Comments Received on SP 800-90A (2011)

From: SAKURAI Gen'ya" <g-sakura@ipa.go.jp>
Date: 8/1/11 5:50 AM

Dear Cryptographic Technology group experts,

I have reviewed the NIST SP800-90A, and have summarized the comments below.
Please consider the following comments in the revision process:

<1> technical comment
P.32 9.3
There is the following statement:
The generate algorithm will check that two consecutive outputs are not the same.

I think that this is intended to implement continuous RBG (or RNG) test as specified in FIPS 140-2. However, generate algorithms specified in this recommendation do not include any continous RBG test. To be consistent, it is recommended to add the step for continuous RBG test to each generate algorithm listed.

<2> editorial comment
Please replace 'auxilliary' with 'auxiliary'

<3> technical question/comment
P.76 11.3.6.2
There is the following requirement:
The DRBG shall not perform any DRBG operations while in the error state, and pseudorandom bits shall not be output when an error state exists.

Please revise the former part of the sentence, so that it is clear whether "any DRBG operations" include "uninstantiate function", and whether the following usage is allowed or not.

Naturally thinking, zeroization of the internal state should be prohibited if the testing on the uninstantiate function fails. If "any DRBG operations" include "uninstantiate function", zeroization of the internal state shall be prohibited while in the error state, even if the error is caused by the other DRBG mechanism functions than "uninstantiate function".

<4> editorial comment
About Appendix F
As written in FIPS 140-2 I.G. 14.5, not so many assertions are covered in CMVP validation process. Therefore it seems that the following sentence does not reflect the current situation. Conformance to many of the requirements in this Recommendation is testable by NIST's CAVP or CMVP.

<5>
About Appendix F

- The requirements in SP800-90A section 7.1 can be verified through CMVP validation, especially through Key Management section.
- The requirements in SP800-90A section 8.5 can be verified through CMVP validation, especially through Roles, Services and Authentication, and Key Management sections.
- The requirement in Section 8.6.3 can be verified through CMVP validation, especially through Key Management section.

<6>
About Appendix G

Please replace FIPS 180-2 with FIPS 180-3.
Please replace FIPS 198 with FIPS 198-1.

Best regards,

Gen'ya

--

Gen'ya SAKURAI

IT Security Center (ISEC),
Information-technology Promotion Agency, JAPAN

E-mail:g-sakura@ipa.go.jp
Tel:+81-3-5978-7545 FAX:+81-3-5978-7548
URL:http://www.ipa.go.jp

From: "Jacobson, David" <djacobso@qualcomm.com>
Date: 8/2/11 1:48 AM

This document seems to be have a somewhat different flavor than most NIST SP 800 documents in that in many places it reads like FIPS 140. In other words, it is not just about algorithms, but also about specialized hardware implementations with security boundaries, and about the validation requirements and process. Of course, such a change is your choice. But when adding this stuff in, there needs to be some background, so the reader without knowledge of FIPS 140 knows what is going on. In addition, the document should provide guidance for readers desiring a good algorithm but for whom FIPS approval is not a requirement.

I think it would be wise to have some introductory material describing the spectrum of possible implementations: a library module linked in with an application; an implementation in the system kernel or even hypervisor, where the memory accessible to only trusted modules; a firmware implementation in a tamper resistant module; a full hardware implementation. These different classes of implementations have different expectations, possible attacks, and infelicities.

For example, consider a library module that is linked in with applications. Perhaps it is linked because another library module needs random numbers, and the application programmer is not even aware that it is there. If the application forks, then we have the possibility of the parent and child processes generating the same random values. It would be good to give guidance on how to avoid such infelicities without depending on the application writer reading documentation on a library he doesn't even know he is using. (See below for a suggestion.)

On the other hand, in most systems, there is no way to really protect the internal state of an instantiation running in user space. "Mechanism boundaries" are at best gentlemen's agreements. About all that can be done is to provide only the proper instantiate, reseed, generate, and uninstantiate interfaces. Somewhere the draft says that the internal state must be inaccessible. This has to be interpreted in the context of the module type.

In my opinion, there are three goals of a random bit generator. For any one seed and sequence of calls (and their parameters), there is a fixed output sequence. First, we want all the bits of that sequence to have statistical properties such that the "look" random. (They are unbiased, and knowledge any subset does not help an attacker guess a bit he hasn't seen.) This property comes from the algorit ms. Second we want the probability that the same sequence is generated twice to be very small. This can be accomplished over time and space (instances) with sufficient entropy, over time with nonces, and over space with personalization strings. Third we want it to be hard for the attacker to guess the seed. Assuming that the nonce and personalization string are not secret, this requires good entropy. (In particular, it rules out things like seeding the RNG with the time of day in milliseconds---yes I found a generator in a well-known cryptographic library that did that.)

I think you should put a discussion about this in an informative section. Since some of these inputs are optional, it would help designers decide whether to implement each feature.

Back to the issue of the fork problem in library implementations. One way to fix this is for the generate function to always grab the process ID, and push it in as though it were "additional input". This way, if a fork has occurred, the parent and child (or multiple child processes) diverge, and do not generate the same sequence. You might want to consider legalizing this. An alternate, and perhaps better, solution is to trigger a reseed if the process ID changes. The generate function is limited to generating max_number_of_bits_per_request bits. For CTR_DBG this is 2^19 (64K bytes). This is a big pain for the application developers, as it forces them to use loops whenever they don't know a limit on the size of the request. It would be nice if you had the generate function divide the request into blocks of max_number_of_bits_per_request bits plus a fragment, and do the present generate on each block, reseeding as necessary.

Section 9.3 requires that the implementation check for consecutive outputs being equal. This is really bad! With CTR_DRBG, outputs can be up to 64K bits long. Naively this requires that the each value be saved for comparison. This eats a lot of memory. Even worse, it spoils backtracking resistance, as the attacker can who gets the internal state immediately has the last output. Of course, we can work around that by computing, say, a hash of the delivered bits and only saving the hash. But that eats a lot of cycles, and requires that a hash be available. I strongly urge you to get rid of this requirement. Besides, how can this possibly happen?

In the glossary, "Full Entropy" is defined as being within 2^-64 bits of ideal. This is ridiculously strict. If an generator has a strength of 256 bits, and the seed was really had only 255.99 bits of entropy, it wouldn't make any difference. The attacker could conceivable accomplish a brute force attack 0.69% faster, but a tiny optimization of his strategy would accomplish way more. Not only is is ridiculously strict, but there is no way to verify whether your source complies. I suggest that a half-bit of entropy shortfall be allowed. Even that would not have any real consequences, but would be a lot easier.

In the glossary, the definition of min-entropy, is not very good. It should just say that it is log2(1/prob(most_probable_output)). But it never actually says that.

There are many places where your are trying to say that the same chunk of entropy data shall not be used for some two or more purposes. One such place is in Section 8.2, where the draft says, "when reseeded, the seed shall be different from the seed used for instantiation". Again, Section 8.6.9 says, "The seed that is used to initialize one instantiation of a DRBG shall not be intentionally used to reseed the same instantiation or used as the seed for another DRBG instantiation." A similar statement appears in Section 8.6.10. You need to precisely define this concept and create a technical term, then use that term. This is important, since si be done. (Actually, the experience I had was when a testing lab interpreted the statement that the SEED and SEEDKEY must not be the same, to mean I had to actually do a comparison. The was for the generator in FIPS 186-2 Appendix 3.) Here is a quick try:

"cryptographically independent". Informally let it mean this: Two values are cryptographically independent if knowledge of either value gives no advantage in guessing the other to an attacker computationally limited according to the security strength. Now you can say "when reseeded, the seed shall be cryptographically independent of the seed used for instantiation." Alternatively you could use the already defined term "fresh entropy", but fix up the definition. The present definition is specific to one source and one DRBG. You should also state that the DRBG instance cannot be reseeded with its own output.

Section 8.6 says, "Reseeding is a means of restoring the secrecy of the output of the DRBG if the seed or internal state becomes known." Well, this is not a very good explanation. If you cared about leakage and attacker getting the state, the frequency would be based on the rate of leakage or a guess at how often an attacker might strike. Rather it appears that numbers for the reseed intervals are based on the algorithm, and probably are derived from an analysis of how many bits an attacker would need to begin to mount a cryptanalytic attack. The justification should be based on that.

Section 8.6.3 says, "The entropy input shall have entropy that is equal to or greater than the security strength of the instantiation." Yet no guidance is given as to how to verify this. Is analysis of the design sufficient? Are users expected to measure the entropy of the source? If so, how? Maybe this will be addressed in SP 800-90B or SP 800-90C. If so, you should mention that. If not, you might consider adding at least some informative material in an appendix about accessing entropy.

In Section 8.8, the paragraph defining prediction resistance seems to be in there twice.

In the glossary, the definition of entropy source is too specific. It wouldn't even cover flipping coins.

Section 7 says that Figure one provides a functional model of a DRBG, but the figure itself claims to be an RBG.

You might require the health test to collect a sample of the entropy source, and verify that it is plausibly random.

Section 9.1 says (of the entropy input and the nonce), "This input shall not be provided by the consuming application as in input parameter during the instantiate function." What does that mean? For the entropy, I think it means that the Get_entropy_input function must be called. But what about the nonce? If the implementation is not using random nonces, then it has to have either non-volatile memory or access to secure time, or something like that. This is making some pretty severe demands on a library-like implementation, or
even an implementation in an embedded system.

Section 9.2 says (regarding entropy for reseeding), "This input shall not be provided by the consuming application as an input parameter during theinstantiate request." Did you mean, "during the reseed request"?

You might want to require that generate and reseed have the semantics that if an error is returned, the internal state is unchanged. You can sort of guess that that was intended, but it would be good to make it explicit.

Section 11.3 says, "All data output from the DRBG mechanism boundary (or sub-boundary) shall be inhibited while these [health] tests are performed." Different instantiations are all independent, yet are all inside the same boundary. In order to test one instance, you would have to block all others. This is both difficult and unnecessary.

Section 11 says, "Therefore, entropy input used for testing shall not knowingly be used for normal operational use." Again, just make a statement that the entropy for normal use be fresh (or some other well defined term), and then this goes away.

Section 11.3.3 says that if reseeding is not supported, an instantiation that has exhausted its allowed generation count must be "shut down". I think you could say "uninstantiated", or if you wanted to keep that distinct from this situation instance."

The Block_Cipher_df (Section 10.4.2) is pretty complicated. It is similar to the counter-mode KDF of NIST SP 800-108. Many environments will already have available code to implement that. You might consider allowing the counter mode KDF of SP 800-108 with the fixedkey indicated in step 8 to be used. Yes, it would require more known answer test cases. But it would simplify implementations, and get rid of sort of mess whose complexity may dissuade implementers from supporting unconditioned entropy.

From: Tom Tkacik <tom.tkacik@freescale.com>
Date: Thursday, August 4, 2011 12:04 PM

On page 9, the definition of Source of Entropy Input states that "An SEI may be an approved entropy source". However, there are currently no NIST approved entropy sources. In other sections of the document, the phrase "appropriate entropy" is used instead.

On page 32, Section 9.3, list item 3 contains a new sentence. This states that the generate algorithm will check that two consecutive outputs are not the same. This was requirement in FIPS140-2, but is not a requirement of FIPS140-3. It is given here with the word "will" rather than "shall", so is not a requirement.

Is this requirement making a come back? It is not listed in Appendix H. Or, is this not an implementation requirement, but rather a statement that the Generate_algorithm actually does this. There does not appear to be any such test in the Generate_algorithm for any of the defined DRBGs.

Thank you for your consideration.
Tom

From: Stuart Audley <stuart@athena-group.com>
Date: Tuesday, August 9, 2011 12:43 PM

I think the new change in section 9.3 on page 32 needs clarification or removal. "The generate algorithm will check that two consecutive outputs are not the same."

This statement is not reflected in any of the generate algorithms in the draft.

This check can be interpreted to necessitate the storage of previously generated output in the DRBG which would mean previous output would be part of the current state and therefore the DRBG would fail to have backtracking resistance.

I think clarification on the definition of consecutive outputs would help. What would the length of outputs be (the requested generate length or an arbitrary length)? Would the consecutive output comparison occur between generates or only within a generate. If comparing output within a single generate request then some cases can be proven that two consecutive outputs won't be the same, for instance comparing 128bit blocks within a CTR_DRBG generate request.

Could the consecutive output check be performed outside the DRBG boundary instead of within the boundary?

Should an error or fatal error be asserted if two consecutive outputs are the same?

Is the motivation for this check to prevent double reading? If so, would other methods that prevent double reading satisfy the intended purpose of the check?

Best Regards,
Stuart Audley

The Athena Group, Inc.
408 W. University Ave. Suite 306
Gainesville, FL 32601
(352) 371-2567
www.athena-group.com