CVE-2017-5462: DRBG Flaw in NSS

Franziskus Kiefer
Mozilla, Germany
mail@franziskuskiefer.de

Vladimir Klebanov
VerifyThis.org
vladimir@verifythis.org

On April 19, 2017, Mozilla Foundation published the Security Advisory 2017-10 outlining several recently fixed security vulnerabilities.[1] One of these vulnerabilities, tracked as CVE-2017-5462, affects the Pseudo-Random Number Generator (PRNG) within the Network Security Services (NSS) library prior to version 3.29.5 and Firefox prior to version 53.

This document provides background information on the vulnerability and its discovery. While the security impact of the particular flaw is low, we take this opportunity to discuss several technical methods of quality assurance for PRNGs.

1. INSIDE THE NSS PRNG

NSS implements a so-called *Hash_DRBG* as PRNG, which is one of several PRNG schemes defined in the NIST Special Publication 800-90 [1][2]. While the standard contains all the details, the features relevant here can be summarized as follows.[3]

The state of a Hash_DRBG is composed of three values:

- A 55-byte integer state variable V, which is updated with each request of new bits
- A 55-byte integer constant C that depends on the seed
- A counter c tracking the number of requests for pseudo-random bits.

To generate random bits, Hash_DRBG concatenates

$$H(V) \,||\, H(V+1) \,||\, H(V+2) \,||\, \ldots$$

until enough bits are generated, where H denotes a cryptographic hash function. NSS uses the SHA-256 hash function with a digest length of 32 bytes. After generating new bits, the state variable V is updated according to the equation

$$V_{c+1} = V + H(0\text{x}03 \,||\, V_c) + C + c \qquad (1)$$

and the counter c is incremented by one. Addition is performed modulo $2^{440} = 2^{8\times 55}$, corresponding to the 55-byte size of V.

[1] https://www.mozilla.org/en-US/security/advisories/mfsa2017-10/#CVE-2017-5462

[2] NIST uses the name *Deterministic Random Bit Generator* (DRBG) for its schemes generating pseudo-random numbers, and *Non-Deterministic Random Bit Generator* (NRBG) for schemes generating "true" random numbers, otherwise also known as TRNGs.

[3] This summary is partially based on the answer posting of Paul Ebermann on Cryptography Stack Exchange: https://crypto.stackexchange.com/a/1399

The PRNG implementation can be found in the file `drbg.c`[4] within the NSS codebase.

2. THE BUG

The bug identified in CVE-2017-5462 is in the code implementing addition. The relevant code excerpt is shown in Figure 2. When the PRNG performs addition in equation (1), it uses the macro `PRNG_ADD_BITS_AND_CARRY`, which first delegates to the macro `PRNG_ADD_BITS` to add the two summands without considering the final carry and then the macro `PRNG_ADD_CARRY_ONLY` to add the carry.

The summands are a shorter number of length `len` stored in `add` and a longer number of length `dest_len` stored in `dest`. The result of addition modulo `dest_len` is stored, again, in `dest`. Note that numbers are represented as sequences of bytes, with byte number zero being the most-significant one.

In this setup, it is clear that the carry should be added at the position *preceding* the original most-significant-byte of the shorter of the two summands. This fact was supposed to be represented by the index `dest_len-len` supplied as parameter to `PRNG_ADD_CARRY_ONLY` in line 29. The essence of the bug is that `dest_len-len` does not point to the correct position of the carry, which should have been added at position `dest_len-len-1` instead.

This situation is illustrated in Figure 1 and a concrete example is shown at the end of Section 4.

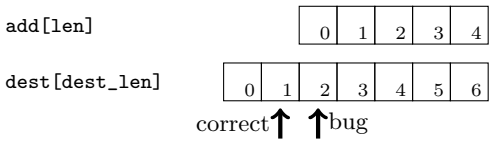

Figure 1: Position for adding the carry when adding two numbers (example with len = 5 and dst_len = 7). The flawed code shown in Figure 2 adds the carry at position dest_len−len = 2 instead of position dest_len−len−1 = 1.

3. FINDING THE BUG WITH TESTING

The easiest way to find most types of bugs in PRNGs following the NIST SP 800-90 standard is by testing the

[4] https://searchfox.org/nss/rev/fcdcad1fc1ddb6e70653637b0ea0f3359b8533f2/lib/freebl/drbg.c

```
/*
 * build some fast inline functions for adding.
 */
#define PRNG_ADD_CARRY_ONLY(dest, start, carry)    \
    {                                              \
        int k1;                                    \
        for (k1 = start; carry && k1 >= 0; k1--) { \
            carry = !(++dest[k1]);                 \
        }                                          \
    }

/*
 * NOTE: dest must be an array for the following to work.
 */
#define PRNG_ADD_BITS(dest, dest_len, add, len, carry)               \
    carry = 0;                                                       \
    PORT_Assert((dest_len) >= (len));                                \
    {                                                                \
        int k1, k2;                                                  \
        for (k1 = dest_len - 1, k2 = len - 1; k2 >= 0; --k1, --k2) { \
            carry += dest[k1] + add[k2];                             \
            dest[k1] = (PRUint8)carry;                               \
            carry >>= 8;                                             \
        }                                                            \
    }

#define PRNG_ADD_BITS_AND_CARRY(dest, dest_len, add, len, carry) \
    PRNG_ADD_BITS(dest, dest_len, add, len, carry)               \
    PRNG_ADD_CARRY_ONLY(dest, dest_len - len, carry)
```

Figure 2: Addition modulo as implemented in NSS

implementation with the reference seeds and outputs accompanying the standard. And indeed, the bug at hand did not go unnoticed after such testing functionality was implemented in NSS. For PRNGs not following the NIST standard, defining corresponding test suites is a good idea to avoid regressions during maintenance.

Functional unit testing would have caught this particular bug as well. As with any software, we advocate factoring out units with clear, testable functionality (such as addition here) also for PRNGs.

Statistical tests such as NIST SP 800-22 or DIEHARDEST[5] are not useful for testing cryptographic PRNGs. Such tests will not fail as long as the internal state of the PRNG does not stay constant and output passes through a cryptographic primitive (such as SHA-256) before reaching the consumer.[6]

4. FINDING THE BUG WITH THE ENTROPOSCOPE STATIC ANALYSIS TOOL

In this section, we describe how the flaw can be found[7] with the help of the Entroposcope tool developed by Felix Dörre and Vladimir Klebanov [2]. Entroposcope is a static analysis for detecting entropy loss in a PRNG. Entropy loss occurs when the number of possible output streams is less than the number of possible seeds, or equivalently, when two different seeds produce the same output stream (a situation also known as collision). Entroposcope is built on top of the bounded model checker CBMC [3], which in turn transforms the problem into a challenge for a SAT solver.

Considering the Hash_DRBG, the question whether the Equation (1) produces a collision and the DRBG loses entropy boils down to whether distinct 55-byte values x_1, x_2 exist, such that

$$x_1 + H(0\text{x}03 \,||\, x_1) = x_2 + H(0\text{x}03 \,||\, x_2) \ .$$

Now, it is clear that this question cannot be answered without either knowing the output of H for each of the 2^{440} inputs (which is infeasible) or some (unknown to us and certainly also to the tool) nontrivial mathematical argument on the nature of H in this context. In this regard, the Hash_DRBG differs from many other PRNGs that employ significantly simpler operations on the output of H (mostly copying) before it makes its way to the PRNG caller.

As a consequence of this design, we cannot use Entroposcope to prove absence of entropy loss in the Hash_DRBG. Nonetheless, we can still use it to find bugs. For this purpose, we consider an idealized "PRNG" with

$$H(b||V) = V \text{ and } C = V_0 \ .$$

This is the same kind of idealization that had helped us uncover previously unknown bugs in OpenSSL and GnuPG. With the idealization, on the first iteration ($c = 0$), Equation (1) becomes:

$$V_1 = 3 \times V_0 \ . \tag{2}$$

Since 3 and 2^{440} are co-prime, (2) will not produce a collision if implemented properly. By checking collision-freedom of the PRNG under an idealized H with Entroposcope, we can find bugs in the parts of the implementation that are not H.

Indeed, given the idealized code, Entroposcope produces a counterexample to entropy preservation with two concrete

[5] https://github.com/ticki/diehardest

[6] Statistical tests can potentially be of value in evaluating external sources of entropy, as long as the collected data has not been passed through a cryptographic primitive.

[7] ... and indeed was found independent of testing efforts.

seeds leading to the same output stream. Tracing these two executions makes it easy to pinpoint the cause of the collision in the addition code. The following example illustrates the collision with summands reduced to one byte in size for clarity.

EXAMPLE 1. *We note that* $3 \times 0x40 = 0xc0$ *under both proper and broken addition (the latter due to absence of carrying in this example). On the other hand,* $3 \times 0x95 = 0x01bf$ *under proper unbounded addition resp.* 0xbf *under proper modulo addition. Yet, the result under broken addition is again* $0x01 + 0xbf = 0xc0$.

5. MORE FORMAL METHODS FOR PRNG SECURITY

We note that Entroposcope is only concerned with entropy loss, which can be detected efficiently and subsumes many of the problems occurring in practice. Formal methods can also be used to prove stronger properties of cryptographic schemes and their implementations, albeit at the cost of higher effort. A machine-checked proof of security of HMAC_DRBG (another PRNG scheme from the NIST SP 800-90 standard) has been recently presented in [4]. The work shows that the output of HMAC_DRBG is pseudorandom under certain standard assumptions on the properties of its building blocks and that the C implementation of HMAC_DRBG in mbedTLS is functionally correct w.r.t. a formalization of the scheme.

6. REFERENCES

[1] E. Barker and J. Kelsey. Recommendation for random number generation using deterministic random bit generators. SP-800-90, U.S. DoC/National Institute of Standards and Technology, 2006. http://doi.org/10.6028/NIST.SP.800-90Ar1.

[2] F. Dörre and V. Klebanov. Practical detection of entropy loss in pseudo-random number generators. In *23rd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, 2016.

[3] D. Kroening and M. Tautschnig. CBMC – C bounded model checker. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[4] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *24th ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, 2017.