Analysis of ESAPI 2.0's Key Derivation Function

Jeffrey Walton
jeffrey.walton@softwareintegrity.com

1. Introduction

OWASP is the Open Web Application Security Project and located at http://www.owasp.org/. The project, among other goals, encourages sound use cryptography by users who might be less savvy. ESAPI is OWASP's Enterprise Security API. The API provides implementations in a number of cryptographic domains, such as symmetric encryption, hashing and key derivation. This document will attempt to provide a reasonable analysis of ESAPI's Key Derivation Function (KDF), *CryptoHelper.computeDerivedKey*.

CryptoHelper.computeDerivedKey (hereafter simply *computeDerivedKey*) is a KDF which makes itself available via a public interface for crypto-savvy users. Internally, this static Java method is primarily used in ESAPI's crypto reference implementation class, *JavaEncryptor*. Most library users will indirectly use the KDF via *JavaEncryptor*. A listing of ESAPI 2.0 symmetric encryption routines can be found at *ESAPI 2.0 Symmetric Encryption User Guide*, http://owasp-esapi-java.googlecode.com/svn/trunk/documentation/esapi4java-core-2.0-symmetric-crypto-user-guide.html.

Reasonable is subjective and surely open for debate. In the past, implementers used NIST SP 800-56A, SP 800-56B, and more recently SP 800-108 (among other standard body recommendations) to help determine the fitness of a particular KDF algorithm.

Early KDFs did not differentiate between entropy extraction and expansion during key derivation. A number functions in the of 'Extraction-then-Expansion' family of functions have recently been proposed. Under this model, entropy extraction and key derivation are distinct processes. From RFC 5869, HMAC-based Extract-and-Expand Key Derivation Function (HKDF):

> HKDF follows the "extract-then-expand" paradigm, where the KDF logically consists of two modules. The first stage takes the input keying material and "extracts" from it a fixed-length pseudorandom key K. The second stage "expands" the key K into several additional pseudorandom keys (the output of the KDF).

Since ESAPI's implementation is not of the 'Extract-then-Expand' family, this document will attempt to analyze *computeDerivedKey* from a NIST SP 800-108 perspective. ESAPI's *computeDerivedKey* appears to be most similar in design to SP 800-108's **KDF in Counter Mode** described in Section 5.1.

Finally, the examination does not include ESAPI's security as a whole, security levels, caller's use of the function, or other tangential functionality.

2. Analysis

The digested Java version of the *computeDerivedKey* implementation is shown in Listing 1 below. Error checking has been omitted for clarity. The heart of the implementation is an HMAC which supplies at least the requested number of bits to the caller.

```
public static SecretKey computeDerivedKey(SecretKey keyDerivationKey,
    int keySize, String purpose)
{
      keySize = ( keySize + 7 ) / 8;    // Bits to octets

      byte[] derivedKey = new byte[ keySize ];
      byte[] inputBytes = purpose.getBytes("UTF-8");

      SecretKey sk = new SecretKeySpec(keyDerivationKey.getEncoded(), "HmacSHA1");

      Mac hmac = Mac.getInstance("HmacSHA1");
      hmac.init(sk);

      int totalCopied = 0;
      int destPos = 0;
      int len = 0;

      do {
            byte[] tmpKey = hmac.doFinal(inputBytes);

            if ( tmpKey.length >= keySize ) {
                  len = keySize;
            } else {
                  len = Math.min(tmpKey.length, keySize - totalCopied);
            }

            System.arraycopy(tmpKey, 0, derivedKey, destPos, len);

            inputBytes = tmpKey;
            totalCopied += tmpKey.length;
            destPos += len;

      } while( totalCopied < keySize );

      return new SecretKeySpec(derivedKey, keyDerivationKey.getAlgorithm());
}
```

Listing 1: Digested *computeDerivedKey*

In Listing 1, *keyDerivationKey* is a function parameter which is a shared secret. The *keyDerivationKey* is used as a key for the HMAC. *keySize* is specified in bits by the caller but immediately converted to an octet count (i.e., the number of 8 bit bytes) for internal use. *purpose* is also a parameter and indicates the intended use of the derived key – encryption or authenticity. Ultimately, *purpose* is used as a label to the HMAC.

The use of a HMAC is a solid design decision. According to Dodis, Gennaro, Håstad, Krawczyk, and Rabin in *Randomness Extraction and Key derivation Using the CBC, Cascade, and HMAC Modes*: “HMAC is the most widely used pseudorandom mode based on functions

such as MD5 or SHA". Dodis, *et al* then proceed to offer formal proofs for its suitability in key derivation.

In addition, a HMAC does not require cascade chaining (as in the use of block ciphers), which simplifies the implementation since an HMAC is, by design, a cascade of compression functions. Dodis, et al formally define the HMAC family in section 5 of their paper.

Using *keyDerivationKey* to key the HMAC is required by SP800-108 (cf., Section 4, Pseudorandom Function (PRF), symbol K_I). This is clearly performed during the execution of:

```
SecretKey sk = new SecretKeySpec(keyDerivationKey.getEncoded(), "HmacSHA1");
```

The do-while loop generates the actual bytes of the derived key which is returned to the caller. If the output size of the HMAC matches or exceeds the size of the parameter *keySize*, c*omputeDerivedKey* will perform a single iteration and fulfill the caller's request. If the caller requests a *keySize* larger than the HMAC's block size, iterations will be performed until enough output from the HMAC has been gathered.

In the case that not all bytes from the output of a HMAC invocation are required to fulfill a request, the leftmost bytes are taken from the output. This is consistent with traditional use of a truncated output.

Finally, non-overlapping segments are used to fulfill requests, so *computeDerivedKey* does not violate SP 800-108 (or other standards): "[keying material] is a binary string, such that any non-overlapping segments of the string with the required lengths can be used as symmetric cryptographic keys."

Upon examining the use of the HMAC in the do-while loop, a call is made to *doFinal* on the instance of the HMAC object to extract bytes:

```
tmpKey = hmac.doFinal(inputBytes);
```

tmpKey is a temporary array of bytes, which is copied into the array which is eventually returned to the caller. *inputBytes* is a binary encoding of the parameter *purpose*, which is the traditional label. Notably absent from the call are the iteration count, context, and other adornments which usually accompany an iteration of a PRF.

NIST's Special Publication 800-108 performs the following in the iterative loop of section 5.1:

```
n := number of blocks required to fulfill request
for i = 1 to n, do

    K(i) := PRF(K_I, [i]_2 || Label || 0x00 || Context || [L]_2)
    result(i) := result(i-1) || K(i)

end
```

where '||' is represents bit string concatenation.

In the above, $[i]_2$ is the big-endian binary representation of the iteration, and $[L]_2$ is the bits requested by the caller.

The above pseudo-code roughly translates to the following, where *derivedKey* is the leftmost *L* bits of *result*. Note that the **only** parameter which changes between invocations is the binary representation of *i*.

```
n := number of blocks required to fulfill request
hmac.init(sk)
for i = 1 to n, do

    K(i) := hmac.DoFinal([i]₂ || Label || 0x00 || Context || [L]₂)
    result(i) := result(i-1) || K(i)

end
```

In section 3.2 of SP 800-108, NIST uses {*X*} to indicate optional data (the emphasis is on the curly braces). Unfortunately, the iteration, null byte, context, and bit length are not optional.

Three of the missing fields are easily correctable and deserve no further study. However, *Context* appears to complicate matters. SP 800-108 defines the context as:

> A binary string containing the information related to the derived keying material. It may include identities of parties who are deriving and/or using the derived keying material and, optionally, a nonce known by the parties who derive the keys.

From Section 7.6 of SP 800-108 on Context Binding, *Context* appears to be an optional field since NIST chose the use of **SHOULD** versus **MUST**:

> Derived keying material should be bound to all relying entities and other information to identify the derived keying material. This is called context binding. In particular, the identity (or identifier, as the term is defined in [NIST SP 800-56A , sic] and [NIST SP 800-56B , sic]) of each entity that will access (meaning derive, hold, use, and/or distribute) any segment of the keying material should be included in the Context string input to the KDF, provided that this information is known by each entity who derives the keying material.

For completeness, SP 800-56A uses AlgorithmID (public), PartyUInfo (public), PartyVInfo (public), and optional public and private information for *Context*. See Appendix A of this document for the relevant partial text of SP 800-56A.

Turning to the ISO/IEC, the definition of a KDF1 from ISO 18033-2, Section 6.2.2.1 is shown below.I2OSP is *Integer to Octet String Primitive*, which is the ISO/IEC's equivalent to a NIST binary string.

> For an octet string x and a non-negative integer l, KDF1 (x, l) is defined to be the first l octets of
>
> ```
> Hash.eval(x ‖ I2OSP(0, 4)) ‖ ··· ‖ Hash.eval(x ‖ I2OSP(k-1, 4))
> ```
>
> where
>
> ```
> k = ceiling(l/Hash.len)
> ```

The ISO/IEC's KDF2 is a similar construction (refer to section 6.2.3.1): rather than iterating from 0 to k – 1, KDF2 iterates from 1 to k. Under both ISO/IEC's constructions, the iteration is used as input into the hash function, and context data is **not** specified.

PKCS #5 and RFC 2898 (i.e., PBKDF v2) do not appear to lend assistance since the initial secret is a low entropy password, and not a random number (generated from a cryptographically secure pseudorandom number generator) or shared secret (for example, ss = g^{xy} from a successful Diffie-Hellman exchange). It is not clear if the leap can be made from PBKDF to KDF. However, it is noted that even PBKDF v2 processes the iteration count in the calculation of the derived key.

SP 800-135 retrofits “Extract-then-Expand” functionality into existing protocols, such as IKE and TLS. Under the 135 Special Publication, context is already specified via the traditional design and use of a KDF, and retrofitting does not relieve the need for context required by the underlying protocol.

3. Recommendations

First, I would recommend a professional cryptographer audit the implementation for cross validation. In the absence of a seasoned professional, I would recommend partial, if not full, compliance with SP 800-108. Compliance with 108 appears to be relatively easy.

Based on NIST 800-108 and ISO/IEC KDF1 and KDF2, context data is not strictly required. However, NIST 800-56A and 800-56B and the IETF are more stringent. Observing past due diligence from NIST and the IETF, and in the spirit of “uniqueness of derivation per KDF instance invocation”, adding an additional optional parameter to *computeDerivedKey* would probably be a very good design decision. Caller's which could supply additional uniqueness would be allowed to do so; while callers which lacked the uniqueness would not be required to retrofit existing implementations for the enhanced interface.

ISO/IEC 18033, ANSI X9, and P1363 specify KDFs which might prove useful for context inspiration. At minimum, any number of the aforementioned standard bodies might provide concrete requirements for the context parameter, or guidance on the selection of operation parameters used as a context.

For those who wish to use the optional context data: if *computeDerivedKey* was used in a Diffie-Hellman exchange, the public keys of each party could be used as context. Distant memories seem to recall an ASN.1 OID indicating usage was also used for context in a KDF. Additional context candidates are public salts, nonces, initialization vectors, and public keys. Indeed, public identities are specifically mentioned in NIST's Special Publications (among other data).

In the case that *context* is not shared between two parties (i.e., it’s a local key used only by a database), consider using the context parameter as an opportunity to tie the database to the installation or server. An encoding of the database path and name would probably transform nicely into a relatively unique, distinguishing value. Tying the database to a unique server identifier (i.e., UUID or GUID) would help deter copy/paste attacks.

In addition to building a conforming KDF, OWASP also has the choice to use HKDF specified in RFC 5869 or upcoming SP 800-56C. For RFC 5869, the context is an optional parameter and represents current state of the art.

Finally, OWASP has the option to accept *computeDerivedKey* as-is, discard SP 800-108 and other standard recommendations, and do nothing. This option has the benefits of reduced complexity from an implementor's viewpoint; and interoperability for existing implementations in the field.

4. Acknowledgments

I would like to thank Dr. Brooke Stephens, Dr. David Wagner, and Kevin Wall for helpful comments.

Normative References

- NIST Special Publication 800-135, *Recommendation for Existing Application-Specific Key Derivation Functions*
- NIST Special Publication 800-108, *Recommendation for Key Derivation Using Pseudorandom Functions*
- NIST SP 800-56B, *Recommendation for Pair-Wise Key Establishment Schemes Using*
- *Integer Factorization Cryptography*
- NIST Special Publication 800-56A, *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*
- ISO/IEC 18033-2 , *Encryption Algorithms – Part 2: Asymmetric Ciphers*
- RFC 5869, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*
- Krawczyk, *Cryptographic Extraction and Key Derivation: The HKDF Scheme*
- Dodis, Gennaro, Håstad, Krawczyk, and Rabin, *Randomness Extraction and Key derivation Using the CBC, Cascade, and HMAC Modes*
- Chevassut, *Key Derivation and Randomness Extraction*

Appendix A - Excerpt from NIST SP 800-56A

3 OtherInfo: A bit string equal to the following concatenation:

AlgorithmID || PartyUInfo || PartyVInfo {|| SuppPubInfo }{|| SuppPrivInfo }

where the subfields are defined as follows:

3.1 AlgorithmID: A bit string that indicates how the derived keying material will be parsed and for which algorithm(s) the derived secret keying material will be used. For example, AlgorithmID might indicate that bits 1-80 are to be used as an 80-bit HMAC key and that bits 81-208 are to be used as a 128-bit AES key.

3.2 PartyUInfo: A bit string containing public information that is required by the application using this KDF to be contributed by party U to the key derivation process. At a minimum, PartyUInfo shall include IDU, the identifier of party U. See the notes below.

3.3 PartyVInfo: A bit string containing public information that is required by the application using this KDF to be contributed by party V to the key derivation process. At a minimum, PartyVInfo shall include IDV, the identifier of party V. See the notes below.

3.4 (Optional) SuppPubInfo: A bit string containing additional, mutually-known public information.

3.5 (Optional) SuppPrivInfo: A bit string containing additional, mutually-known private information (for example, a shared secret symmetric key that has been communicated through a separate channel).

Each of the three subfields AlgorithmID, PartyUInfo, and PartyVInfo shall be the concatenation of an application-specific, fixed-length sequence of substrings of information. Each substring representing a separate unit of information shall have one of these two formats: Either it is a fixed-length bit string, or it has the form Datalen || Data, where Data is a variable-length string of zero or more bytes, and Datalen is a fixed-length, big-endian counter that indicates the length (in bytes) of Data. (In this variable-length format, a null string of data shall be represented by using Datalen to indicate that Data has length zero.)

An application using this KDF shall specify the ordering and number of the separate information substrings used in each of the subfields AlgorithmID, PartyUInfo, and PartyVInfo, and shall also specify which of the two formats (fixed-length or variable-length) is used for each substring. The application shall specify the lengths for all fixed-length quantities, including the Datalen counters.

...