An Analysis of OpenSSL's Random Number Generator

Falko Strenzke

cryptosource GmbH,
Darmstadt, Germany
fstrenzke@cryptosource.de

Abstract. In this work we demonstrate various weaknesses of the random number generator (RNG) in the OpenSSL cryptographic library. We show how OpenSSL's RNG, knowingly in a low entropy state, potentially leaks low entropy secrets in its output, which were never intentionally fed to the RNG by client code, thus posing vulnerabilities even when in the given usage scenario the low entropy state is respected by the client application. Turning to the core cryptographic functionality of the RNG, we show how OpenSSL's functionality for adding entropy to the RNG state fails to be effectively a mixing function. If an initial low entropy state of the RNG was falsely presumed to have 256 bits of entropy based on wrong entropy estimations, this causes attempts to recover from this state to succeed only in long term but to fail in short term. As a result, the entropy level of generated cryptographic keys can be limited to 80 bits, even though thousands of bits of entropy might have been fed to the RNG state previously. In the same scenario, we demonstrate an attack recovering the RNG state from later output with an off-line effort between 2^{82} and 2^{84} hash evaluations, for seeds with an entropy level n above 160 bits. We also show that seed data with an entropy of 160 bits, fed into the RNG, under certain circumstances, might be recovered from its output with an effort of 2^{82} hash evaluations. These results are highly relevant for embedded systems that fail to provide sufficient entropy through their operating system RNG at boot time and rely on subsequent reseeding of the OpenSSL RNG. Furthermore, we identify a design flaw that limits the entropy of the RNG's output to 240 bits in the general case even for an initially correctly seeded RNG, despite the fact that a security level of 256 bits is intended.

1 Introduction

The ability to generate high entropy random numbers is crucial to the generation of secret keys, initialization vectors, and other values that the security of cryptographic operations depends on. Thus, random number generators (RNGs)

are the backbone of basically any cryptographic architecture. Numerous works have already dealt with the security of RNGs of operating systems [1],[2],[3],[4]. In [5], the predictability of OpenSSL's [6] RNG on the Android [7] operating system is investigated. That work reveals the problem of a too low entropy level of the OpenSSL RNG output as a consequence of its weak seeding through the operating system entropy sources at boot time.

In contrast, in the present work, we analyse the security features of the OpenSSL RNG itself. Specifically, we analyse the behaviour of the RNG in high and low entropy states. In a low entropy state, i.e. before the RNG has been properly seeded, we certainly know it to be unable to produce cryptographically secure random numbers – but we also expect it not to do any damage if we respect this condition by refraining from using it in cryptographic algorithms. Furthermore, in such a situation, we expect the RNG to produce cryptographically strong output after it has been reseeded with fresh high entropy seed data. As we shall see, neither property is fulfilled by the OpenSSL RNG.

We wish to point out that we are not addressing the RNG recovery problem by continuous entropy collection, which is for instance the subject of [8]. The OpenSSL RNG does not even attempt this, but solely relies on reseeding by the client application. The problems discussed here are in the context of the explicit invocation of these methods for reseeding the RNG from the client application.

As a very fundamental result, we prove that even when seeded initially with a 256 bit entropy seed, the RNG output may only have an entropy level of 240 bits for up to several hundreds of output bytes. The remainder of our findings is concerned with the behaviour of the RNG when it is initially in a low entropy level. We show that in this state, various functions of OpenSSL silently feed data to the RNG that is potentially secret and of low entropy. As a consequence, the RNG's function for outputting low entropy random numbers, which is available before the complete seeding of the RNG, is prone to leak these low entropy secrets. The potentially leaked values we have identified are keys of weak ciphers such as DES and the previous contents of buffers overwritten with random bytes. The latter is problematic in that wiping secret data by overwriting them with (pseudo) random data is an established practice.

Furthermore, we analyse the recovery ability of the RNG from a low entropy state. There are two scenarios in which this becomes relevant: first, the RNG might falsely presume a high entropy state based on false entropy estimations by the client application feeding it with seed data, or during the automatic seeding from the operating system RNG, which OpenSSL performs for instance on Linux systems. This makes our findings relevant for embedded system that feature only a small entropy level in their operating system RNG at boot time, for instance if they rely on reseeding the OpenSSL RNG by using a seed-file after the initial automatic seed from the operating system RNG. The second scenario is that though the RNG was seeded with correctly estimated high entropy data, its current state is revealed to an attacker through a break-in into the system. The latter scenario mainly applies to standard platforms such as servers or personal computers. In either scenario, it is vital that through the addition of further high

entropy seeding data to the RNG state an immediate recovery is possible in the sense that subsequently generated output will also be of high entropy. We find that this recovery essentially fails: when the RNG is in a state with low entropy level close or equal to zero, we show that despite the feeding of a arbitrarily high amount of entropy to the RNG, various attacks are possible that allow to predict previous and future output of the RNG with a computational effort of around 2^{80} hash evaluations.

The adversarial model underlying all the weaknesses presented in this work is that of a passive adversary who receives output from the attacked RNG and wishes to predict previous or future RNG output, or in some cases even the seed values. Most of the weaknesses identified in this work, in order to be actually exploited, demand a computational effort beyond what is believed be practical today even for computationally strong adversaries but might easily become feasible within a decade. Due to the entirely passive nature of the attacks, it would be possible for an adversary to record the RNG output and carry out the computational part of the attack at a later point in time when he has gained sufficient computational power.

All our results are based on the analysis of OpenSSL version 1.0.2a, but to the best of our knowledge apply to all versions, as the RNG is legacy feature of the library.

2 API, Life Cycle of OpenSSL's RNG, and the Associated Vulnerabilities

In this section we describe the RNG's API functions that are relevant to our analysis and how they relate to its life cycle states. The purpose of this description is to give a high level understanding of RNG's operation as it is necessary to understand the low entropy secret leakage issues discussed in Section 3 and introduce our formal life cycle states that are relevant for the remaining issues. Furthermore, we give an overview of the vulnerabilities presented in this work and relate them to the respective life cycle states where they are manifest. The core cryptographic operation of the RNG will be introduced later in Section 4.

The implementation of the RNG is found in the file `md_rand.c`. It defines the default RNG to be used in OpenSSL, though in principle the framework allows for switching to different RNG implementations provided by the user. As any purely software-based RNG it is based on a pseudo random number generator (PRNG). The API of functions related to OpenSSL's RNG are described in the respective manual page [9]. The functions relevant to our problem domain are described in the following.

```
void RAND_add(const void *buf, int num, double entropy)
```

"adds" the entropy contained in `buf` of length `num` to the RNG's internal state, where `entropy` shall be an estimate of the actual entropy contained in `buf`. The newly fed data modifies the RNG's state such that any subsequently generated random output will be affected by it. In this work we show that the claimed

"addition" of entropy suffers from a severe weakness. However, for the issue of low entropy secret leakage, this feature is irrelevant.

`void RAND_seed(const void *buf, int num)` is a wrapper for `RAND_add`. It calls that function with `entropy` = `num`, i.e. it expects the seed data to have maximal entropy.

`int RAND_poll()` draws entropy from the operating system's randomness sources, e.g. `/dev/random` on Linux, and feeds it to the RNG.

`int RAND_load_file(const char *filename, long max_bytes)` just loads a file and feeds up to `max_bytes` from that file to the RNG using `RAND_add`. Depending on the operating system, it feeds some further data to the RNG, but this is irrelevant to the issues discussed in this work.

`int RAND_bytes(unsigned char *buf, int num)` outputs `num` random bytes into `buf`. If the entropy level of the RNG, computed as the sum of the estimates provided by the calls to `RAND_add`, is less than the specified minimum of 256 bits, this function returns with an error code.

`int RAND_pseudo_bytes(unsigned char *buf, int num)`
performs the same operation for the random output generation as `RAND_bytes`, except that it also generates output if the minimum entropy level has not been reached.

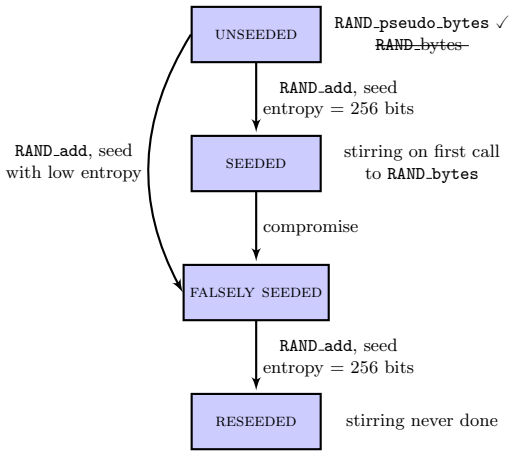

Fig. 1. Depiction of the life cycle states of OpenSSL's RNG.

Figure 1 shows the formalized life cycle states of the RNG. It always starts in the state UNSEEDED. In this state, it has zero entropy. If on the system a random device such as `/dev/random` on Unix is available, a call to `RAND_bytes` or `RAND_pseudo_bytes` will transfer the RNG automatically into the state SEEDED or FALSELY SEEDED by drawing 32 bytes of randomness from that device with a presumed entropy of 256 bits and feeding them to the RNG through a call to

RAND_add. The distinction between SEEDED and FALSELY SEEDED is not reflected by the RNG state directly, but only implicitly through the quality of the seed. In case of a seed with considerably lower entropy than 256 bits drawn from the random device, we identify the resulting state as FALSELY SEEDED. Another possibility of entering this state is through a compromise of the SEEDED state through a break-in into the system. Recovery from the FALSELY SEEDED state is attempted by feeding the RNG with a high entropy seed.

The resulting state is referred to as RESEEDED. It is distinguished from the SEEDED state by the fact that the so-called "stirring" operation, which distributes the entropy within the RNG state, is never carried out in this state. This operation is executed in the first call to RAND_bytes in the SEEDED state and never again after that. But, as we shall see, it is essential that it is carried out after a call to RAND_add for the distribution of the entropy in the RNG state and its safety. The details of these considerations will be given in the later sections when we turn to the core cryptographic design of the RNG and also explain the exact effect of the stirring operation.

We want to point out how easy it is to get into the RESEEDED state on a device with low boot time entropy provided by the operating system RNG. It is for instance achieved through the following sequence:
RAND_pseudo_bytes()
RAND_load_file().
The first call triggers the automatic initial seeding of the RNG. The second call attempts to seed the RNG using a high entropy seed file.

Table 1 shows an overview of our contributions. In the first column, following a short identifier for easier referencing of the respective issue, a brief description is given. In the next column the state in which the issue arises is listed. The remaining two columns specify the condition under which the issue arises and the number of the section within this work that explains the issue. LESLI is the abbreviation of "low entropy secret leakage issue". The label ELO... is based on the abbreviation of "entropy limitation of output". The ELO-issues apply to the roughly 1kB of output generated after the reseeding. The last two issues allow an attacker to recover the RNG state, if he gets output at a specific "position" after the reseeding. Here, "position" refers to an offset determined by the sum of length parameters to later calls to either RAND_add or RAND_bytes.

3 Low Entropy Secret Leakage in Low Entropy States of the RNG

A low entropy state of an RNG certainly makes it impossible to generate secure keys or to carry out cryptographic operations safely that depend on generation of random values. But there is no indication for application developers that are using a cryptographic library through its API to assume that it is generally unsafe either to use functions of the library appearing totally disjoint from the RNG functionality or to make use of the RNG for purposes where the low entropy state would not be a problem from a cryptographic perspective. Note that

Issue	State	Condition	Section
LESLI: low entropy secret leakage in output of `RAND_pseudo_bytes`	UNSEEDED, FALSELY SEEDED	attacker has access to output of `RAND_pseudo_bytes`	3
ELO-240: entropy limitation of the output of `RAND_bytes` to 240 bits	SEEDED	attacker has access to some output from the same call to `RAND_bytes` as that which he wishes to predict	5
ELO-80: entropy limitation of the output of `RAND_bytes` to 80 bits	RESEEDED	attacker has access to some output from the same call to `RAND_bytes` as that which he wishes to predict	6
ELO-160: entropy limitation of the output of `RAND_bytes` to 160 bits	RESEEDED	attacker has access to output after the reseeding	6
DEJA-SEED: recovery of the seed data of entropy of 160 bits and the resulting RNG state with an effort of about 2^{82} hash evaluations given that the seed is prepended with a known value of a specific length	RESEEDED	attacker has access to output after the reseeding at a specific offset	7.1
DEJA-STATE: for instance recovery of the RNG state after a 320-bit entropy reseed with an effort of 2^{84} hash evaluations	RESEEDED	attacker has access to output after the reseeding at a specific offset	7.2

Table 1. Overview of the identified weaknesses.

OpenSSL's function `RAND_pseudo_bytes` explicitly has the purpose of generating output before the RNG is sufficiently seeded. In the following sections we learn that OpenSSL violates the above assumptions, potentially resulting in the leakage of various secrets through the RNG output.

3.1 The General Problem

The basis of the low entropy secret leakage problems we investigate in the following, which we refer to as LESLI, is a fundamental one: given an RNG in a low entropy state, any further seed data fed to the RNG to increase its entropy, is leaked through the RNG output if the resulting state still fails to have a sufficient entropy level. The attack is simply carried out by iterating through all the possible seed inputs, generating the resulting outputs in the attacker's own instance of the RNG, and comparing these outputs to those of the attacked device. If they are equal, the seed values used in that attack iteration are the actual values used to seed the attacked RNG. Thus, any secret value that was part of the seed data is recovered. A requirement for this attack to work is that the number of output bits from RNG is approximately at least as high as the number of entropy bits in its state from the attacker's point of view. In general, the expectation value for the number of collisions, i.e. the number of wrong input values that map to the output value identified as correct, is

$$\mathrm{e} = \frac{2^n - 1}{2^l}, \tag{1}$$

where n is the entropy of the input in bits and l is the size of the output in bits. This relation holds under the assumption that the mapping of RNG input to its output is a random mapping. Assuming $n = l$, we find that on average there will be one collision.

On the basis of these considerations, it is rather doubtful that OpenSSL's manual pages suggest the feeding of low entropy secrets such as user-entered passwords through the function `RAND_add()` to increase the RNG's entropy level. They state: "RAND_add() may be called with sensitive data such as user entered passwords. The seed values cannot be recovered from the PRNG output"[10]. This is, as we have seen above, only true if the resulting entropy level of the RNG is sufficiently high.[1] From this analysis we learn that the feeding of low entropy secrets to RNGs such as that of OpenSSL is a risky and doubtful approach. It is only safe in situations where the RNG already has an entropy level that is secure with respect to brute force state recovery attacks – and in these situations it is needed the least.

However, to avoid the same-state problem, the feeding of low entropy data is indeed useful. Given that two systems share the same but otherwise high entropic RNG state, the feeding of a single bit with value zero to the first RNG and one bit with value one to the second, both RNGs will be in a secure state –

[1] Generally, an entropy of 80 bits is regarded as the minimum to achieve at least short term security.

as long as they don't appear as adversaries to one another. OpenSSL uses this approach to be secure with respect to the well known process-forking problem of RNGs by feeding the process-ID to the RNG before generating output [11].

But even forearmed with this knowledge, not following OpenSSL's manual pages' encouragement to feed low entropy secrets to the RNG, we run into problems, as various OpenSSL API functions silently feed secret data to RNG, as we explain in the following.

3.2 Leakage of Secrets Overwritten with Random Data

The first leakage problem we present occurs in the following scenario: Assume an application is developed for an embedded system using OpenSSL as the cryptographic library. The system designers are aware of the fact that they might have a low entropy state problem in OpenSSL's RNG. However, they only use OpenSSL for the following purposes (possibly they might later seed it with a fresh high-entropy seed and use it also for different purposes): They overwrite short PIN numbers the system temporarily stores during user interactions. They follow a common security advice to overwrite these critical secrets with random numbers using OpenSSL's `RAND_pseudo_bytes` function. This measure to wipe secret data is not necessarily useful in all application contexts. However, it is useful when one cannot exclude the possibility of working on memory-mapped files[12]. Furthermore it can be useful to prevent compiler optimizations from removing the wiping procedure [13][2]. Another reason for this measure is side-channel security: using a fixed byte value such as zero to overwrite the secret bytes could result in leakage of their Hamming weights through power consumption or electromagnetic emission.

Furthermore, in our scenario, we assume that the application uses the RNG to generate weak random numbers with calls to `RAND_pseudo_bytes`, which are output by the device, for instance as nonces for cryptographic purposes.

Neither usage of the potentially predictable random numbers is a problem from a cryptographic point of view: given that there is at least some minimal entropy in the RNG state, side-channel attacks will be severely complicated and also the other two purposes of the random overwriting will not be impeded. Nonce values only need to have the property to be non-repetitive, a property that is not affected by the low entropy state problem even if the RNG state was completely known to the attacker – at least until a reboot that might incur the same-state problem.

However, in the described usage scenario, the secret PIN is leaked through the random numbers output by the device. This is due to the fact that the `RAND_add` function uses the initial content of the memory area to be overwritten as an additional seed value. For the detailed description of function of `RAND_add`,

[2] In that reference randomizing the target buffer is not suggested, however, since calls to an RNG function have a side effect (on the RNG state), it is almost impossible for the compiler to remove that call. OpenSSL itself uses a similar approach internally, though not by using an actual RNG.

which shows how exactly the initial buffer contents affect the PRNG state, refer to Section 4. The RNG's state becomes dependent on the PIN number, and the attacker simply has to execute a brute force search on the joint input space of all possible RNG states before the call to `RAND_add` and all possible PIN values as additional seeds and match the resulting RNG output to the nonces recorded from the device under attack.[3]

Furthermore, despite the realistic scenario where the `RAND_pseudo_bytes` function is used to wipe secrets from RAM, there is certainly also a potential leakage problem when previously uninitialized buffers are overwritten with random bytes. Uninitialized buffers on the heap or stack may also by chance contain sensitive low entropy data from previous operations of the application. Which memory locations are reused in parts of the program on the heap or stack is often deterministic or under the influence of the attacker. In addition, a program may implement buffer reuse at the source code level as an optimization technique.

3.3 Leakage of DES Keys in PKCS#8 Conversion

In the file `evp_pkey.c`, in a function used for converting private keys to PKCS#8 format, `RAND_add` is called with the key data as a seed. Given that DES keys, that might otherwise be used in a secure cryptographic construction, can be brute force attacked, they are at risk to be leaked through `RAND_pseudo_bytes`.

4 Detailed Description of OpenSSL's RNG

Algorithm 1 Simplified algorithmic description of `RAND_add`

Input: md_0, s, p, q, $b = (b_0||b_1||\ldots||b_{n-1})$ where each b_i is 20 bytes long except for the final one which is potentially shorter

Output: md_0, s, p, q

1: **for** $i = 1$ to n **do**
2: $\quad t = \mathrm{size}(b_{i-1}) - 1$
3: $\quad \mathrm{md}_i = \mathrm{SHA1}(\mathrm{md}_{i-1}||s[p : p + t \bmod 1023]||b_{i-1})$
4: $\quad s[p : p + t \bmod 1023] = s[p : p + t \bmod 1023] \oplus \mathrm{md}_i[0 : t]$
5: $\quad p = p + t + 1 \bmod 1023$
6: **end for**
7: $\mathrm{md}_0 = \mathrm{md}_i \oplus \mathrm{md}_0$
8: $q = \min(q + \mathrm{size}(b), 1023)$
9: **return** md_0, s, p, q

[3] Note that the usage of uninitialized memory for the purpose of random number generation can lead to an even greater threat, namely the compiler's decision to remove subsequent operations on variables that become "tainted" by the uninitialized data [14]. However, this does not seem to apply to OpenSSL's implementation [15].

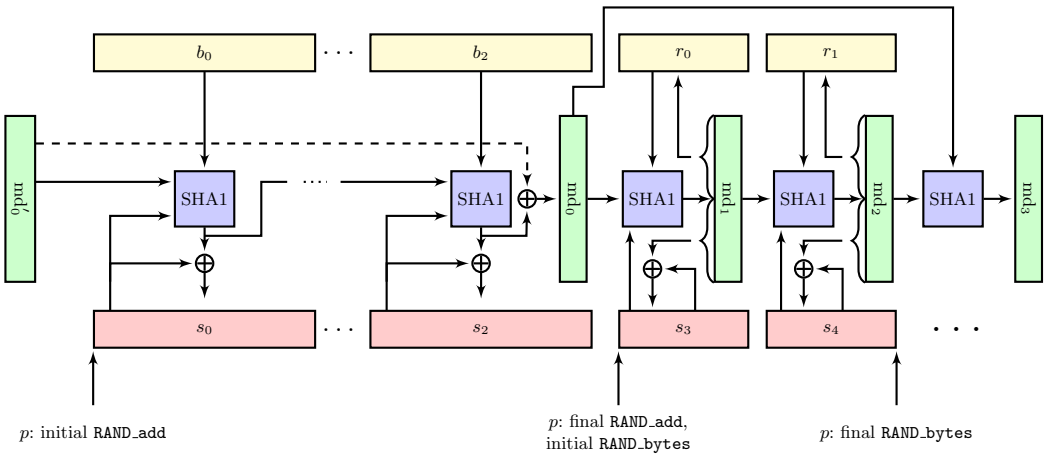

Fig. 2. Depiction of actions induced by a call to function `RAND_add` and a subsequent call to `RAND_bytes`. The blocks b_i, s_0 to s_2 have a length of 20 bytes each, r_i, s_3 and s_4 have a length of 10 bytes each.

Algorithm 2 Simplified algorithmic description of `RAND_bytes`

Input: md_0, s, p, q, $r = (r_0||r_1||\ldots||r_{n-1})$ where each r_i is 10 bytes long except for the final one which is potentially shorter
Output: md_0, s, p, $r = (r_0||r_1||\ldots||r_{n-1})$

for $i = 1$ to n **do**
  $\text{md}_i = \text{SHA1}(\text{md}_{i-1}||r_{i-1}||s[p : p + 9 \bmod q]$
  $r_{i-1} = \text{md}_i[10 : 10 + \text{size}(r_{i-1}) - 1]$
  $s[p : p + 9 \bmod q] = s[p : p + 9 \bmod q] \oplus \text{md}_i[0 : 9]$
  $p = p + 10 \bmod q$
end for
$\text{md}_0 = \text{SHA1}(\text{md}_i||\text{md}_0)$
return md_0, s, p, r

Algorithm 3 The stirring operation executed within `RAND_bytes`

$i = 0$
$c = c[0 : 19]$ // constant
while $i < 1023$ **do**
  call `RAND_add` with $r = c$
  $i = i + 20$
end while

In this section we give a complete description of the OpenSSL's RNG, since this is necessary to explain the further vulnerabilities presented in this work. However, we omit some details such as the seeding of the RNG with the process-ID (PID) and certain counter values. The PID generally does not feature any entropy since PIDs are predictable on Linux [16], and the counters only depend on the number of bytes provided in the calls to `RAND_add` and `RAND_bytes` and thus can be assumed to be known from the application program's source code and the sequence of high level operations. For a description involving these counters, see [5]. Furthermore, management operations such as checking and updating the level of the estimated entropy are ignored. Algorithm 1 and 2 provide the algorithmic descriptions – simplified in this sense – of the functions `RAND_add` and `RAND_bytes`. First, we explain the symbols used in the algorithmic description. The RNG state is comprised of four elements: md_0 refers to a message digest of 20 bytes length. The state bytes are an array of 1023 bytes represented by s. Furthermore, the index of the current state byte is labelled p (corresponds to the variable `state_index` in the source code) and q is the sum of the number of state bytes updated by `RAND_add` (corresponds to the code variable `state_num`). Both algorithms update all four state elements, with the exception that `RAND_bytes` does not update q. Both s and md_0 have arbitrary starting values as they use uninitialized memory. However, this can in general not be viewed as reliable source of entropy. The initial value of p and q is zero. q is increased up to 1023 in each call to `RAND_add` by the number of input bytes. For most of the analyses we conduct in the following sections q is always equal to 1023, since in the state RESEEDED, which will be the starting point for all remaining issues except for $ELO - 240$, the previous stirring operation has already increased q to 1023.

Algorithm 1 specifies the cryptographic operations carried out by a call to `RAND_add`. Apart from the four state elements, it has an additional input b which represents the seed data. It is partitioned into blocks b_i having a size of 20 bytes each, except for the last block, which has a potentially smaller size. The "size()" operation returns the size of the respective block in bytes. In the loop, iteratively, new values of md_i are computed as the SHA1 hash value of the specified elements. Here "$x[y]$" indicates the y-th byte of x and "$||$" denotes concatenation. $x[y:z]$ is the block formed by the bytes $x[y]||x[y+1]||\ldots||x[z]$, where the index z may also indicate a byte position before y, in which case the wraparound is performed at the highest byte position of x. During each iteration, up to 20 state bytes are updated with the iteratively computed md_i. At the end of the operation md_0 is updated as the XOR of the previous value of md_0 and the final md_i.

The action of the function `RAND_bytes` is given in Algorithm 2. Additionally to the state elements, the memory area r to be filled with random bytes is an input and output value. Here, r is partitioned into blocks r_i, each having a size of 10 bytes except for the last block, which again has a potentially smaller size. After the computation of the value md_i in each iteration of the loop, one half of this byte string is XORed back to the state bytes that were used to feed the hash function in that iteration, and the other half is written to the 10 byte output block r_{i-1}. If the final output block is shorter than 10 bytes, a correspondingly

smaller number of bytes is written to it. Finally, the value md_0 as part of the RNG state is updated.

Note that both Algorithms 1 and 2 implement a seamless wraparound when reaching the end of the state bytes s, so that the beginning and end of this array do not have any special properties. Algorithm 3 specifies the stirring operation. As already explained in Section 2, this algorithm is carried out on the first call to `RAND_bytes` after the RNG state has reached an entropy level of 256 bits by its own accounting based on the entropy measures provided in the calls to `RAND_add`. Here, c is a 20 byte constant value. By feeding a total of 1040 bytes to the RNG, a certain distribution of the entropy in the state s is achieved.

Figure 2 depicts the operations carried out by a call to `RAND_add` and a subsequent call to `RAND_bytes`. On the bottom, the respective values of p are indicated: its initial value at the start of `RAND_add`, its value after the execution of that function, which corresponds to the initial value in the subsequent call to `RAND_bytes`, and its final value after termination of the latter function. In the figure, for better readability, the state bytes have been arranged as blocks s_i. Note that this partitioning of the state bytes is done dynamically within both functions starting from the current position indicated by p when they are called.

5 Restriction of the RNG's Output to an Entropy of 240 Bits

The first design flaw of OpenSSL's core RNG, which we refer to as ELO-240, leads to the possibility that generated keys are limited to 240 bits of security whereas it tries to achieve 256 bit – however, due to lack of documentation, the intended security level can be only inferred from the source code. In contrast to further weaknesses reported in this work, this issue arises even when the RNG is initially seeded with correctly estimated high entropy data before any output is generated. From a practical point of view, this problem is meaningless, since it does not allow any practical attacks, not even in the foreseeable distant future, but it shows us a design flaw of the RNG, which will turn out to be relevant for the further issues reported in this work. From a strictly formal point of view, we find that it is commonly agreed that keys with 256 bit security shall be used for applications where long term security matters, as it is reflected by the standardized key sizes for AES and elliptic curve keys, and that thus an RNG should produce output with the corresponding entropy level.

In order to understand how this limitation arises, we consider the following example. The RNG, in its initial state, is seeded with a 256 bit entropy seed, the length of which is rather irrelevant as long as it remains considerably shorter than the state length of 1023 bytes. To simplify things, we assume that the length of the seed data is 256 bits. As a result of this seeding operation, the first 32 bytes of the state contain high entropy data, the remaining state bytes contain zero entropy, and p points to the byte with index 32 (counted from zero). With a subsequent call to `RAND_bytes` the stirring operation is induced. With a sequence of calls to `RAND_add`, the stirring operation, Algorithm 3, completely

cycles over all state bytes once, except for the first 17 bytes pointed to by p before the operation, which are processed twice. During this operation, from the entropy added into the first 32 bytes, only 160 bits flow into the remaining state bytes through md_0. Accordingly, an attacker could theoretically enumerate all possible values of the state bytes from position 32 to 1022 by 2^{160} guesses. We now assume that a call to `RAND_bytes` is made to draw 42 random bytes from the RNG. The first ten bytes are generated on the basis of current state block indicated by p. During the output generation, the new value of md_i is calculated as $\mathrm{md}_1 = \mathrm{SHA1}(\mathrm{md}_0||r_0^{\mathrm{init}}||s_0)$, with r_0^{init} being the initial value of the output buffer, which we assume to carry no entropy. Half of md_1, i.e. 80 bits, are output as r_0. Then the following 10-byte blocks and the 2-byte final block are computed iteratively in the same manner. Now assume that the first generated 10-byte block, r_0, is output to the attacker. This means that the attacker learns 80 bits of md_1. Accordingly, the entropy of md_1 is reduced to 80 bits from his point of view. Since all state bytes that flow into the output generation of the further output blocks r_1, r_2, and r_3 have a total entropy of 160 bits as we have seen above, and that 80 bits of entropy flow into the generation from md_1, the remaining generated 32 bytes not output to the attacker have an entropy of 240 bits.

Note that this theoretical attack does not apply when the attacker receives output bytes from one call to `RAND_bytes` and the output value he wishes to predict from a different call. This is due to the fact that at the end of `RAND_bytes`, as given in Algorithm 2, md_0 is updated as $\mathrm{md}_0 = \mathrm{SHA1}(\mathrm{md}_i||\mathrm{md}_0)$, where md_0 on the right hand side is another source of entropy, which foils his knowledge gained through the output r_0. Thus, we conclude that this weakness can be seen as due to a design fault which causes the security of the RNG output to depend on the call sequence. In reality it may be well possible that a client application generates multiple symmetric keys for different users or a key along with the first CBC initialisation vector in a single call to `RAND_bytes`. In the theoretic view of RNGs, implementation details such as concrete call sequences to draw random bytes find no regard.

6 Low Entropy Recovery Failure

We now investigate what happens when OpenSSL's RNG executed the stirring operation in a low entropy state and the client application subsequently adds high entropy seed data to the RNG before generating 256 bit cryptographic keys. The two issues identified in this scenario are ELO-80 and ELO-160.

A principally useful notion for a function to feed entropy to an RNG is that of a mixing function, defined as follows [2]: A mixing function f guarantees that

$$H(f(I,S)) \geqslant H(S) \text{ and } H(f(I,S) \geqslant H(I),$$

where $H()$ denotes the Shannon entropy, I the input seed data and S the state of the RNG. A function adhering to this notion guarantees that it does not reduce

the entropy of the RNG state and that after its operation the state will have at least the same entropy as the input seed data.

From a purely formal perspective, OpenSSL's RNG fulfils both requirements. However, the definition of the mixing function as provided in the reference and shown above turns out to be of limited usefulness: Given an RNG that uses only a part of its internal state for the production of output, such as it is the case with OpenSSL' RNG, the definition should refer to the entropy of newly generated output instead of that of the state S. With respect to this adjusted definition of a mixing function, we will learn that `RAND_add` fulfils the first requirement, but not the second: when the RNG is in a low entropy state, even after adding high entropy seed data, the RNG will effectively remain in a low entropy state in short term, i.e. generate low entropy output.

We develop the following scenario. After an initial entropy update of the RNG of 32 bytes with a believed entropy of 256 bits, but an actual entropy of $x < 256$ bits, the stirring operation is carried out. This causes, following the analysis from Section 5, the whole state $s[0:1023]$ to contain x bits of entropy. Now we assume that a second call to `RAND_add` is made, this time with a 32 byte string with the full entropy of 256 bits. After the stirring operation, p pointed to position $32 + 17 = 59$. Thus the high entropy update affects the state bytes $s[59]$ to $s[90]$, and p afterwards points to $s[91]$. Now a call is made to `RAND_bytes` for the output comprised of the 10 byte blocks r_0, r_1, ..., r_y. The only limitations of the length of the output is that it may only cause `RAND_bytes` to process low entropy state bytes, i.e. not to reach the high entropy block starting at position 59 again, and thus can be of a length of several hundreds of bytes. The first three blocks r_0, r_1, and r_2 are output to the attacker. We now show that he can fully recover the remaining output r_3, r_4, ..., r_y with a complexity of 2^{80+x} hash evaluations. The analysis follows that of Section 5.

Through the stirring operation carried out after the initial addition of the x bit entropy seed, x bits of entropy were distributed to all the state blocks $s[i]$. After the second high entropy update, only the state bytes $s[59]$ to $s[90]$ hold 256 bits of entropy. In the subsequent call to `RAND_bytes`, the only source of entropy are md_0, carrying 160 bits, and the processed state bytes starting from $s[91]$. After having seen output r_0, the entropy of md_1 is reduced to 80 bits for the attacker. He now iterates through all the possible values of the unknown 80 bits from md_1 and x bits of the initial entropy seed, i.e. a total of 2^{80+x} possibilities. Each guess for the 2^x initial states implies a value for state bytes $s[91]$ through $s[1022]$. With access to the values of r_0, r_1 and r_2 he can reliably identify the correct guess by comparing his simulated RNG output to the actual output. He now has completely determined the state of the RNG except for the high entropy block spanning from $s[59]$ to $s[90]$ and the value of md_0 before the call to `RAND_add`. He can thus now predict as many output bytes from the same call as can be generated before again processing the high entropy state bytes, which is a little less than one 1 kB. After that call, he loses information about the new value of md_0, according to the update $\mathrm{md}_0 = \mathrm{SHA1}(\mathrm{md}_i||\mathrm{md}_0)$ at the end of `RAND_bytes`. This is the manifestation of ELO-80.

If the attacker has no access to output from the same call, we find the ELO-160 issue, described in the following. After the high entropy reseed without the stirring operation carried out, output bytes of the RNG while processing the low entropy state bytes $s[91]$ through $s[1022]$ have an entropy of $160 + x$ bits. However, if he sees enough output bytes from a single call to `RAND_bytes` to reliably determine the initial seed with entropy level x and thus the values of all the state bytes $s[91]$ through $s[1022]$ as described above, then the entropy of output produced with further calls to `RAND_bytes` when processing these state bytes is only 160 bits (stemming only from md_0).

7 State Recovery Attacks in the RESEEDED State

We now investigate the possibility of another class of attacks against the RESEEDED state. We label these attacks after the deja-vu effect since they exploit the "re-entering" of the state bytes where the high entropy seed was added by exploiting the wraparound at the end of the state bytes in the RNG. In the following sections, we develop two different attacks in the same scenario: we assume that in the FALSELY SEEDED state with zero entropy a high entropy bit string v is fed to RNG. The first attack, DEJA-SEED, explained in Section 7.1, recovers the seed v and the RNG state after the reseed. The second attack, subject of Section 7.2, is named DEJA-STATE and only recovers the RNG state after the feeding of v. Both attacks recover any secret random values generated after the reseeding.

For the development of both attacks in the following sections, we assume an initial seeding with an entropy of zero. Section 7.3 explains how the attacks can be adjusted if that is not the case and how this affects their complexity.

7.1 Recovery of RNG State and Seed Data

The DEJA-SEED attack presumes the following scenario: The RNG is in the state FALSELY SEEDED with zero entropy. Then, a 160-bit entropy seed v is fed to the RNG using `RAND_add`. Afterwards, a 160-bit key (for instance for HMAC) is generated. To simplify the discussion, we use again fixed positions of the state bytes. Let the 160-bit entropy seed data be fed to the RNG when $p = 40$. Assume the added seed data v is of the following form: a publicly known 10-byte constant part followed by the 20 byte seed data with full entropy. After the seed has been added, p points to s[70]. Now the 160-bit key k is generated (from the analysis of Section 6 it follows that it achieves the full 160-bit entropy level, provided that no RNG output from the same call to `RAND_bytes` is accessible to the attacker). Afterwards, either through further additions of seed data or through output generation, p reaches the value 0 again. In this situation, a 90 byte output is generated with a single call to `RAND_bytes`, which is accessible to the attacker. The attacker uses this output to recover the seed v and subsequently the 160-bit key k as follows.

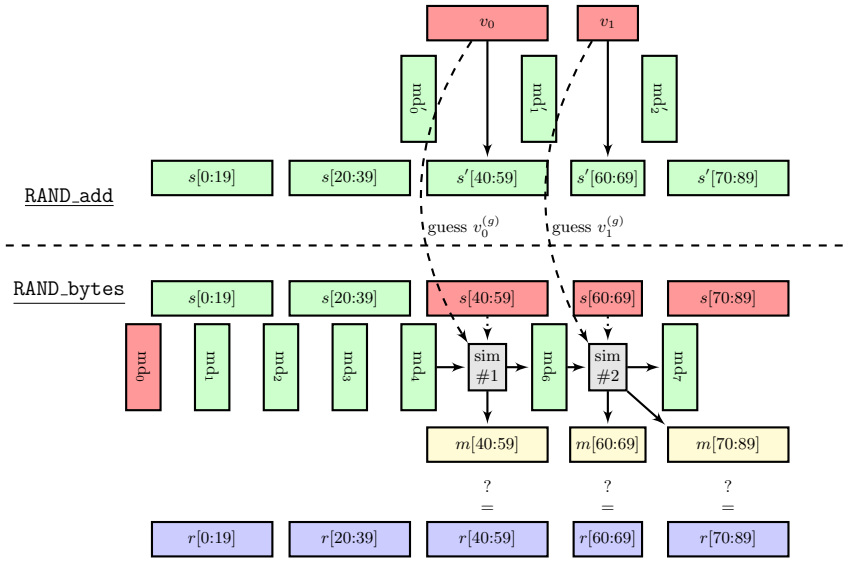

Fig. 3. Depiction of the DEJA-SEED attack.

Figure 3 depicts the attack. The state bytes shown at the top represent the RNG's state after the initial zero entropy seed and before the high entropy seed with v. As indicated, the feeding of v alters the state bytes $s[40:69]$. The values of md entering the respective computations are shown above the state bytes as md'_i, with md'_0 being the value before the call to `RAND_add`. Here, the state bytes prior to the feeding are labelled as s' where they differ from those of the state s after the feeding of v and the key generation. In the lower half of the figure, the execution of the call to `RAND_bytes` is depicted that produces the output the attacker uses for the attack. Below the state bytes s after the feeding of v, the values of md_i belonging to the respective RNG state are shown. The attacker uses $r[0:39]$ to recover md_1 with an effort of 2^{80} simulations of output generation: With the knowledge of $r[0:9]$ he has to guess 2^{80} possibilities of the other half of md_1. The identification of the correct guess is reliable since he can use $r[10:39]$ with a total size of 240 bits for the matching.

For the output generation of a single 20-byte block, a SHA-1 computation with an input of 38 bytes has to be carried out. Since in the vast majority of the guesses, already $r[10:19]$ differs from the simulated output, the cost for the generation of further simulated output values for the few cases where $r[20:39]$ has to be checked, can be ignored. Accordingly, the cost for this step is 2^{80} computations of SHA-1 with a 38-byte input.

Note that the state block $s[0:39]$ is still completely known to the attacker and thus the 80-bit second half of md_1 is the only value unknown to him that serves as an input to the output generation of $r[10:39]$. The attacker knows the

value of md_i for as long as known state bytes are processed, i.e. until the start of the processing of $s[40]$. From Algorithm 1, Steps 3 and 4, we find the relation

$$s[40:59] = s'[40:59] \oplus \mathrm{SHA1}(\mathrm{md}'_0||s'[40:59]||v_0) = f(v_0), \tag{2}$$

where md'_0 is the value of md_0 at the beginning of Algorithm 1 during the feeding of v and $s'[40:59]$ indicates the respective value of the variable prior to the feeding of v and which is thus known to the attacker. The result of the hash function on the right hand side amounts to md'_1. Accordingly, we view $s[40:59]$ as a function $f()$ of v_0, the only unknown value for the attacker in the update of $s[40:59]$. The seed bytes of v_0, which flow into that computation, contain 80 bits of entropy: their first half is the fixed 10 byte value, their second half has full 80 bit entropy. The attacker recovers the value of v_0 as follows: He iterates through all the 2^{80} possible values of v_0. For each guessed value $v_0^{(g)}$, he computes the resulting value of $s[40:59]$ according to (2). This procedure is indicated in Figure 3 as "sim#1". For each guess of $v_0^{(g)}$, this procedure outputs a value $m[40:59]^{(g)}$ where, according to Algorithm 2, Steps 2 and 3,

$$m[40:49]^{(g)} = \mathrm{md}_5[10:19] = \mathrm{SHA1}\left(\mathrm{md}_4||r[40:49]^{\mathrm{init}}||f(v_0^{(g)})[0:9]\right)[10:19],$$

$m[50:59]^{(g)}$ is computed accordingly as

$$m[50:59]^{(g)} = \mathrm{md}_6[10:19] = \mathrm{SHA1}\left(\mathrm{md}_5||r[50:59]^{\mathrm{init}}||f(v_0^{(g)})[10:19]\right)[10:19],$$

and r_i^{init} indicates the initial contents of the output buffer, which in our model does not contain any entropy. If the attacker finds $m[40:59]^{(g)} = r[40:59]$, then he concludes that $v_0 = v_0^{(g)}$ and he has determined the first seed block.

Since here the output space ($r[40:59]$ with 160 bits) used for the verification has the double size of the input space (v_0 with 80 bits), the chance for a collision is overwhelmingly small and the attacker can determine $v_0^{(g)}$ with certainty.

After having recovered v_0, he applies the same brute force recovery to v_1. This procedure is denoted as "sim#2" in the figure. The only difference to the attack on v_0 is that for each guess of $v_1^{(g)}$, also the subsequent state bytes starting from $s[70]$ are determined. Using the 320 bits of $r[60:69]$ and $r[70:89]$ to match the guess values $m[60:69]^{(g)}$ and $m[70:89]^{(g)}$, he can reliably verify his 80-bit guess of v_1. At this point, the attacker has completely recovered the state of the RNG after the feeding of v and v itself. If no further entropy was added after the feeding of v, he can predict any future output of the RNG. In any case he recovers the key k directly generated after the feeding of v.

In this example, the attacker needs an effort of $3 \cdot 2^{80} \approx 2^{82}$ hash evaluations (recovery of md with 56-byte hash input; of v_0, and v_1 each with 38-byte hash inputs, and with an average effort of 2^{80} hash evaluations for each of the three) to recover a 160 bit seed. With less prepended constant data, and thus a greater entropy in v_0, the attack complexity increases accordingly.

7.2 Recovery of only the RNG State

In this section we present the DEJA-STATE attack, also applicable in the RNG state RESEEDED. It is similar to the DEJA-SEED attack from Section 7.1, the difference being that not the high entropy seed value is recovered, but only the RNG state. Furthermore, there is no requirement on the seed data v to be prepended with constant data, since here we attack the state bytes in blocks of 80 bits, as they are processed by `RAND_bytes`, anyway. This attack allows the prediction of all output of the RNG after the high entropy reseeding like in the DEJA-SEED attack.

We assume that a 160-bit seed with full entropy was fed to the RNG through a call to `RAND_add` when p pointed to $s[40]$. Accordingly, the state bytes $s[40:59]$ are affected by this update. The attack starts in the same way as the DEJA-SEED attack from Section 7.1, including the recovery of md_1 through the first output blocks, up to the point where in the DEJA-SEED attack the values for v_0 and v_1 would be guessed. From this point on, the DEJA-STATE attack proceeds as follows.

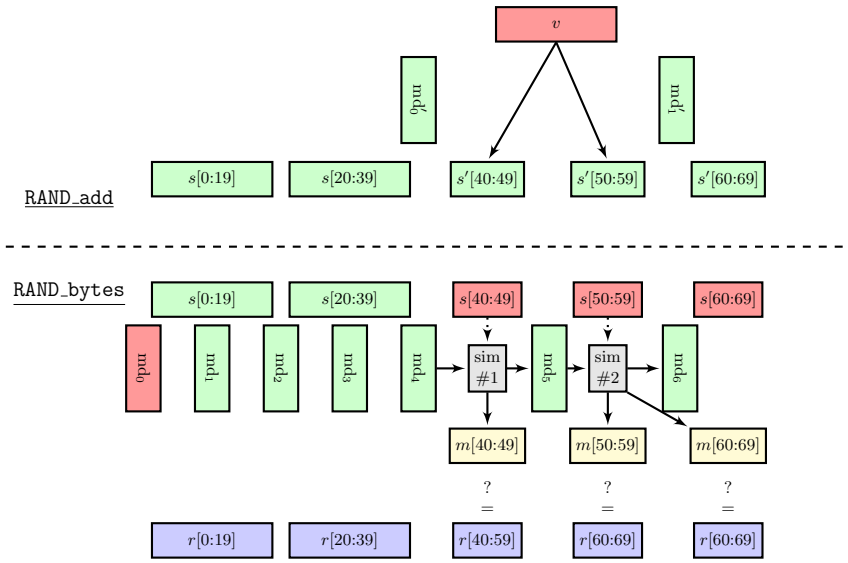

Fig. 4. Depiction of the DEJA-STATE attack.

The state of the attack is that md_4, the value entering the output generation of $r[40:49]$ based on $s[40:49]$, is known. Figure 4 depicts this analogously to the previous attack. Now the attacker performs the following steps:

1. He iterates through all the 2^{80} possible values of $s[40:49]$. For each guess $s[40:49]^{(g)}$, he simulates the attacked RNG's output generation, creating a

match value

$$m[40:49]^{(g)} = \mathrm{md}_5[10:19] = \mathrm{SHA1}\left(\mathrm{md}_4 || r[40:49]^{\mathrm{init}} || s[40:49]^{(g)}\right)[10:19].$$

Here we again assume all values r^{init} to be known. He determines the correct guess as the one where $m[40:49]^{(g)} = r[40:49]$. This allows him to verify his input space of 2^{80} on an output space of 2^{80}, giving him an expectation value for the number of collisions of one according to (1), since also a hash function with truncated output can be viewed as a random mapping. In the following, we proceed with the description of the attack as though there were no collisions and take them into account when we calculate the complexity of the attack. This procedure is indicated as "sim#1" in the figure.

2. He applies the same procedure to recover $s[50:59]$, where he uses the updated value of md that enters this iteration as md_5, thus recovering the value of md_6. This procedure is indicated as "sim#2" in the figure, which also includes the following two items.
3. Now he knows $s[40:59]$ and also md_6. Since he also knows $s'[40:59]$, the state before the feeding of v, he can calculate md'_1, the value of md after the processing of v during `RAND_add`, as $\mathrm{md}'_1 = s'[40:59] \oplus s[40:59]$, according to Algorithm 1, Step 4.
4. He computes the final updated value of md at the end of `RAND_add`, during the feeding of v, as $\mathrm{md}''_0 = \mathrm{md}'_1 \oplus \mathrm{md}'_0$, according to Algorithm 1, Step 7. Again, md'_0 is known to him as a value from the state prior to the feeding of v. With $s[40:59]$ and the updated value md''_0, he has recovered the complete state after the reseeding with v. This allows him to identify the correct guess for $s[40:49]$ and $s[50:59]$ with respect to the occurring collisions based on further output from $r[60]$ on.

We now estimate the average complexity of the attack: The attacker iteratively applies the single block recovery procedure of a complexity of 2^{80} hash evaluations to the initial recovery of md_0 and to each of the two blocks $r[40:49]$ and $r[50:59]$. Since for each block he has on average one collision, on average he has to process $r[50:59]$ two times. This means he has to go through an effort of 2^{80} hash evaluations for four times on average, thus yielding an average effort of 2^{82} hash evaluations of the attack.

The cost estimation for this attack when two full 20-byte blocks with full entropy are used in the seeding is achieved easily: there is on average an additional effort in terms of hash evaluations of $4 \cdot 2^{80}$ for the third and $8 \cdot 2^{80}$ for the fourth 10-byte block, yielding a total average effort of $16 \cdot 2^{80} = 2^{84}$ hash evaluations.

In our attack, we chose a length of a single 20 byte blocks on purpose to simplify the description. Now we consider the case of seed data the length of which is not a multiple of 20-byte blocks. If the seed, for instance, has a length between 31 and 39 bytes, the second block is shorter than 20 bytes. Then, according to Algorithm 1, in Step 4 only a part of md'_2 is XORed to $s[70:79]$. This means that the attacker can recover only a part of md'_2. Assuming for instance a seed length of 32 bytes, 64 bits of md'_2 are not recoverable from the state bytes.

Accordingly, they have to be recovered through brute force effort, using further output blocks for the matching. Obviously, if the final block v_l becomes shorter than 80 bits, it is more efficient to iterate through the possible values of v_l than to guess the lost part of md'_2. From these considerations we see that all possible seed value lengths of maximally 40 bytes can be attacked with an extra average effort 2^{83}, where we also account for the 8 expected collisions when processing $r[70:79]$.

Like in the DEJA-SEED attack from Section 7.1, the attacker recovers completely the state of the RNG after the feeding of v. Thus he can predict all RNG output from that point on without any further effort.

7.3 Dealing with Non-Zero Initial Entropy

In Sections 7.1 and 7.2 we have assumed an initial entropy of zero for the RNG state before the reseed. Given that the seed used for the initial seeding of the RNG had a low but non-zero entropy of x bits, two approaches can be considered dealing with this entropy in the DEJA-SEED and DEJA-STATE attacks.

If the attacker has access to RNG output before the reseeding, he can recover the initial state with an effort of 2^x hash evaluations. This effort will be negligible compared to that of the presented attacks for actual low entropy states. If the attacker has no access to output prior to the reseeding, he has to recover the initial state parallel to the recovery of md_1 using the first output blocks. This means that additionally to the 2^{80} input space of the unknown half of md_1, he has also to iterate over the 2^x input space for the initial entropy (each guess for the initial seed implies a guess for the whole state s), yielding a total effort of 2^{80+x} hash evaluations. Thus, depending on the initial entropy level, this can become the dominating cost for DEJA-SEED and DEJA-STATE attacks.

8 Conclusion

In this section we discuss the impact, the possibilities for removing of the discovered issues and summarize the theoretical conclusions of our findings.

8.1 Theoretical Aspects of our Findings

Our central result is that under worst conditions, OpenSSL's RNG only achieves a security level of 80 bits. This sounds devastating, but when we discuss the impact of our results, we will find that from a realistic perspective the majority of real-world systems will be affected to a lesser extend, if at all.

The common notions of security applied to RNGs [18,4] are the well established *forward* and *backward security*, i.e. the security of an RNG under the assumption of the disclosure of its state at a point forward or backward in time, respectively; as well as the notion of *resilience*. The latter would be violated if an attacker can reduce the entropy of the RNG's output by feeding specially prepared seed data to the RNG.

From these notions, backward security is not even attempted by the RNG itself, but when it is attempted by the client application, then it suffers from the DEJA-SEED and DEJA-STATE attacks that allow the disclosure of the RNG state based on the generated output after a reseeding. Accordingly, the backward security of the RNG is impaired. The RNG's forward security in the state RESEEDED is affected by the same attacks since they allow the recovery of previous output – which certainly is also possible with access to the RNG state instead of its output.

Our findings do not suggest that the RNG's resilience is defective in any way. As it seems, the transformations leading to updated RNG states during the addition of seed data are sound in this respect.

Moreover, we find that the addition of seed data to the RNG is not optimal in the sense that if the RNG is in a low entropy state, then the added seed data remains recoverable from the RNG state with a much lower complexity than that corresponding to their combined entropy. Since, to our knowledge, the security of the seed data in the RNG state is not covered by any of the existing notions, we propose the *forward security of seed data* as a new security notion for RNGs.

We also had to point out a shortcoming in the existing notion of a mixing function: using the entropy of the RNG state in its definition, its application leads to meaningless results if the RNG does not use its complete RNG state in a symmetric way for the output production. Accordingly, we propose the notion of an *effective mixing function*, wherein the role of the RNG state is replaced by the subsequent output of the RNG.

What remains from our findings is the low entropy secret leakage issues (LESLI). It is a rather trivial requirement for an RNG not to produce output before sufficiently seeded, however, to our knowledge, this has so far only been viewed from the angle of the low-entropy-recovery problem, which imposes different restrictions than the prevention of seed data leakage. Accordingly, also this problem seems to require formal treatment in RNG security models.

8.2 Impact

Estimating the impact of our findings, we conclude that, excluding the possibility of system break-ins revealing the RNG state to an attacker for the time being, any application running on a system that features a sufficiently high boot time entropy and automatic seeding of the OpenSSL RNG will be safe, since then the initial seed will be of high entropy. The same applies to systems that perform the high entropy reseeding before ever having generated any random numbers. Thus, the only issue such systems suffer from will be ELO-240, meaning that the RNG will produce about 1KB of 240-bit entropy output before it runs at full 256-bit security. Though a clear error in the cryptographic design, this vulnerability can be seen as purely cosmetic problem of the RNG, as generally no system building on ordinary software implementations will be able to achieve the corresponding security level with respect to other aspects such as a general assured security features and physically secure key storage.

However, for any system where the potentially automatic seeding from the operating system RNG delivers a low entropy level, or where this feature is absent, and the client application for instance relies on the loading of a seed-file, the issues ELO-80, ELO-160, DEJA-SEED and DEJA-STATE come up as a threat. The first two are a comparatively minor threat, since ELO-80 depends on the attacker receiving some bytes from the same call to `RAND_bytes` as the one he wishes to predict output from, which is presumably not possible in most designs. And ELO-160 at least maintains a reasonable security level of 160 bits which will most likely remain impossible to break even for "nation-strength" adversaries in the long term. However, especially DEJA-STATE is applicable in general scenarios and has such a small complexity that it forms a concrete threat, since an entropy level of around 80 bits is generally assumed to provide only short term security. So-called "lightweight" cryptographic algorithms such as PRESENT [17] provide 80 bit security. Even though today such a computational effort must be assumed to still be impossible to realize, this can quickly change in the near future. Since all the attacks presented in this work are entirely passive procedures, the RNG output values can be recorded and the computational part of the attack be carried out once the necessary computational resources become available to the attacker. From this perspective, the OpenSSL RNG must be considered as broken in the RESEEDED state, i.e. a scenario where the stirring operation is ceased due to a falsely believed high entropy level. In order to assess the security of an application on a potentially vulnerable system, one must assure that before the high entropy seeding, no other low entropy seeding with subsequent output generation was performed. This is an undesirable situation, since such an analysis is inherently non-local, making it a complex and error prone task.

Good news is that RSA key generation remains safe if it is performed directly after the reseeding, i.e. without any other output generation in between. First of all, there are no RSA keys in use with a security level of 256 bits (this would correspond to a modulus of 15360 bits), so that ELO-240 has no effect, but much more importantly, in an RSA key generation so much output is drawn from the RNG that none of the issues apply any more after such an operation. This is due to the fact that also the RNG's output generation has a "stirring" effect on the state. However, this helpful effect of RSA key generation certainly only applies if it is performed before the generation of any other random output.

LESLI remains an issue for any system that temporarily operates in the UNSEEDED or FALSELY SEEDED state. Here, the leakage of intentionally overwritten or uninitialized/reused memory is a potential threat that could affect real world systems. But the number of actually exploitable applications must be deemed to be small, since in general the use of the library with the RNG being in a low entropy state will be unintentional and is thus likely to incur greater problems than low entropy secret leakage.

The main category of systems potentially affected by the any of the issues identified in this work must be assumed to be embedded systems and mobile platforms. As it is well known, such platforms often feature insufficient entropy

levels of their operating system RNGs at least at boot time, without this being detectable by the RNG implementation. Accordingly, the use of a seed file, which stores entropy for applications across application restarts or system reboots, is a common mitigating measure under these circumstances. As we have learned, this approach is at risk to be affected by the issues reported in this work.

8.3 Repair of OpenSSL's RNG

From our analysis of the individual issues it becomes clear what the two main problems of OpenSSL's RNG are: the cessation the stirring operation after having entered the SEEDED state, and subversion of its remaining backbone, the running md, by leaking 80 bits of it through the output. We want to point out that this problem must have been clear to some extent to the designer, since a source code comment in `md_rand.c` just above the code implementing the stirring operation states:

```
/*
 * In the output function only half of 'md' remains secret, so we
 * better make sure that the required entropy gets 'evenly
 * distributed' through 'state', our randomness pool. The input
 * function (ssleay_rand_add) chains all of 'md', which makes it more
 * suitable for this purpose.
 */
```

However, it seems that neither the exact nature of the entropy distribution through the stirring operation nor the necessity of being able to recover from a compromised state at any point during the RNG's life cycle was seen. In any case, if the RNG was intentionally designed to be only one-time seedable, then at least this would have to be stated in the documentation.

The straightforward repair is given by two measures. First, make the updated md in `RAND_bytes` dependent (through hashing or XOR) on the previous value of md after the generation of each block, as it is the case when the vulnerable implementation is used only with calls generating 10 bytes or less of output. Second, the stirring operation must be carried out after each call to `RAND_add` before the generation of new output.

This brings us to a further point, already discussed in the previous section, namely the forward security of the RNG with respect to the recovery of potentially low entropy seed data fed to it. With the current approach of executing the stirring function from `RAND_bytes`, even in the case of correct entropy estimation during the initial seeding, the seed values would remain recoverable through an attack in the style of DEJA-SEED directly on the state bytes s in the period between their feeding through `RAND_add` and the next call to `RAND_bytes`. The correct approach would be to implement `RAND_add` in such a way that a brute force attack on the state s would have the same complexity as the total entropy of all the data fed to it so far.

Concerning LESLI, the only solution is to let `RAND_pseudo_bytes` generate output using a different RNG state than that used for `RAND_bytes`. This would also remove another subtle issue concerning the security with respect to process

forking and the RNG's entropy calculation, which is reported in a source code comment of `md_rand.c` itself.

The above described measures will remove all vulnerabilities discovered in this work. However, with respect to efficiency aspects, a standard construction for instance using a CTR mode generator would be much more favourable than a repair of the current design. An AES-based RNG will generally be able to achieve a higher efficiency due to the wide-spread hardware support for this cipher, whereas hardware support of hash functions is very rare. As it becomes evident from our results, the employment of such a large state as used by the OpenSSL RNG does not have any positive effects on security. It remains completely unclear what was the goal of this design choice. A substantial reduction of its size could also help to increase the RNG's performance.

8.4 Countermeasures in Client Code

In order for users of vulnerable versions of OpenSSL to be able to use the RNG without being affected by the issues ELO-240, ELO-80, ELO-160, DEJA-SEED, and DEJA-STATE, in Appendix A, we provide secure wrapper functions for OpenSSL's RNG functionality. The LESLI issue is based on a fundamental design problem that cannot be repaired by wrapper functions. Accordingly, we advise not to use the function `RAND_pseudo_bytes` at all. All calls to that function should be replaced by calls to `RAND_bytes` instead. Note that it is important to check the return value of `RAND_bytes`: if this function fails due to an insufficiently seeded RNG, although returning an error value, it still outputs random bytes. Our secure version of `RAND_bytes` retains this behaviour. Failing to check the error value means, among other problems, that the LESLI issue remains even when abstaining from using `RAND_pseudo_bytes`.

Calls to `RAND_add` shall be replaced with `RAND_add_secure_240bits` or `RAND_add_secure_256bits`. Which of the respective version, "240bits" or "256bits" shall be used depends on whether the user deems it sufficient to have 240 bit entropy output or needs the full 256 bit security. These functions are implemented as follows: after adding the seed through `RAND_add`, the 240 bit version executes a single stirring operation, the 256 bit version repeats this operation once more. The secure functions to replace `RAND_seed`, `RAND_poll` and `RAND_load_file` follow the same principle and naming convention.

Calls to `RAND_bytes` shall be replaced with `RAND_bytes_secure`. This function sets the target buffer to all zeroes before calling `RAND_bytes`, thus avoiding the leakage of its previous contents in future RNG output under all conditions. If this behaviour is not desired, the function can be modified correspondingly. Furthermore, the function makes calls to `RAND_bytes` block-wise with a block length of maximally 10 bytes. This avoids the issues that rely on the learning of half of md through the RNG output, which are ELO-240, ELO-80, DEJA-SEED, and DEJA-STATE.

For users which cannot dispense with `RAND_pseudo_bytes`, we also provide the function `RAND_pseudo_bytes_secure`, which at least prevents leakage of the previous contents of the target buffer.

References

1. Gutterman, Z., Pinkas, B., Reinman, T.: Analysis of the Linux Random Number Generator. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy. SP '06, Washington, DC, USA, IEEE Computer Society (2006) 371–385
2. P. Lacharme, A. Röck, V. Strubel, M. Videau: The Linux Pseudorandom Number Generator Revisited (2012) http://eprint.iacr.org/2012/251.pdf.
3. Kaplan, D., Kedmi, S., Hay, R., Dayan, A.: Attacking the Linux PRNG on Android: Weaknesses in Seeding of Entropic Pools and Low Boot-time Entropy. In: Proceedings of the 8th USENIX Conference on Offensive Technologies. WOOT'14, Berkeley, CA, USA, USENIX Association (2014) 14–14
4. Dodis, Y., Pointcheval, D., Ruhault, S., Vergniaud, D., Wichs, D.: Security Analysis of Pseudo-random Number Generators with Input: /Dev/Random is Not Robust. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. CCS '13, New York, NY, USA, ACM (2013) 647–658
5. Kim, S.H., Han, D., Lee, D.H.: Predictability of Android OpenSSL's Pseudo Random Number Generator. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer – Communications Security. CCS '13, New York, NY, USA, ACM (2013) 659–668
6. The OpenSSL Library: www.openssl.org.
7. The Android operating system: https://www.android.com/.
8. Dodis, Y., Shamir, A., Stephens-Davidowitz, N., Wichs, D.: How to Eat Your Entropy and Have It Too - Optimal Recovery Strategies for Compromised RNGs. In Garay, J., Gennaro, R., eds.: Advances in Cryptology - CRYPTO 2014. Volume 8617 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2014) 37–54
9. OpenSSL: Manual page of RAND https://www.openssl.org/docs/crypto/RAND.html .
10. OpenSSL: Manual page of RAND_add() https://www.openssl.org/docs/crypto/RAND_add.html .
11. Viega, J.: Practical Random Number Generation in Software. In: Proceedings of the 19th Annual Computer Security Applications Conference. ACSAC '03, Washington, DC, USA, IEEE Computer Society (2003)
12. Ferguson, N., Schneier, B., Kohno, T.: Cryptography Engineering. John Wiley & Sons, Inc. (2010)
13. Software Engineering Institute: https://buildsecurityin.us-cert.gov/bsi-rules/home/g1/771-BSI.html.
14. Xi Wang: More randomness or less (2012) http://kqueue.org/blog/2012/06/25/more-randomness-or-less/.
15. Falko Strenzke: uninitialized RAM and random number generation: threats from compiler optimizations (2015) http://cryptosource.de/posts/uninit_data_rng_en.html.
16. Michael Brooks: stackoverflow: How does Linux determine the next PID? http://stackoverflow.com/questions/3446727/how-does-linux-determine-the-next-pid.
17. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: THE PROCEEDINGS OF CHES 2007, Springer (2007)
18. Barak, B., Halevi, S.: A Model and Architecture for Pseudo-random Generation with Applications to /dev/random. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. CCS '05, ACM (2005) 203–212

A Secure Wrapper Functions for the OpenSSL RNG

In this section we give secure wrapper functions for the OpenSSL RNG functionality. Please refer to Section 8.4 for an explanation of the countermeasures used in these functions.

```
void RAND_add_secure_240bits( const void* buf , int num, double entropy)
{
  int n = 1023;
  const unsigned char dummy_seed[20] = { 0 };
  if(buf)
  {
    RAND_add(buf, num, entropy);
  }
  while (n > 0)
  {
    RAND_add(dummy_seed, sizeof(dummy_seed), 0.0);
    n -= sizeof(dummy_seed);
  }
}

void RAND_add_secure_256bits(const void* buf , int num, double entropy)
{
  RAND_add_secure_240bits( buf, num, entropy);
  RAND_add_secure_240bits( NULL, 0, 0.0);
}

void RAND_seed_secure_240bits(const void *buf, int num)
{
  RAND_add_secure_240bits(buf, num, (double)num);
}

void RAND_seed_secure_256bits(const void *buf, int num)
{
  RAND_add_secure_256bits(buf, num, (double)num);
}

int RAND_poll_secure_240bits()
{
  int result = RAND_poll();
  RAND_add_secure_240bits( NULL, 0, 0.0);
  return result;
}

int RAND_poll_secure_256bits()
{
  int result = RAND_poll();
  RAND_add_secure_256bits( NULL, 0, 0.0);
  return result;
}

int RAND_load_file_secure_240bits(const char *file, long max_bytes)
{
 int ret = RAND_load_file(file, max_bytes);
 RAND_add_secure_240bits(NULL, 0, 0.0);
 return ret;
}

int RAND_load_file_secure_256bits(const char *file, long max_bytes)
{
 int ret = RAND_load_file(file, max_bytes);
 RAND_add_secure_256bits(NULL, 0, 0.0);
 return ret;
}
```

```
int RAND_bytes_secure(unsigned char *buf, int num)
{
  memset(buf, 0, num);
  int final_ret = 1;
  while(num)
  {
    int ret;
    int this_round = num > 10 ? 10 : num;
    ret = RAND_bytes(buf, this_round);
    if(ret != 1)
    {
      final_ret = ret;
    }
    buf += this_round;
    num -= this_round;
  }
  return final_ret;
}


int RAND_pseudo_bytes_secure(unsigned char *buf, int num)
{
  memset(buf, 0, num);
  return RAND_pseudo_bytes(buf, num);
}
```