Two New DRBGs Based on Any Approved Hash Function
John Kelsey, NIST, January 2004

1 Introduction

X9F1 needs a secure and trusted determinisitic random bit generator based on a hash function. Our previous candidate had some security problems. In this paper, I describe two very closely related new DRBGs, one based on any hash function whose construction includes a compression function with both message inputs and chaining value inputs, and the other based on any hash function that can be turned into a PRF by the use of the HMAC construction. The overall design, and nearly all of the analysis, is identical between the two.

1.1 The Basic DRBG Algorithm Interface

A DRBG algorithm needs to be able to do two things:

a. Generate outputs that are indistinguishable from ideal random outputs to someone who doesn't know its DRBG's working state.

b. Update or initialize its working state in a secure way, potentially in the face of an attacker who is choosing part or all of the input seed material to try to prevent the DRBG getting to a secure state.

These are reflected in three public methods that any DRBG algorithm must support: Generate(bytes,optionalInput), Initialize(seed), and Reseed(seed).

1.2 Basic Construction

Both DRBGs are constructed in the same way. All security resides in the Generate() method; the Initialize() and Reseed() methods all that method.

In both cases, the hash function is used to construct a Pseudo-Random Function family, or PRF. A PRF is essentially a keyed function for which, when the key is unknown, each new input yields an output indistinguishable from an ideally-generated random number; a natural way to represent this is F_K(X), where K is the key, and X is the input.

To generate a new random bit string, the generate methods in both DRBGs do the same basic sequence of things:

a. If there is optional input, use it to derive a new K and X.

b. Generate output by computing:
```
temp = ""
while len(temp)< requested number of bytes:
  X = F_K(X)
  temp = temp || X
```

c. Derive a new K and X to prevent backtracking attacks.

To reseed and initialize, both DRBGs call the Generate() method with the seed string as the optional input.

For the rest of this document, N is the hash output size, M is the compression function message input size, and Q is the compression function output size. For example, with SHA1, Q=N=160 and M=512, while with SHA384, N=384, Q=512, and M=1024.

1.3 HMAC_DRBG Overview

The HMAC_DRBG bases its security on the security properties of both the HMAC PRF construction, and the properties of the underlying hash function. Recall that HMAC takes a fixed-length key, typically of N bits, where N is the number of bits in the hash function's output, and can handle the same enormous range of data input sizes as the corresponding hash function.

The HMAC_DRBG has two pieces of working state: K and X. Each is N bits wide.

The three methods are defined as follows:

```
def Generate(bytes,userInput):

  # Update the state in a secure way with user input.
  if userInput exists:
    K = hmac(K,X || 0x00 || userInput)
    X = hmac(K,X)

  # Generate the bits.
  tmp = ""
  while len(tmp)<bytes:
    X = hmac(K,X)
    tmp = tmp || X

  # Get to a new key and state to prevent backtracking attacks.
```

```
  if userInput exists:
    K = hmac(K,X || 0x01 || userInput)
    X = hmac(K,X)
  else:
    K = hmac(K,X || 0x01)
    X = hmac(K,X)

  # Finally, return the bits.
  return tmp[:bytes]

# Note: The result from the Generate() call can be saved as a check
# value, to ensure that the DRBG is not set to the same working state
# twice.
def Reseed(seed):
  Generate(N,seed)

def Initialize(seed):
  X = 0x00 0x00 ... 0x00
  K = 0x01 0x01 ... 0x01
  Reseed(seed)
```

1.4 KHF_DRBG Overview

The KHF_DRBG also bases its security on both a PRF construction and the hash function, but does so in a slightly different way. This trades off a certain elegance of design for a factor of two or so improvement in performance.

The PRF construction used here is

F_K(X) = Compress(K_0,K_1 xor (X||0x00 0x00 ... 0x00))

where K_0 is Q bits (the number of bits in the compression function's output), and K_1 is M bits wide, the same number of bits as the compression function's message input block.

The KHF_DRBG algorithm's working state includes three variables: K_0, and X, both of Q bits, and K_1, of M bits. Note that because KHF_DRBG uses the underlying compression function of the hash, the PRF has a Q-bit output, even when Q>N. For example, in SHA384, the PRF has a 512-bit output, even though the hash function as a whole has a 384-bit output.

There are two internal functions used in the KHF_DRBG, called _F() and _Renew().

```
def _F(x):
```

```
  return Compress(K_0,K_1 xor (X || 0x00 0x00 ... 0x00))

def _Renew(seed):
  T = ""
  # Divide the bit counts by 8 to get byte counts.
  while len(T)< (Q+M)/8:
    X = _F(X)
    T = T || _F(hash(X||seed))
  K_0 = T[:Q/8]    # First Q bits go to K_0.
  K_1 = T[Q/8:M/8] # Next M bits go to K_1.
  X = _F(X)
  return
```

These are then used as follows:

```
def Generate(bytes,userInput):

  # If we have user input, use it to get a whole new key.  This
  # provides prediction resistance.
  if userInput exists:
    _Renew(userInput)

  # Generate the output bytes.
  temp = ""
  while len(temp)<bytes:
    X = _F(X)
    temp = temp || X

  # Go to a new key to prevent backtracking attacks.
  if userInput exists:
    _Renew(userInput)
  else:
    _Renew("")

  return temp[:bytes]

def Reseed(seed):
  Generate(Q,seed)

def Initialize(seed):
  X = 0x00 0x00 ... 0x00
  K_0 = 0x01 0x01 ... 0x01
  K_1 = 0x02 0x02 ... 0x02
  Reseed(seed)
```

2 Analysis

2.1 Analysis of the Generic Construction

Both of these constructions are built on the same structure: All security-relevant operations take place in the Generate() functions, and those functions have the same basic outline:

F_K(X) is assumed to be a PRF. For the HMAC_DRBG, K and X are each N bits wide. For KHF_DRBG, K is Q+M bits wide, and X is Q bits wide. Generate works as follows for each:

a. If we have user input, use that to derive a new K and X from the current K and X, and also from the user input. This is done in a way that will provide prediction resistance if the user input has enough entropy.

b. Run the PRF in output feedback mode until we've generated enough output bits. That is:

```
tmp = ""
while len(tmp)<bytes:
  X = F_K(X)
  tmp = tmp || X
```

c. Derive a new key and chaining value, from whatever external input is available, or just from the internal state if no such external input is available.

2.1.1 The initial use of userInput

If there is optional input from the user or seed data required to achieve prediction resistance, then the DRBG must immediately generate a new K and X. In both DRBGs, this is done in a way that should guarantee that the resulting K is no easier to guess than the harder of the previous K or the input.

2.1.1.1 HMAC_DRBG

For HMAC_DRBG, this is very simple:

```
K = HMAC(K,X||0x00||additionalInput)
```

2.1.1.1.1 Unknown K, known/chosen input.

We assume that HMAC is a PRF. So, an attacker who cannot guess the starting K can't distinguish the resulting K from a random N-bit string, unless he's seen the result of this specific key and input

string being fed into HMAC before. However, this cannot happen: the only HMAC outputs that are released are derived using only X as the input; this input string is a different length, and so cannot be the same as any of those. So, an attacker cannot have seen this output before, and so can't distinguish it from a random N-bit string. (This could be formalized.) This is true even if the attacker chooses the inputs maliciously.

Further, assuming the key never cycles, we derive a new key exactly once using any key, so there are no opportunities short of key collisions for an attacker to cause or observe the results of two different derivations of a new key, from the same starting key. As K is an N-bit output which is assumed to be indistinguishable from random, a sequence of 2^{64} of these K values have only a 2^{127-N} probability of having a collision. Note that even if K repeats, unless X does, too, an attacker will still never have a pair of identical inputs to HMAC. If both K and X cycle at the same time, the DRBG will begin repeating a sequence of previously-generated bits.

2.1.1.1.2 Known K, unknown input

When the key is known, we cannot assume anything about HMAC being a PRF. Instead, we must consider it as an application of the hash function. We then must assume that the hash function does a good job of distilling entropy. In the strongest sense, we need to assume that computing

hash(K xor opad || hash(K xor ipad || X || 0x00 || seed))

gives us an output that is indistinguishable from a random HMAC key, when an attacker doesn't know the seed. (This is inevitable, and we end up making the same assumptions in the hash-based DF, and in most fielded implementations of DRBGs. However, note that this is much weaker than the PRF assumptions described before, because the seed is not under the attacker's control, and because the attacker is assumed not to get large numbers of queries.)

The practical requirement is a bit weaker: we need the above hash function to give us a suitable HMAC key that can't be guessed with substantially less work than the N-bit security level would imply, with seed material as input that an attacker doesn't know, and didn't generate or choose all of.

2.1.1.2 KHF_DRBG

K really consists of Q+M bits. The new K is generated like this:

```
temp = ""
while len(temp)<(Q+M)/8:
  X = F_K(X)
  temp = temp || hash(X || seed )

K_0 = first N bits of temp
K_1 = next M bits of temp
```

For concreteness, when the underlying hash function is SHA1, the hash function is computed five times; when it is SHA256, the hash function is computed only three times.

We again have to resort to some additional assumptions about the hash function, though plausible ones:

a. When F_K(X) is computed with a known K and unknown X, the result is as hard to guess as X, and in fact, an attacker who can't guess X can't distinguish F_K(X) from a random number.

b. When hash(X||seed) is computed for many different unknown X values, an attacker choosing seed cannot force collisions for members of the sequence of these values, even when he's able to do 2^N work in choosing his seed.

c. When hash(seed) is computed for an unguessable seed which is not chosen by attacker, the output is unguessable as well.

In some fundamental sense, these end up being minimal assumptions on the hash function for securely initializing any DRBG based on them; if any N bit block of the input is unknown, then the output must also be unknown. Also, the real requirements are much weaker than those stated here; the attacker mustn't be able to find an attack based on any deviation from random behavior he can cause in the keys to the PRF.

2.1.1.2.1 Unknown K/known or chosen seed

In this case, the attacker knows or chooses the seed, but doesn't know K. Since he doesn't know the first N bits of each hash input, assumption (b) says he can't force the hash(X||seed) values to collide. This means that we get distinct inputs for each call to our PRF, and so we get new key values indistinguishable from random. (This can be formalized.)

2.1.1.2.2 Known K/unknown seed

In this case, we must rely on (a) and (c). Specifically, (c) tells us that when the seed is unknown, the input to F_K() will be unknown. (a) then tells us that the result will be indistinguishable from a random number.

2.1.2 Output generation

This is very simple: Let F_K(X) be a PRF, where an attacker gets A online queries, B work, and C advantage. Now, we have an output sequence generated as:

X[i] = F_K(X[i-1])

So, an attacker gets X[0,1,2,...,R-1]. Suppose he can distinguish this sequence from a sequence of random bits. Then one of two things has happened:

a. X[i]==X[j] for some i<>j.
b. The attacker has an algorithm that distinguishes F_K(X) from a random function with R queries.

The reason we're confident (a) doesn't happen is because the X[i] are N bits wide, where N>= 160, and these outputs are indistinguishable from random N-bit blocks. So, the probability that a pair of X[i] collide is quite low. This is ensured by forbidding more than 2^{32} output bytes on any one call, which keeps the probability of a collision down to less than 2^{-96}.

Let A(X[0,1,...,R-1]) be the algorithm to distinguish this sequence of X values from an ideal random sequence. The attacker simply chooses his own random value for X[0], and computes X[1,2,...,R-1] by submitting R-1 queries to F_K(). Once he has the whole sequence, he calls A() and returns the result. If A() distinguishes the output-feedback generated sequence from random, then this algorithm distinguishes F_K(X) from a random function.

What does this mean? If you can distinguish the OFB sequence from random, then you can violate the PRF assumption. But many ways of violating the PRF construction won't actually let you distinguish the OFB sequence from random. (This is why I used an output-feedback construction here, rather than a counter-mode construction; we aren't sending a sequence of closely-related inputs into our PRF this way.)

Basically, an attack on the output-feedback construction is going to have to be based upon either a known-input attack on the PRF, or upon

some special property of iterating the PRF that leads to bad results.

The only piece missing here is the question of whether or not these two hash constructions give us PRFs. That's covered below.

2.1.3 Regenerating the key again

Above, I discussed why regenerating the key with a seed is legitimate. How about without one?

2.1.3.1 HMAC_DRBG

For HMAC_DRBG, this translates to

```
K = HMAC(K,X||0x01)
X = HMAC(K,X)
```

Again, if HMAC is assumed to be a PRF, and X hasn't cycled, then the new K will be indistinguishable from an N-bit random string.

2.1.3.2 KHF_DRBG

For KHF_DRBG, this translates to

```
T = ""
while len(T)<(N+M)/8:
  X = F_K(X)
  T = T ||F_K(hash(X))
K0 = leftmost N bits
K1 = next M bits
```

If F_K is a PRF, then so long as the X don't repeat, all these new key bits are indistinguishable from random. As discussed above, the X won't repeat in practice. (If they do, then there is almost certainly a major problem with the PRF assumption.) And if X doesn't repeat, then there is a negligible probability that hash(X) will repeat.

2.1.3.3 General Concerns

2.1.3.3.1 Will K Cycle?

For HMAC_DRBG, K is only N bits wide, so this could happen in principle. For 2^{64} Generate requests, we will have a 2^{-33} probability of having a collision in K for two of the keys used in the Generate routines with SHA1, which is our worst case. However, so long as the X values don't also cycle, this will not lead to any

distinguishable change in the DRBG's output.

For KHF, K is so large that cycling isn't a problem.

2.1.3.3.2 Backtracking

Renewing K after each output sequene is generated prevents backtracking attacks. An attacker with a previously-used key and the first N bit block of an output sequence can predict the rest of the output flawlessly. To prevent this, the key is updated in a random way at the end of each Generate() calculation. The randomness of these outputs are based on the same PRF assumptions as the randomness of the Generate() outputs.

2.1.3.3.3 Entropy Loss

For the HMAC_DRBG, K and X each have at least 160 bits. Consider a usage pattern as follows:

2^{32} Generate requests, each for 2^{32} outputs. (This is a little more than would be allowed, but it lets us set a clean upper bound.) This means that during each Generate() operation, we cycle X = F_K(X) 2^{32} times, which will leave X in a set of about 2^{129} possible values. (The specific set will be dependent on K.) So, after a Generate() operation, we are in one of about 2^{289} states. We then generate a new K, using the 289 bits, possibly with no additional entropy coming in; assuming that the PRF works as expected, this should have 160 bits of entropy, but the total will still have only 289 bits; given K, there are only 2^{129} states X could be in. We iterate this again, and I think this time we are more stable--we basically go to 2^{128} states or so that X could be in. (Am I missing something; this *seems* right.)

I think after 2^{32} such requests, we will have dropped down to something like 2^{257} possible (K,X) states. I don't see how this can cause any problems for the security of the DRBG, but I'd really like someone else to look over my numbers, here!

For the KHF_DRBG, the total state is always at least 676 bits, and I am very confident we won't spiral down to a too-small number of states.

2.1.3.3.4 Precomputation Attacks

A basic attack on any DRBG involves precomputing as many states as you can afford, and then waiting to see if the DRBG happens to land in one of them. For the SHA1_DRBG with seedlen=160, this allows an attacker

to compute 2^{128} states, observe 2^{33} outputs, and recover the DRBG internal state with reasonably high probability.

For both these DRBGs, the keys make that class of attack impossible. Specifically, the DRBGs claim an N-bit security level, but have at least 2N bits of state. Even an attacker who can precompute 2^N states has a negligible chance of seeing an output corresponding to one of those states, for either DRBG.

2.1.4 Reseed and Initialize

Both of these routines, for both DRBGs, transparently depend on the Generate() functions. In both cases, we generate a 20-byte output, which can be stored for comparison with later initializations or reseedings.

2.2 Is HMAC a PRF?

HMAC is widely assumed to be a PRF. Attacking it in the model of an attacker with huge numbers of F_K() queries to distinguish it from random appears very difficult, but this doesn't prove that it is a PRF. (This is the sort of assumption that can never be more than an assumption, unless someone develops an attack, and shows that it was a mistaken assumption.)

The structure of HMAC as used in output generation is like this:

F_K(X) = Compress(K0, Compress(K1, X||padding))

Now, the output we see is the result of this outer Compress() function. And that function always starts with the same chaining value (for the same key), and gets a random-looking N-bit input in its message input block. Note that many of those input bits are fixed padding bits; for SHA1, that means that 352/512 message input bits are fixed and known to the attacker. However, this doesn't seem to lead to any attack. In an adaptive chosen input attack, the goal would be to find some property of these outputs from the outer Compress() function that deviated from random. I don't see any useful way to proceed in attacking this as a PRF.

2.3 Is My F_K(X) a PRF?

My PRF is based on a single Compress() call:

F_K(X) = Compress(K0,K1 xor (X || 0x000000...00))

This has the advantage that every single input bit for the compression function is unknown. It has the disadvantage that the attacker knows the actual XOR differences of the message inputs, and that only N bits in the message input to the compression function vary from function call to function call.

2.4 Summary: Do We Have a PRF and Do We Care?

If there is a way to distinguish the compression function from a PRF based on having a large number of known message input bits, then HMAC will be in trouble. If there is a way to distinguish it from a PRF based on having a large number of fixed message input bits, and a small N-bit XOR difference in the message input be the only thing that varies from call to call, then KHF is in trouble.

In reality, both these scenarios seem unlikely.

A real-world attack on these DRBGs will not just need to violate the PRF and other assumptions on these F_K() constructions, it will need to distinguish relatively short runs of output generated in output feedback mode from random in some useful way. Generate() limits its calls to 2^{32} bytes of output.

To relate this to existing schemes, note that the SHA1_DRBG uses the SHA1 compression function with the chaining input fixed and known to everyone, and the message input potentially held constant in all but 160 bits. Those 160 bits vary in a way that's perfectly understood by the attacker who sees the outputs from the SHA1_DRBG. This looks very similar to the situation in the KHF_DRBG, except the attacker here knows no input bits to the compression function. Intuitively, differential-type attacks are something any hash function designer must consider (to ensure there are no nonzero input differences that lead to zero output differences with high probability). Attacks based on knowing input differences seem unlikely to get very far, especially when none of the raw input bits at all are known.

My original design didn't include any key material in the message input block of the compression function. That seemed too risky to me (you get to bypass a good fraction of the actual work done by all the SHA hashes). Known message bits mixed with unknown ones seems much harder to attack in the SHA-type hashes, though with the RIPE-MD family of hashes, specific known message bits mean specific known words fed into the steps of the compression function; this seems like a much bigger opening for an attack.

3 Selecting a DRBG

Based on my analysis, I believe both of these DRGBs meet their security targets. However, I'm the designer; that should be taken with a large grain of salt until other people have done some independent analysis.

Should we make both approved DRBGs? I think so, because it allows people to trade off between something with a good pedigree (HMAC), and something that's twice as fast, but new.

3.1 Why I Like HMAC_DRBG

HMAC_DRBG has four big advantages:

a. It is a very elegant and simple design. It's easy to see that all the security resides in HMAC, which itself is a pretty easy-to-understand design. The only cryptographic function we ever call is HMAC.

b. The security assumptions for generating new key material from seeds and such are much more straightforward than in the KHF_DRBG.

c. Using HMAC needs no explanation; the public crypto community is (in my opinion) a little *too* trusting of HMAC. But it will certainly not be hard to justify. Nobody will scratch their head and say "Why the heck did they use *that* oddball construction?" when we propose something with HMAC as the PRF.

d. It can be coded up from standard HMAC or SHA libraries with a minimum of effort, and is built on top of components that have already been validated.

3.2 Why I Like KHF_DRBG

There are three advantages I can see to KHF_DRBG over HMAC_DRBG:

a. It's twice as fast. KHF_DRBG produces output bits as fast as a hash-based scheme reasonably can--one compression function generates N bits of output. HMAC_DRBG uses two compression fuction computations per N-bit output, and it's honestly hard to see that this makes a lot of sense. (I think I could come up with better uses of two compress() calls per output.)

b. It uses every bit of the compression function input. HMAC_DRBG's outputs are generated by a compression function computation where half or more of the message input block is known to the attacker.

c. It retains more state than HMAC_DRBG. HMAC_DRBG keeps N bits of secret state, and N bits of known variable state. KHF_DRBG keeps Q+M bits of secret state, and Q bits of known variable state.

Comments?

--John Kelsey, NIST, January 2004