLENGE_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A challenge must be supplied. CK_OTP_PARAM_OPTIONAL = A challenge may be supplied but need not be. CK_OTP_PARAM_IGNORED = A challenge, if supplied, will be ignored.
CKA_OTP_TIME_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A time value must be supplied. CK_OTP_PARAM_OPTIONAL = A time value may be supplied but need not be. CK_OTP_PARAM_IGNORED = A time value, if supplied, will be ignored.
CKA_OTP_COUNTER_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A counter value must be supplied. CK_OTP_PARAM_OPTIONAL = A

Attribute	Data type	Meaning
		counter value may be supplied but need not be. CK_OTP_PARAM_IGNORED = A counter value, if supplied, will be ignored.
CKA_OTP_PIN_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A PIN value must be supplied. CK_OTP_PARAM_OPTIONAL = A PIN value may be supplied but need not be (if not supplied, then library will be responsible for collecting it) CK_OTP_PARAM_IGNORED = A PIN value, if supplied, will be ignored.
CKA_OTP_COUNTER	Byte array	Value of the associated internal counter. Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_TIME	RFC 2279 string	Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. *ulValueLen*= 0).
CKA_OTP_USER_IDENTIFIER	RFC 2279 string	Text string that identifies a user associated with the OTP key (may be used to enhance the user experience). Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_SERVICE_IDENTIFIER	RFC 2279 string	Text string that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_SERVICE_LOGO	Byte array	Logotype image that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_SERVICE_LOGO_TYPE	RFC 2279 string	MIME type of the CKA_OTP_SERVICE_LOGO attribute value. Default value is empty (i.e. *ulValueLen* = 0).
CKA_VALUE[1, 4, 6, 7]	Byte array	Value of the key.
CKA_VALUE_LEN[2, 3]	CK_ULONG	Length in bytes of key value.

Refer to Table 15 in [1] for table footnotes.

Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-PKCS #11 application should therefore always consult the state of the CKA_OTP_PIN_REQUIREMENT attribute before each call to **C_SignInit**, as the value of this attribute may change dynamically.

For OTP tokens with multiple keys, the keys may be enumerated using **C_FindObjects**. The **CKA_OTP_SERVICE_IDENTIFIER** and/or the **CKA_OTP_SERVICE_LOGO** attribute may be used to distinguish between keys. The actual choice of key for a particular operation is however application-specific and beyond the scope of this document.

For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set in accordance with [1], Table 27.

4 OTP-related notifications

This document extends the set of defined notifications as follows:

CKN_OTP_CHANGED	Cryptoki is informing the application that the OTP for a key on a connected token just changed. This notification is particularly useful when applications wish to display the current OTP value for time-based mechanisms.

5 OTP mechanisms

The following table shows, for the OTP mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

Table 2: OTP mechanisms vs. applicable functions

Mechanism	**Functions**						
	Encrypt & Decrypt	**Sign & Verify**	**SR & VR**[1]	**Digest**	**Gen. Key/ Key Pair**	**Wrap & Unwrap**	**Derive**
CKM_SECURID_KEY_GEN					✓		
CKM_SECURID		✓					
CKM_HOTP_KEY_GEN					✓		
CKM_HOTP		✓					
CKM_ACTI_KEY_GEN					✓		
CKM_ACTI		✓					

The remainder of this section will present in detail the OTP mechanisms and the parameters that are supplied to them.

5.1 OTP mechanism parameters

♦ CK_PARAM_TYPE

CK_PARAM_TYPE is a value that identifies an OTP parameter type. It is defined as follows:

```
typedef CK_ULONG CK_PARAM_TYPE;
```

The following **CK_PARAM_TYPE** types are defined:

Table 3: OTP parameter types

Parameter	**Data type**	**Meaning**
CK_OTP_PIN	RFC 2279 string	A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values.
CK_OTP_CHALLENGE	Byte array	Challenge to use when computing or verifying challenge-based OTP values.
CK_OTP_TIME	RFC 2279 string	UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values.
CK_OTP_COUNTER	Byte array	Counter value to use when computing or verifying counter-based OTP values.
CK_OTP_FLAGS	CK_FLAGS	Bit flags indicating the characteristics of the sought OTP as defined below.
CK_OTP_OUTPUT_LENGTH	CK_ULONG	Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported.
CK_OTP_FORMAT	CK_ULONG	Returned OTP format (allowed values are the same as for CKA_OTP_FORMAT). This parameter is only intended for **C_Sign** output, see below. When not present, the returned OTP format will be the same as the value of the CKA_OTP_FORMAT attribute for the key in question.
CK_OTP_VALUE	Byte array	An actual OTP value. This parameter type is intended for **C_Sign** output, see below.

The following table defines the possible values for the CK_OTP_FLAGS type:

Table 4: OTP Mechanism Flags

Bit flag	**Mask**	**Meaning**
CKF_NEXT_OTP	0x00000001	True (i.e. set) if the OTP computation shall be for the next OTP, rather than the current one (current being interpreted in the context of the algorithm, e.g. for the current counter value or current time window). A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if the CKF_NEXT_OTP flag is set and the OTP mechanism in question does not support the concept of "next" OTP or the library is not capable of generating the next OTP[3].
CKF_EXCLUDE_TIME	0x00000002	True (i.e. set) if the OTP computation must not include a time value. Will have an effect only on mechanisms that do include a time value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_COUNTER	0x00000004	True (i.e. set) if the OTP computation must not include a counter value. Will have an effect only on mechanisms that do include a counter value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_CHALLENGE	0x00000008	True (i.e. set) if the OTP computation must not include a challenge. Will have an effect only on mechanisms that do include a challenge in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.

[3] Applications that may need to retrieve the next OTP should be prepared to handle this situation. For example, an application could store the OTP value returned by C_Sign so that, if a next OTP is required, it can compare it to the OTP value returned by subsequent calls to C_Sign should it turn out that the library does not support the CKF_NEXT_OTP flag.

Bit flag	**Mask**	**Meaning**
CKF_EXCLUDE_PIN	0x00000010	True (i.e. set) if the OTP computation must not include a PIN value. Will have an effect only on mechanisms that do include a PIN in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_USER_FRIENDLY_OTP	0x00000020	True (i.e. set) if the OTP returned shall be in a form suitable for human consumption. If this flag is set, and the call is successful, then the returned CK_OTP_VALUE shall be a UTF8-encoded printable string. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if this flag is set when CKA_OTP_USER_FRIENDLY_MODE for the key in question is CK_FALSE.

Note: Even if CKA_OTP_FORMAT is not set to CK_OTP_FORMAT_BINARY, then there may still be value in setting the CKF_USER_FRIENDLY flag (assuming CKA_USER_FRIENDLY_MODE is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned OTP value to a user should not set the CKF_USER_FRIENDLY_OTP flag.

♦ **CK_OTP_PARAM; CK_OTP_PARAM_PTR**

CK_OTP_PARAM is a structure that includes the type, value, and length of an OTP parameter. It is defined as follows:

```
typedef struct CK_OTP_PARAM {
   CK_PARAM_TYPE type;
   CK_VOID_PTR pValue;
   CK_ULONG ulValueLen;
} CK_OTP_PARAM;
```

The fields of the structure have the following meanings:

type	the parameter type
pValue	pointer to the value of the parameter
ulValueLen	length in bytes of the value

If a parameter has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. Note that *pValue* is a "void" pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

CK_OTP_PARAM_PTR is a pointer to a **CK_OTP_PARAM.**

♦ **CK_OTP_PARAMS; CK_OTP_PARAMS_PTR**

CK_OTP_PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:

```
typedef struct CK_OTP_PARAMS {
   CK_OTP_PARAM_PTR pParams;
   CK_ULONG ulCount;
} CK_OTP_PARAMS;
```

The fields of the structure have the following meanings:

pParams	pointer to an array of OTP parameters
ulCount	the number of parameters in the array

CK_OTP_PARAMS_PTR is a pointer to a **CK_OTP_PARAMS**.

When calling **C_SignInit** or **C_VerifyInit** with a mechanism that takes a **CK_OTP_PARAMS** structure as a parameter, the **CK_OTP_PARAMS** structure shall be populated in accordance with the **CKA_OTP_*X*_REQUIREMENT** key attributes for the identified key, where ***X*** is **PIN**, **CHALLENGE**, **TIME**, or **COUNTER**.

For example, if **CKA_OTP_TIME_REQUIREMENT** = CK_OTP_PARAM_MANDATORY, then the **CK_OTP_TIME** parameter shall be present. If **CKA_OTP_TIME_REQUIREMENT** = CK_OTP_PARAM_OPTIONAL, then a **CK_OTP_TIME** parameter may be present. If it is not present, then the library may collect it (during the **C_Sign** call). If **CKA_OTP_TIME_REQUIREMENT** = CK_OTP_PARAM_IGNORED, then a provided **CK_OTP_TIME** parameter will always be ignored. Additionally, a provided **CK_OTP_TIME** parameter will always be ignored if CKF_EXCLUDE_TIME is set in a **CK_OTP_FLAGS** parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (**CKR_MECHANISM_PARAM_INVALID**) to set the CKF_EXCLUDE_TIME flag when the **CKA_TIME_REQUIREMENT** attribute is CK_OTP_PARAM_MANDATORY.

The above discussion holds for all **CKA_OTP_*X*_REQUIREMENT** attributes (*i.e.*, **CKA_OTP_PIN_REQUIREMENT**, **CKA_OTP_CHALLENGE_REQURIEMENT**, **CKA_OTP_COUNTER_REQUIREMENT**, **CKA_OTP_TIME_REQUIREMENT**). A library may set a particular **CKA_OTP_*X*_REQUIREMENT** attribute to CK_OTP_PARAM_OPTIONAL even if it is required by the mechanism as long as the token (or the library itself) has the capability of providing the value to the computation. One example of this is a token with an on-board clock.

In addition, applications may use the **CK_OTP_FLAGS**, the **CK_OTP_OUTPUT_FORMAT** and the **CK_OUTPUT_LENGTH** parameters to set additional parameters.

♦ **CK_OTP_SIGNATURE_INFO, CK_OTP_SIGNATURE_INFO_PTR**

CK_OTP_SIGNATURE_INFO is a structure that is returned by all OTP mechanisms in successful calls to **C_Sign** (**C_SignFinal**). The structure informs applications of actual parameter values used in particular OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key belongs to the class CKO_OTP_KEY and is defined as follows:

```
typedef struct CK_OTP_SIGNATURE_INFO {
   CK_OTP_PARAM_PTR pParams;
   CK_ULONG ulCount;
} CK_OTP_SIGNATURE_INFO;
```

The fields of the structure have the following meanings:

pParams	pointer to an array of OTP parameter values
ulCount	the number of parameters in the array

After successful calls to **C_Sign** or **C_SignFinal** with an OTP mechanism, the *pSignature* parameter will be set to point to a **CK_OTP_SIGNATURE_INFO** structure. One of the parameters in this structure will be the OTP value itself, identified with the **CK_OTP_VALUE** tag. Other parameters may be present for informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP validations, authentication protocols may permit authenticating parties to send some or all of these parameters in addition to OTP values themselves. Applications should therefore check for their presence in returned **CK_OTP_SIGNATURE_INFO** values whenever such circumstances apply.

Since **C_Sign** and **C_SignFinal** follows the convention described in Section 11.2 of [1] on producing output, a call to **C_Sign** (or **C_SignFinal**) with *pSignature* set to NULL_PTR will return (in the *pulSignatureLen* parameter) the required number of bytes to hold the **CK_OTP_SIGNATURE_INFO** structure *as well as all the data in all its **CK_OTP_PARAM** components*. If an application allocates a memory block based on this information, it shall therefore not subsequently de-allocate components of such a received value but rather de-allocate the complete **CK_OTP_PARAMS** structure itself. A Cryptoki library that is called with a non-NULL *pSignature* pointer will assume that it points to a *contiguous* memory block of the size indicated by the *pulSignatureLen* parameter.

When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAMS** structure returned by a call to **C_Sign**. The **CK_OTP_PARAMS** value supplied in the **C_VerifyInit** call sets the values to use in the verification operation.

CK_OTP_SIGNATURE_INFO_PTR points to a **CK_OTP_SIGNATURE_INFO.**

5.2 RSA SecurID

5.2.1 RSA SecurID secret key objects

RSA SecurID secret key objects (object class **CKO_OTP_KEY,** key type **CKK_SECURID**) hold RSA SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in addition to the common attributes defined for this object class:

Table 5: RSA SecurID secret key object attributes

Attribute	Data type	Meaning
CKA_OTP_TIME_INTERVAL[1]	CK_ULONG	Interval between OTP values produced with this key, in seconds. Default is 60.

Refer to Table 15 in [1] for table footnotes.

The following is a sample template for creating an RSA SecurID secret key object:

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_SECURID;
CK_DATE endDate = {...};
CK_UTF8CHAR label[] = “RSA SecurID secret key object”;
CK_BYTE keyId[]= {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_ULONG needPIN = CK_OTP_PARAM_MANDATORY;
CK_ULONG timeInterval = 60;
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_END_DATE, &endDate, sizeof(endDate)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_ID, keyId, sizeof(keyId)},
   {CKA_OTP_FORMAT, &outputFormat,
        sizeof(outputFormat)},
   {CKA_OTP_LENGTH, &outputLength,
        sizeof(outputLength)},
   {CKA_OTP_PIN_REQUIREMENT, &needPIN, sizeof(needPIN)},
   {CKA_OTP_TIME_INTERVAL, &timeInterval,
        sizeof(timeInterval)},
   {CKA_VALUE, value, sizeof(value)}
};
```

5.2.2 RSA SecurID key generation

The RSA SecurID key generation mechanism, denoted **CKM_SECURID_KEY_GEN**, is a key generation mechanism for the RSA SecurID algorithm.

It does not have a parameter.

The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE_LEN**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be specified in the template for the key, or else are assigned default initial values

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of SecurID key sizes, in bytes.

5.2.3 RSA SecurID OTP generation and validation

CKM_SECURID is the mechanism for the retrieval and verification of RSA SecurID OTP values.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

When signing or verifying using the **CKM_SECURID** mechanism, *pData* shall be set to NULL_PTR and *ulDataLen* shall be set to 0.

5.2.4 Return values

Support for the **CKM_SECURID** mechanism extends the set of return values for **C_Verify** with the following values:

- CKR_NEW_PIN_MODE: The supplied OTP was not accepted and the library requests a new OTP computed using a new PIN. The new PIN is set through means out of scope for this document.
- CKR_NEXT_OTP: The supplied OTP was correct but indicated a larger than normal drift in the token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the application should provide the next one-time password to the library for verification.

5.3 OATH HOTP

5.3.1 OATH HOTP secret key objects

HOTP secret key objects (object class **CKO_OTP_KEY,** key type **CKK_HOTP**) hold generic secret keys and associated counter values.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

For HOTP keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for a **CK_OTP_COUNTER** value in a **CK_OTP_PARAM** structure.

The following is a sample template for creating a HOTP secret key object:

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_HOTP;
```

```
CK_UTF8CHAR label[] = “HOTP secret key object”;
CK_BYTE keyId[]= {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_END_DATE, &endDate, sizeof(endDate)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_ID, keyId, sizeof(keyId)},
   {CKA_OTP_FORMAT, &outputFormat,
        sizeof(outputFormat)},
   {CKA_OTP_LENGTH, &outputLength,
        sizeof(outputLength)},
   {CKA_OTP_COUNTER, counterValue,
        sizeof(counterValue)},
   {CKA_VALUE, value, sizeof(value)}
};
```

5.3.2 HOTP key generation

The HOTP key generation mechanism, denoted **CKM_HOTP_KEY_GEN**, is a key generation mechanism for the HOTP algorithm.

It does not have a parameter.

The mechanism generates HOTP keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_OTP_COUNTER**, **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the HOTP key type may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of HOTP key sizes, in bytes.

5.3.3 HOTP OTP generation and validation

CKM_HOTP is the mechanism for the retrieval and verification of HOTP OTP values based on the current internal counter, or a provided counter.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism, *pData* shall be set to NULL_PTR and *ulDataLen* shall be set to 0.

For verify operations, the counter value **CK_OTP_COUNTER** must be provided as a **CK_OTP_PARAM** parameter to **C_VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAMS** structure in the case of an earlier call to **C_Sign**.

5.4 ActivIdentity ACTI

5.4.1 ACTI secret key objects

ACTI secret key objects (object class **CKO_OTP_KEY,** key type **CKK_ACTI**) hold ActivIdentity ACTI secret keys.

For ACTI keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for the **CK_OTP_COUNTER** value in the **CK_OTP_PARAM** structure.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

The **CKA_OTP_TIME** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

The following is a sample template for creating an ACTI secret key object:

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_ACTI;
CK_UTF8CHAR label[] = “ACTI secret key object”;
CK_BYTE keyId[]= {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_END_DATE, &endDate, sizeof(endDate)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_ID, keyId, sizeof(keyId)},
   {CKA_OTP_FORMAT, &outputFormat,
   sizeof(outputFormat)},
   {CKA_OTP_LENGTH, &outputLength,
   sizeof(outputLength)},
   {CKA_OTP_COUNTER, counterValue,
   sizeof(counterValue)},
   {CKA_VALUE, value, sizeof(value)}
};
```

5.4.2 ACTI key generation

The ACTI key generation mechanism, denoted **CKM_ACTI_KEY_GEN**, is a key generation mechanism for the ACTI algorithm.

It does not have a parameter.

The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of ACTI key sizes, in bytes.

5.4.3 ACTI OTP generation and validation

CKM_ACTI is the mechanism for the retrieval and verification of ACTI OTP values.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

When signing or verifying using the **CKM_ACTI** mechanism, *pData* shall be set to NULL_PTR and *ulDataLen* shall be set to 0.

When verifying an OTP value using the **CKM_ACTI** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAMS** structure in the case of an earlier call to **C_Sign**.

A. Manifest constants

Note: A C or C++ source file in a Cryptoki application or library can define all the types, mechanisms, and other constants described here by including the header file otp-pkcs11.h. When including the otp-pkcs11.h header file, it should be preceded by an inclusion of the top-level Cryptoki header file pkcs11.h, and the source file must also specify the preprocessor directives indicated in Section 8 of [1].

A.1 Object classes

```
#define CKO_OTP_KEY                   0x00000008
```

A.2 Key types

```
#define CKK_SECURID                   0x00000022
#define CKK_HOTP                      0x00000023
#define CKK_ACTI                      0x00000024
```

A.3 Mechanisms

```
#define CKM_SECURID_KEY_GEN           0x00000280
#define CKM_SECURID                   0x00000282

#define CKM_HOTP_KEY_GEN              0x00000290
#define CKM_HOTP                      0x00000291

#define CKM_ACTI_KEY_GEN              0x000002A0
#define CKM_ACTI                      0x000002A1
```

A.4 Attributes

```
#define CKA_OTP_FORMAT                 0x00000220
#define CKA_OTP_LENGTH                 0x00000221
#define CKA_OTP_TIME_INTERVAL          0x00000222
#define CKA_OTP_USER_FRIENDLY_MODE     0x00000223
#define CKA_OTP_CHALLENGE_REQUIREMENT  0x00000224
#define CKA_OTP_TIME_REQUIREMENT       0x00000225
#define CKA_OTP_COUNTER_REQUIREMENT    0x00000226
#define CKA_OTP_PIN_REQUIREMENT        0x00000227
#define CKA_OTP_USER_IDENTIFIER        0x0000022A
#define CKA_OTP_SERVICE_IDENTIFIER     0x0000022B
#define CKA_OTP_SERVICE_LOGO           0x0000022C
#define CKA_OTP_SERVICE_LOGO_TYPE      0x0000022D
#define CKA_OTP_COUNTER                0x0000022E
#define CKA_OTP_TIME                   0x0000022F
```

A.5 Attribute constants

```
#define CK_OTP_FORMAT_DECIMAL         0
```

```
#define CK_OTP_FORMAT_HEXADECIMAL    1
#define CK_OTP_FORMAT_ALPHANUMERIC   2
#define CK_OTP_FORMAT_BINARY         3

#define CK_OTP_PARAM_IGNORED         0
#define CK_OTP_PARAM_OPTIONAL        1
#define CK_OTP_PARAM_MANDATORY       2
```

A.6 Other constants

```
#define CK_OTP_VALUE                 0
#define CK_OTP_PIN                   1
#define CK_OTP_CHALLENGE             2
#define CK_OTP_TIME                  3
#define CK_OTP_COUNTER               4
#define CK_OTP_FLAGS                 5
#define CK_OTP_OUTPUT_LENGTH         6
#define CK_OTP_FORMAT                7

#define CKF_NEXT_OTP                 0x00000001
#define CKF_EXCLUDE_TIME             0x00000002
#define CKF_EXCLUDE_COUNTER          0x00000004
#define CKF_EXCLUDE_CHALLENGE        0x00000008
#define CKF_EXCLUDE_PIN              0x00000010
#define CKF_USER_FRIENDLY_OTP        0x00000020
```

A.7 Notifications

```
#define CKN_OTP_CHANGED              1
```

A.8 Return values

```
#define CKR_NEW_PIN_MODE   0x000001B0
#define CKR_NEXT_OTP       0x000001B1
```

B. Example code

B.1 Disclaimer concerning sample code

For the sake of brevity, sample code presented herein is somewhat incomplete. In particular, initial steps needed to create a session with a cryptographic token are not shown, and the error handling is simplified.

B.2 OTP retrieval

The following sample code snippet illustrates the retrieval of an OTP value from an OTP token using the **C_Sign** function. The sample demonstrates the generality of the approach described herein and does not include any OTP mechanism-specific knowledge.

```
    CK_SESSION_HANDLE hSession;
    CK_OBJECT_HANDLE hKey;
```

```
CK_RV rv;
CK_SLOT_ID slotId;
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)} };
CK_UTF8CHAR time[] = {...};
/* UTC time value for OTP, or NULL */
CK_UTF8CHAR pin[] = {...};
/* User PIN, or NULL */
CK_BYTE counter[] = {...};
/* Counter value, or NULL */
CK_BYTE challenge[] = {...};
/* Challenge, or NULL */
CK_MECHANISM_TYPE_PTR allowedMechanisms = NULL_PTR;
CK_MECHANISM_INFO mechanismInfo;
CK_MECHANISM mechanism;
CK_ULONG i, ulOTPLen, ulKeyCount, ulChalReq, ulPINReq,
      ulTimeReq, ulCounterReq;
CK_ATTRIBUTE mechanisms[] = { {CKA_ALLOWED_MECHANISMS,
      NULL_PTR, 0} };
CK_ATTRIBUTE attributes[] = {
  {CKA_OTP_CHALLENGE_REQUIREMENT, &ulChalReq,
      sizeof(ulChalReq)},
  {CKA_OTP_PIN_REQUIREMENT, &ulPINReq,
      sizeof(ulPINReq)},
  {CKA_OTP_COUNTER_REQUIREMENT, &ulCounterReq,
      sizeof(ulCounterReq)},
  {CKA_OTP_TIME_REQUIREMENT, &ulTimeReq,
      sizeof(ulTimeReq)} };

CK_OTP_PARAM param[4];
CK_OTP_PARAMS params;
CK_BYTE *pOTP; /* Storage for OTP result */

do {

  /* N.B.: Minimal error and memory handling in this
     sample code. */

  /* Find first OTP key on the token. */
  if ((rv = C_FindObjectsInit(hSession, template, 1))
      != CKR_OK) {
    break;
  };
  if ((rv = C_FindObjects(hSession, &hKey, 1,
      &ulKeyCount)) != CKR_OK) {
    break;
  };
  if (ulKeyCount == 0) {
    /* No OTP key found */
    break;
  }
  rv = C_FindObjectsFinal(hSession);

  /* Find a suitable OTP mechanism. */
  if ((rv = C_GetAttributeValue(hSession, hKey,
      mechanisms, 1)) != CKR_OK) {
    break;
  };

  if ((allowedMechanisms = (CK_MECHANISM_TYPE_PTR)
      malloc(mechanisms[0].ulValueLen)) == 0) {
    break;
  };

  mechanisms[0].pValue = allowedMechanisms;
  if ((rv = C_GetAttributeValue(hSession, hKey,
      mechanisms, 1)) != CKR_OK) {
    break;
  };

  for (i = 0; i < mechanisms[0].ulValueLen/
      sizeof(CK_MECHANISM_TYPE); ++i) {
    if ((rv = C_GetMechanismInfo(slotId,
      allowedMechanisms[i], &mechanismInfo)) ==
      CKR_OK) {
      if (mechanismInfo.flags & CKF_SIGN) {
        break;
      }
    }
  }

  if (i == mechanisms[0].ulValueLen) {
    break;
  }

  mechanism.mechanism = allowedMechanisms[i];
  free(allowedMechanisms);

  /* Set required mechanism parameters based on
     the key attributes. */
  if ((rv = C_GetAttributeValue(hSession, hKey,
       attributes, sizeof(attributes) /
       sizeof(attributes[0]))) != CKR_OK) {
```

```
      break;
    }

    i = 0;
    if (ulPINReq == CK_OTP_PARAM_MANDATORY) {
      /* PIN value needed. */
      param[i].type = CK_OTP_PIN;
      param[i].pValue = pin;
      param[i++].ulValueLen = sizeof(pin) - 1;
    }
    if (ulChalReq == CK_OTP_PARAM_MANDATORY) {
      /* Challenge neded. */
      param[i].type = CK_OTP_CHALLENGE;
      param[i].pValue = challenge;
      param[i++].ulValueLen = sizeof(challenge);
    }
    if (ulTimeReq == CK_OTP_PARAM_MANDATORY) {
      /* Time needed (would not normally be
         the case if token has its own clock). */
      param[i].type = CK_OTP_TIME;
      param[i].pValue = time;
      param[i++].ulValueLen = sizeof(time) -1;
    }
    if (ulCounterReq == CK_OTP_PARAM_MANDATORY) {
      /* Counter value needed (would not normally
         be the case if token has its own counter.*/
      param[i].type = CK_OTP_COUNTER;
      param[i].pValue = counter;
      param[i++].ulValueLen = sizeof(counter);
    }

    params.pParams = param;
    params.ulCount = i;

    mechanism.pParameter = &params;
    mechanism.ulParameterLen = sizeof(params);

    /* Sign to get the OTP value. */
    if ((rv = C_SignInit(hSession, &mechanism, hKey))
      != CKR_OK) {
      break;
    }

    /* Get the buffer length needed for the OTP Value
       and any associated data. */
    if ((rv = C_Sign(hSession, NULL_PTR, 0, NULL_PTR,
        &ulOTPLen)) != CKR_OK) {
```

```
        break;
      };

      if ((pOTP = malloc(ulOTPLen)) == NULL_PTR) {
        break;
      };

      /* Get the actual OTP value and any
         associated data. */
      if ((rv = C_Sign(hSession, NULL_PTR, 0, pOTP,
               &ulOTPLen)) != CKR_OK) {
        break;
      }

      /* Traverse the returned pOTP here. The actual
         OTP value is in CK_OTP_VALUE in pOTP. */

    } while (0);
```

B.3 User-friendly mode OTP token

This sample demonstrates an application retrieving a user-friendly OTP value. The code is the same as in B.1 except for the following:

```
  /* Add these variable declarations */

    CK_FLAGS flags = CKF_USER_FRIENDLY_OTP;
    CK_BBOOL bUserFriendlyMode;
    CK_ULONG ulFormat;

  /* Replace the declaration of the "attributes" and the
     "param" variables with: */

    CK_ATTRIBUTE attributes[] = {
      {CKA_OTP_CHALLENGE_REQUIREMENT, &ulChalReq,
      sizeof(ulChalReq)},
      {CKA_OTP_PIN_REQUIREMENT, &ulPINReq,
      sizeof(ulPINReq)},
      {CKA_OTP_COUNTER_REQUIREMENT, &ulCounterReq,
      sizeof(ulCounterReq)},
      {CKA_OTP_TIME_REQUIREMENT, &ulTimeReq,
      sizeof(ulTimeReq)},
      {CKA_OTP_USER_FRIENDLY_MODE, &bUserFriendlyMode,
      sizeof(bUserFriendlyMode)},
      {CKA_OTP_FORMAT, &ulFormat,
      sizeof(ulFormat)}
    };
```

```
        CK_OTP_PARAM param[5];

    /* Replace the assignment of the "pParam" component
       of the "params" variable with: */

        if (bUserFriendlyMode == CK_TRUE) {
          /* Token supports user-friendly OTPs */
          param[i].type = CK_OTP_FLAGS;
          param[i].pValue = &flags;
          param[i++].ulValueLen = sizeof(CK_FLAGS);
        } else if (ulFormat == CK_OTP_FORMAT_BINARY) {
          /* Some kind of error since a user-friendly
             OTP cannot be returned to an application
             that needs it. */
          break;
        };

        params.pParams = param;

    /* Further processing is as in B.1. */
```

B.4 OTP verification

The following sample code snippet illustrates the verification of an OTP value from an RSA SecurID token, using the **C_Verify** function. The desired UTC time, if a time is specified, is supplied in the CK_OTP_PARAMS structure, as is the user's PIN.

```
    CK_SESSION_HANDLE hSession;
    CK_OBJECT_HANDLE hKey;
    CK_UTF8CHAR time[] = {...};
    /* UTC time value for OTP, or NULL */
    CK_UTF8CHAR pin[] = {...};
    /* User PIN or NULL (if collected by library) */
    CK_OTP_PARAM param[] = {
      {CK_OTP_TIME, time, sizeof(time)-1},
      {CK_OTP_PIN, pin, sizeof(pin)-1}
    };
    CK_OTP_PARAMS params = {param, 2};
    CK_MECHANISM mechanism = {CKM_SECURID, &params,
            sizeof(params)};
    CK_ULONG ulKeyCount;
    CK_RV rv;
    CK_BYTE OTP[] = {...};     /* Supplied OTP value. */
    CK_ULONG ulOTPLen = strlen((CK_CHAR_PTR)OTP);
    CK_OBJECT_CLASS class = CKO_OTP_KEY;
    CK_KEY_TYPE keyType = CKK_SECURID;
```

```
   CK_ATTRIBUTE template[] = {
     {CKA_CLASS, &class, sizeof(class)},
     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   };

   /* Find the RSA SecurID key on the token. */
   rv = C_FindObjectsInit(hSession, template, 2);
   if (rv == CKR_OK) {
      rv = C_FindObjects(hSession, &hKey, 1, &ulKeyCount);
      rv = C_FindObjectsFinal(hSession);
   }

   if ((rv != CKR_OK) || (ulKeyCount == 0)) {
      printf(" \nError: unable to find RSA SecurID key on
           token.\n");
      return(rv);
}

   rv = C_VerifyInit(hSession, &mechanism, hKey);
   if (rv == CKR_OK) {
     ulOTPLen = sizeof(OTP);
     rv = C_Verify(hSession, NULL_PTR, 0, OTP, ulOTPLen);
   }

   switch(rv) {
       case CKR_OK:
           printf("\nSupplied OTP value verified.\n");
           break;

       case CKR_SIGNATURE_INVALID:
           printf("\nSupplied OTP value not verified.\n");
           break;

       default:
           printf("\nError:Unable to verify OTP value.\n");
           break;
   }

   return(rv);
```

C. Intellectual property considerations

RSA Security makes no patent claims on the general constructions described in this document, although specific underlying techniques may be covered. The RSA SecurID technology is covered by a number of US patents (and foreign counterparts), in particular US patent nos. 4,856,062, 4,885,778, 5,097,505, 5,168,520, and 5,657,388. Additional patents are pending.

D. References

[1] RSA Laboratories, *PKCS #11: Cryptographic Token Interface Standard*. Version 2.20, June 2004. URL: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf.

[2] Rigney et al, "Remote Authentication Dial In User Service (RADIUS)", IETF RFC2865, June 2000. URL: http://ietf.org/rfc/rfc2865.txt.

[3] Aboba et al, "Extensible Authentication Protocol (EAP)", IETF RFC 3748, June 2004. URL: http://ietf.org/rfc/rfc3748.txt.

E. About OTPS

The *One-Time Password Specifications* are documents produced by RSA Laboratories in cooperation with secure systems developers for the purpose of simplifying integration and management of strong authentication technology into secure applications, and to enhance the user experience of this technology.

Further development of the OTPS series will occur through mailing list discussions and occasional workshops, and suggestions for improvement are welcome. As four our PKCS documents, results may also be submitted to standards forums. For more information, contact:

OTPS Editor
RSA Laboratories
174 Middlesex Turnpike
Bedford, MA 01730 USA
otps-editor@rsasecurity.com
http://www.rsasecurity.com/rsalabs/