10.1.2Hash Function DRBG Using HMAC with Any Approved Hash (HMAC_DRBG)

10.1.2.1 Discussion

This section discusses a new DRBG based on using any approved hash function in the HMAC construction for making a keyed hash function. Any application that has access to an approved hash function can implement HMAC, though dedicated implementations of HMAC will be considerably more efficient.

The maximum security level supported by HMAC_DRBG is the output size of the underlying hash function, in bits. Although some applications of HMAC use only a subset of the output bits to save bandwidth, the HMAC_DRBG is defined using all the bits of the HMAC output. (This costs nothing, as any HMAC implementation *computes* all the output bits, even when it never emits some of them.)

10.1.2.2 Interaction with HMAC_DRBG

10.1.2.2.1 Instantiating HMAC_DRBG (...)

Prior to the first request for pseudorandom bits, the **HMAC_DRBG (...) shall** be instantiated using the following call:

status = **Instantiate_Hash_DRBG** (*usage_class, requested_strength, prediction_resistance_flag, personalization_string*),

as described in Section 9.6.1.

10.1.2.2.2 Reseeding a HMAC_DRBG (...) Instantiation

When a **HMAC_DRBG (...)** instantiation requires reseeding, the DRBG **shall** be reseeded using the following call:

status = **Reseed_HMAC_DRBG_Instantiation** (*usage_class*)

as described in Section 9.7.2.

10.1.2.2.3 Generating Pseudorandom Bits Using HMAC_DRBG (...)

An application may request the generation of pseudorandom bits by **HMAC_DRBG (...)** using the following call:

(*status, pseudorandom_bits*) = **HMAC_DRBG** (*usage_class, requested_no_of_bits, requested_strength, additional_input_flag, prediction_resistance_flag*)

as discussed in Section 9.8.2.

10.1.2.3 Specifications

10.1.2.3.1 General

The instantiation of **HMAC_DRBG (...)** consists of obtaining a *seed* with the appropriate amount of entropy, which is used to define the initial *state* of the DRBG. The *state* consists of:

1. The *usage_class* for the DRBG instantiation (if the DRBG is used for multiple *usage_classes*, requiring multiple instantiations, then the *usage_class* parameter **shall** be present, and the implementation **shall** accommodate multiple *states* simultaneously; if the DRBG will be used for only one *usage_class*, then the

usage_class parameter **may** be omitted).

2. The value X, which is updated each time another N bits of output are produced (where N is the number of output bits in the underlying hash).
3. The value K, which is updated at least once each time the DRBG generates pseudorandom bits.
4. The size of the hash function output, N.
5. A *prediction_resistance_flag* that indicates whether or not prediction resistance is required by the DRBG. Note that if the DRBG is implemented to always or never support prediction resistance, then this parameter is not required in the state.
6. (Optional) The hash of the most recently used entropy source output, to be compared with the next entropy source output used in reseeding. .

The variables used in the description of **HMAC_DRBG (...)** are:

additional_input	Additional input.
Get_entropy (128,160, 512)	A function that acquires a string of bits from an entropy source. 128 indicates the minimum amount of entropy to be provided in the returned bits; 160 indicates the minimum number of bits to be returned; 512 indicates the maximum number of bits to be returned. See Section 9.6.2.
HMAC(K,X)	Apply the HMAC keyed hash function with key K to message input X.
old_transformed_seed	The *transformed_seed* from the previous seeding of the instantiation.
prediction_resistance_supported	A flag indicating whether or not prediction resistance is to be supported by the DRBG. 1 = yes; 0 = no.
prediction_resistance_flag	A flag indicating whether this HMAC_DRBG call will require prediction resistance.
pseudorandom_bits	The string of pseudorandom bits that are generated during a single "call" to the **HMAC_DRBG (...)** process.
requested_no_of_bits	The number of bits requested from the DRBG.
requested_strength	The requested security strength for the pseudorandom bits obtained from the DRBG.
N	The number of bits in the hash function output.
Seed material	The seed material used to initialize or reseed this instance of the **HMAC_DRBG(...)**.
state	The state of **HMAC_DRBG (...)** that is carried between calls to the DRBG. In the following specifications, the entire state is defined as {*usage_class,* N, X, K, *prediction_resistance_flag, transformed_seed*}. A particular element of the *state* is specified as *state.element*, e.g., *state.K.*
status	The *status* returned from a function call, where *status* = "Success" or an indication of a failure. Failure messages are:

	1. Invalid *requested_strength*. 2. Failure indication returned by the entropy source. 3. State not available for the indicated *usage_class*. 4. Entropy source failure. 5. Invalid *additional_input_flag* value. 6. Failure from request for *additional_input*. 7. *additional_input* too large. 8. Too many bits of output requested.
t	The initial value of the hash function. See Annex E.
temp	A temporary value.
transformed_seed	A one-way transformation of the *seed* for the **HMAC_DRBG(...)** instance.
usage_class	The purpose(s) of a DRBG instance.

10.1.2.3.2 Instantiation of HMAC_DRBG(...)

The following process or its equivalent **shall** be used to initially instantiate the **HMAC_DRBG (...)** process in Section 10.1.2.3.4:

Instantiate_HMAC_DRBG (...):

Input: integer (*usage_class*, *requested_strength*, *prediction_resistance_supported*, *personalization_string*)

Output: string *status*.

Process:

1. If requested_strength>N, then Return("Invalid requested_strength)
2. If prediction_resistance_supported=True, then verify that the entropy source can support prediction resistance. If not, Return("Cannot support prediction resistance.")
3. (status,seed) = Get_entropy(Max(N,128),N,2^{32})
4. If (*status* = "Failure"), then **Return** ("Failure indication returned by the entropy source").
5. transformed_seed = hash(seed)
6. K = N/8 bytes of binary zeros, 0x00 00 00 ... 00
7. X = N/8 bytes of binary ones, 0x01 01 01 ... 01
8. K = HMAC(K,X || 0x00 || seed || personalization_string)
9. X = HMAC(K,X)
10. K = HMAC(K,X || 0x01 || seed || personalization_string)
11. X = HMAC(K,X)
12. state = {usage_class, prediction_resistance_supported,K,X,transformed_seed}
13. **Return** ("Success").

10.1.2.3.2.1 Additional Comments and Caveats

1. Note that if an implementation maintains multiple DRBG instances, it must also provide separate storage and *usage_class* for each. [[We should just call that a *drbg_handle*, instead.--JMK]]
2. If an implementation does not need the *usage_class* as a calling parameter (i.e., the implementation does not handle multiple usage classes), then the *usage_class* calling parameter may be omitted, , and the *usage_class* indication in the *state* (see step 10)

must be omitted[1]. (Note that *usage_class* never affects the operation of the algorithm; it is an implementation convenience only.)

3. If an implementation will only instantiate for one[[]] N bits of security, then the *requested_strength* parameter and step 1 can be omitted.
4. If an implementation does not need the *prediction resistance supported* as a calling parameter (i.e., the **HMAC_DRBG (....)** routine in Section 10.1.2.3.4 either always or never acquires new entropy in step 9), then the *prediction_resistance_supported* in the Instantiate call and in the *state* (see step 10) must be omitted.
5. If an implementation will never be reseeded using the process specified in Section 10.1.2.3.3, then step 6 may be omitted, as well as the *transformed seed* in the *state* (see step 10). This does not preclude using the **Instantiate_HMAC_DRBG (...)** process to create a new instantiation.
6. If an implementation will never use the personalization string, it may be replaced in this pseudocode with an empty string ("").
7. Note that regardless of the requested security level, the HMAC_DRBG is always instantiated to support an N-bit security level. Thus, when the HMAC_DRBG based on SHA256 is instantiated at the 80-bit security level, the DRBG requests 256 bits of entropy from the entropy source. [[Is this okay? It simplifies things a great deal!]]

10.1.2.3.3 Reseeding a HMAC_DRBG(...) Instantiation

The following or an equivalent process **shall** be used to explicitly reseed the **HMAC_DRBG (...)** process:

[[*Some questions:*

1. *Should Reseed() verify that the entropy source can support an independent reseed? Is that the same as being able to support prediction resistance? This ought to be part of the interface with a seed source—the DRBG needs to be able to ask it if it can really provide prediction resistance, whether it's a seed string or an entropy source or an RBG with or without an entropy source, etc. We need to think about this and discuss it at the next meeting.*
2. *Should Reseed() allow some application-level input, comparable to the optional input and personalization string?*
3. *Should optional inputs like the personalization string include some way of distinguishing whether or not they exist? Should that be an explicit flag, or just a NULL sort of indicator?*
4. *In step 1, below, we talk about checking to see if a given usage_class is available. But where did the application* get *the usage_class? Either there's an additional call like DRBG_Setup() that returns it, or Instantiate should return the usage_class to be used from now on, right?*

--JMK]]

Reseed_HMAC_DRBG_Instantiation (...):

Input: integer (*usage_class*).
Output: string *status*.
Process:

1. If a *state* is not available for an indicated *usage_class*, then **Return** ("State not

[1]Actually, it's not so clear that the usage_class belongs in the state at all. The state is indexed by the usage_class, just like a file handle or a process ID.

available for the indicated *usage_class*").

2. Get the appropriate *state* values for the indicated *usage_class*, e.g., *K* = *state.K*, *N* = *state.N*, *transformed_seed* = *state.transformed_seed*.
3. Perform the following steps:
   3.1. (status,seed) = Get_entropy(Max(N,128),N,2^{32})
   3.2. If (*status* = "Failure"), then **Return** ("Failure indication returned by the entropy source").
   3.4 If transformed_seed==hash(seed) then Return("Entropy source failure")
   Else state.transformed_seed = hash(seed)
4. K = HMAC(K,X||0x00||seed)
5. X = HMAC(K,X)
6. K = HMAC(K,X||0x01||seed)
7. X = HMAC(K,X)
9. state.X = X
10. state.K = K
11. **Return ("Success").**

If an implementation does not need the *usage_class* as a calling parameter (i.e., the implementation does not handle multiple usage classes), then the *usage_class* calling parameter and step 1 can be omitted and step 2 acquires the only *state* that is specified.

10.1.2.3.4 Generating Pseudorandom Bits Using HMAC_DRBG(...)

The following process or an equivalent **shall** be used to generate pseudorandom bits:

HMAC_DRBG(...):

Input: integer (*usage_class*, *requested_no_of_bits*, *requested_strength*, *additional_input_present*, *additional_input*, *prediction_resistance_flag*).

Output: string (*status*, *pseudorandom_bits*).

Process:

2. If a *state* for the indicated *usage_class* is not available, then **Return** ("State not available for the indicated *usage_class*", Null).
1. If ((*requested_strength* >N), then **Return** ("Invalid *requested_strength*", Null).
4. If requested_no_of_bits>2^{35} then Return("Too many bits requested.",Null)
5. Get the appropriate *state* values in accordance with the indicated *usage_class*, e.g., *K* = *state.K*, *t* = *state.t*, etc.
7. If prediction_resistance_flag then
   7.1 If not state.prediction_resistance, then Return("Prediction resistance not supported.")
   Else seed = Get_entropy(Max(N,128),N,2^{32})
   Else seed = ""
8. If additional_input_present then seed = seed || additional_input
9. If (seed <> "") then:
   9.1 K = HMAC(K,X||0x00||seed)
   9.2 X = HMAC(K,X)
   9.3 K = HMAC(K,X||0x01||seed)
   9.4 X = HMAC(K,X)

10. temp = “”
11. While (len(temp)<requested_no_of_bits) do:
    11.1 X = HMAC(K,X)
    11.2 temp = temp || X
12. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) of (*temp*).
13. if (seed<>””) then
    13.1 K = HMAC(K,X||0x00||seed)
    13.2 X = HMAC(K,X)
    13.3 K = HMAC(K,X||0x01||seed)
    13.4 X = HMAC(K,X)
14. else (if seed==””) then
    14.1 K = HMAC(K,X||0x00)
    14.2 X = HMAC(K,X)
15. state.X = X
16. state.K = K
17. Return(“Success”,*pseudorandom_bits*)

Additional Notes and Caveats:

1. If an implementation does not need the *usage_class* as a calling parameter (i.e., the implementation does not handle multiple usage classes), then the usage_class parameter and step 3 can be omitted, and step 4 acquires the only state available.
2. If an implementation will never require more than N bits of security, then the *requested_strength* parameter and step 1 can be omitted.
3. If an implementation will never include additional input, then the *additional_input_present* in the calling parameter may be omitted, the *additional_input* term may be removed, and step 8, above, may be omitted.
4. If an implementation does not use the *prediction_resistance_flag* in the *state* (see Section 10.1.2.3.2), then the *prediction_resistance_flag* is not acquired, and the reference to the *prediction_resistance_flag* is omitted.
5. If neither additional input nor prediction resistance are ever used, then the temporary variable *seed* can simply be set to “”, and steps 9 and 13 can be entirely omitted. (Step 14 is then always executed, as *seed* is always “”.)

10.1.2.4 Generator Strength and Attributes

The HMAC_DRBG is designed to meet the security requirements for all DRBG algorithms. Specifically:

1. When the value of *K* is not known to or guessable by an attacker, the outputs will be indistinguishable from random to that attacker. This property is based on the assumption of pseudorandomness for the HMAC construction with an approved hash function.
2. Compromise of the current state has no effect on the security of previous outputs. That is, the DRBG algorithm provides *backtracking resistance.*
3. When the *prediction_resistance_requested* flag is set in a DRBG call, or when the optional input is unguessable to an attacker, then the DRBG state recovers from a compromised state, and the security of future outputs is not affected by the previous compromise. That is, the DRBG is capable of supporting *prediction resistance* when it gets sufficient entropy to do so.

4. On instantiation, if the string formed by concatenating the personalization string and the entropy source output is unguessable to the attacker, then the DRBG is instantiated to a secure state.

10.1.2.5 Reseeding and Optional Input

Instantiation, reseeding, and generating pseudorandom bits with prediction resistance are all equivalent in security terms. Indeed, the mechanism for instantiating the HMAC_DRBG is nothing more than setting K and X to constant bitstrings, and then doing operations equivalent to generating N bits of DRBG output with prediction resistance, and reseeding the DRBG is nothing more than generating N bits of DRBG output with prediction resistance, but discarding the N bits of output.

When pseudorandom bits are called with optional input, the HMAC_DRBG guarantees that if the optional input has at least N bits of entropy, then the DRBG will achieve prediction resistance—meaning that even if the DRBG state was known to an attacker before the pseudorandom bits were generated, neither the pseudorandom bits nor the DRBG state after generating the bits will be known to the attacker.

If an application has a slow source of entropy, such as keystroke timings, it **should** accumulate the entropy until it estimates that it has N bits, and then feed all the entropy into the DRBG as a single optional input. This will permit the DRBG to recover from any compromise.