Strong(er) Randomness

Robert Edquist Linus Back
Email: {robed335,linba708}@student.liu.se
Supervisor: Jan-Åke Larsson, {jan-ake.larsson@liu.se}
Project Report for Information Security Course
Linköpings universitet, Sweden

Abstract

A Cryptographically Secure Pseudo Random Number Generator (CSPRNG) is an algorithm that produces random entropy for cryptographic purposes, requiring it to fulfill certain criteria other than the ones that apply to a regular pseudo random number generator (PRNG). In this report we take a look at what differentiate a CSPRNG from other PRNGs and how it is tested and proved. We investigate some instances where weaknesses in CSPRNGs have been found and what consequences they have had, resulting in our conclusion that it is hard to test randomness and that a weak CSPRNGs effects can propagate far and cause a lot of problems.

1. Introduction

The aim of this report is to give some clarity to what makes a pseudo random number generator cryptographically secure. In addition to this we will take a look at some examples of reports detailing cases when weak pseudo random generator has been used in cryptographic systems.

The first sections will be dedicated to the difference between a good pseudo random number generator and a cryptographically secure pseudo number generator. We will later move on to look at which the most commonly used types of random number generators are. In connection with this we will look at what effects a cryptographically weak pseudo number generator has on a cryptographic system.

Lastly we will look at and discuss some previous cases/reports on weak randomness being used in implementations, specifically we will take a closer look at the "Dual_EC_DRBG" case, addressing its use, how it is a problem and how problems of this kind can be solved.

In short the following questions will be asked, addressed and answered in the report.

- What makes a pseudo random generator cryptographically secure?
- What is the difference between a good pseudo random generator and a cryptographically secure pseudo random generator?
- What are most common types of pseudo random generators used?
- What effect does a weak pseudo random generator have on a cryptographic system such as OpenSSL or on an elliptic curve discrete logarithm problem?
- What report has there been on weak randomness being used in implementations (For example in the case of Dual_EC_DRBG)?

2. Background

PRNGs are used in many different applications in computer science and are central in the areas of simulations and cryptography [1].

Historically tables have been used for mathematical computations that require random numbers. This changed with the invention of the electronic computer during the 1940s. Partly because of the fact that mathematical computations could be calculated faster, demanding an increase in the speed at which numbers where delivered. But also because the sequence of numbers needed were longer than before.

Certain physical processes can create truly random numbers but this approach were rejected at the time because of the need to be able to re run the same computation, with the same random numbers, again during testing [2].

An arithmetic approach called the middle-square method was proposed by John von Neumann in 1949. The algorithm works by taking a number and squaring it, taking the middle-part of the resulting bigger number as output, and for the next random number repeating the process but using the output as input. This algorithm was known to have flaws, for example if using a 4-digit input, an output of 0000, will continue to generate only 0000 as output. The algorithm was used despite this, as it was faster than using random numbers read from punch cards [2].

PRNGs have since been refined and evolved and are today critical components of cryptographic systems, where if you break the random number generator often the whole security system is broken. Ideally to get good results from the PRNGs, making the cryptographic systems secure, we need a good truly random source for the seed and high entropy.

Often, practically, more random numbers are needed than hardware generation can provide and PRNGs can then use the existing entropy and "stretch" it to provide larger random numbers. This generation of random numbers in the PRNG uses entropy obtained from a higher quality source, like a hardware random number generator or an unpredictable system process, and then generates larger random numbers based on this entropy, circumventing the slow processes for extracting randomness from a running system.

2.1 Concepts

Randomness in the form of a sequence means we have a series of numbers or symbols without a predictable pattern, unordered and without coherence. A random sequence, in the mathematical cryptology field, consists of numbers that does not conform to a deterministic pattern.

Determinism is the view that for every event there exist conditions that could cause no other event. In the context of cryptology the concept of determinism can be viewed as "cause and effect", prior events completely determine later events.

Entropy, within computer science and cryptology, is the randomness collected by the OS or application to be used for cryptographic purposes or other purposes requiring random data. Usually collected through hardware, examples include mouse movements, keyboard, IDE timings or other similar sources of random noise [3]. The original term stems from physics and is a measurement of the degree of a system's disorder, a property of the system's state [4].

2.2 Terminology

PRNG (pseudo random number generator) also called DRBG (deterministic random bit generator), will be used in the report to describe the algorithms designed to generate a sequence of numbers with properties as close as possible to truly random numbers.

Seed, is the number used to initialize the PRNG, often crucial to the security of the cryptographic system since having the seed will often allow one to obtain the output of the PRNG. Random seeds for the PRNG are usually generated by collecting entropy from the computer system on which the PRNG is running.

CSPRNG is a PRNG that is suitable for cryptographic purposes, and it is an abbreviation for cryptographically secure pseudo random number generator.

Procedural generation, a term widely used within media production. It refers to content generated algorithmically rather than manually, meaning spontaneous or extemporaneous created content and not content created prior to distribution.

3. General overview

3.1 PRNG requirements

PRNGs are algorithms that generate sequences of numbers that approximates the properties of random numbers, the sequence is not truly random but based upon a set of initial values called a seed. PRNGs are important in practice due to their ability to generate larger quantities of numbers than otherwise would have been possible through hardware during the same time.

PRNGs are generally used in applications such as simulations and procedural generation, these kinds of applications usually require less elaborate algorithms than PRNGs aimed at cryptographic applications. Common types of PRNG algorithms include linear congruential generators, Lagged Fibonacci generators (an improved variant of the linear congruential generator) and linear feedback shift registers. [5][6][7]

To make sure the output produced by the PRNG is sufficiently random, careful mathematical analysis usually has to be performed to validate the statistical properties of the PRNG. This is means the PRNG will be put through a battery of statistical tests, measuring the quality of the PRNG and its statistical randomness.

3.1.1 Statistical randomness and statistical tests

A numeric sequence is considered statistically random when it does not contain any recognizable patterns or regularities. Statistical randomness is not the same as true randomness and only implies an objective unpredictability, a pseudo randomness suitable for statistics, hence the name statistical randomness.

There are a lot of different tests relating to statistical randomness, some of the first were four test developed by M.G. Kendall and B.B. Smith in the 1930s [8]. These four test was, the frequency test (checking that there were about the same number of 0s, 1s, 2s etc.), the serial test (basically checking the same thing but for 00s, 01s etc.), the poker test (checking the distribution of sequences like xxxxx, xxxxy, xxxyy etc.) and the gap test (looked at the distance between numbers, here for example 00 would have a distance of 0, 010 a distance of 1 and 01230 a distance of 3).

As the use for random numbers have increased, a greater number of test with increasing sophistication has been created to verify the randomness of the sequences. One of more modern would be the “Diehard tests” developed by George Marsaglia. This test suite was first released in 1995 and consists of a battery of test designed to measure the statistical randomness quality of a PRNG. The following are some (not all) of the test that are included in the test suite [9], with a short description of what the test might involve (not intended as complete explanation).

- *Birthday spacing’s*. The spacing’s between randomly chosen points on an interval should be asymptotically exponentially distributed. The name is derived from “the birthday paradox”.
- *Overlapping permutations*. Analyzes sequences of five consecutive random numbers and checks if the 120 possible orderings of these occur with statistically equal probability.
- *Ranks of matrices*. Involves selecting a number of bits from a number of random numbers, then forming a matrix. Determining the rank of this matrix and counting the ranks.
- *Monkey tests*. Sequences of numbers are seen as words and the overlapping words in a stream are counted. The number of sequences that do not appear should follow a known distribution. The name is derived from “the infinite monkey theorem”.
- *Count the 1s*. Involves counting the 1 bits in successive or chosen bytes, then converting this number to a “letter”, and counting occurrences of five-letter "words".
- *The squeeze test*. Involves multiplying 231 with random floats until reaching 1. Repeating this, a great number of times, then looking at the distribution of the number of floats needed to reach 1.

If a sequence given by a PRNG is able to pass a statistical test, it usually does it with a certain degree of significance that is used to determine the degree of statistical randomness for the algorithm. PRNGs require these tests as verification of their randomness, since they are based on deterministic algorithms and not truly random.

3.1.2 Mersenne twister

The Mersenne twister is the most widely used PRNG, the period length of the algorithm is chosen to be a Mersenne prime (a prime number on the form 2^n-1) [10], hence the name. It was developed by Makoto Matsumoto and Takuji Nishimura in 1997, providing fast generation of high quality pseudo random integers.

The most commonly used versions of the Mersenne twister is based upon the prime $2^{19937}-1$, and there are two standard implementations; MT19937 using a 32-bit word length and MT19937-64 using a 64-bit word length. One advantage the Mersenne twister has, a desirable property, is its long period ($2^{19937}-1$), not a guarantee of quality for a PRNG but there can be problems with short periods.

The Mersenne twister in its original form is however not suitable for cryptographic purposes, observing a sufficient number of iterations will allow an attacker to predict all future iterations. For the MT19937 the exact number of iterations needed is 624, which corresponds to the state vector from which all future iterations are generated.

These kind of cryptographic disadvantages are not due to faulty programming, and can often not be fixed with a tweak, but a result of the algorithm not fundamentally designed to be cryptographically secure. A CSPRNG often needs to be designed from the ground up to be secure and not modified from an non-secure PRNG.

3.2 Difference between PRNG and CSPRNG

CSPRNGs can be seen as a subclass to PRNGs, since it is a kind of PRNG with some additional requirements to make sure it is suitable for cryptographic use.

A requirement for a CSPRNG is that it must pass all polynomial time statistical tests in connection with the seed, compared to a regular PRNG which is only required to pass certain statistical tests. It should also be impossible to reconstruct the stream or predict future values of the stream even if the seed is revealed. These properties cannot be proven. However they can be reduced to hard mathematical problem, such as integer factorization, that in turn can provide strong evidence for the security of the CSPRNG.

Among the most common types of CSPRNGs used are stream ciphers, block ciphers (with counter or output feedback mode running), combinations of PRNGs (where several primitive PRNGs are combined in an attempt to create a higher degree of randomness), PRNGs designed around especially hard mathematical assumptions (usually these provide a strong security proof, however they can be slow compared to the others and be impractical for many applications) and PRNGs that are designed to be cryptographically secure such as Yarrow's algorithm and Fortuna.

These common types could be classified under the following three classes.

- *Cryptographic Primitives*. Includes the PRNGs based upon cryptographic primitives, such as stream ciphers, block ciphers and cryptographic hashes. It is important to keep the initial values of these cryptographic primitives secret, otherwise all security will be lost. Some examples include a block cipher running in counter mode, a hash of a counter or a stream cipher combining a pseudo random stream with plaintext.
- *Hard mathematical problems*. Includes the PRNGs based upon difficult mathematical problems. Here we have examples like the Blum Blum Shub algorithm based on the quadratic residuosity problem [11]. The only known way to solve the problem is to factor the modulus, and integer factorization is generally regarded as hard mathematical problem capable of providing security proof. Other examples include algorithms based on the discrete logarithm problem [12] or algorithms like Dual_EC_DRBG, which is based upon the assumed hardness of the decisional Diffie–Hellman (DDH) assumption [13].
- *Special designs*. Often introduce additional entropy if available, the output is not always determined by the initial state which can prevent attacks even if the initial state is compromised. There are a number of PRNGs specially designed to be cryptographically secure. Some examples include the Yarrow algorithm, the Fortuna algorithm and CryptGenRandom (from Microsoft's Cryptographic Application Programming Interface).

The difficulty of measuring security in a CSPRNG means that CSPRNGs often require years of review before certification.

3.3 CSPRNG requirements

CSPRNGs needs to fulfill the same requirements as an ordinary PRNG, which means that they will need to pass statistical randomness test, and in addition to this they need to stay secure even in the event where the initial state or the running state of the CSPRNG is revealed to the attacker.

A lot of PRNGs will be able to pass statistical randomness test, and have output that appears random. However many of them will not be able to pass more specialized test, such as determined reverse engineering tests checking for state compromise, showing that their output is not truly random. This means most PRNGs are not suitable as CSPRNGs.

Some CSPRNGs based on cryptographic primitives and hard mathematical problems are used, however all of these do not ensure protection against state compromise, but relies on keeping states and values hidden or secret. CSPRNGs that want to ensure security should be designed explicitly to withstand this sort of cryptanalysis.

3.3.1 Next-bit test

This means that a CSPRNG need to pass something called the "Next-bit test", passing this test will ensure that the PRNG will pass all other polynomial time statistical tests for randomness which was proved in 1982 [14].

The "Next-bit test" consist of, given the first k bits of a random sequence, the attacker will not be able to predict the (k+1)th bit in polynomial time. Passing this test will not ensure that the PRNG is safe for cryptographic use however, since it does not guarantee that it passes tests aimed at testing if it can withstand state compromise.

An example of this would be if we looked at a PRNG that produces output by computing bits of pi in sequence, starting from an unknown point in its binary expansion (the term in base-2 power). This could pass the "Next-bit test", but an attacker that determines which bit is currently in use (current state of algorithm) will be able to calculate all preceding bits as well. This leads us to the next type of test that needs to be performed to provide proof that a PRNG is cryptographically secure, the state compromise tests.

3.3.2 State compromise tests

The state compromise tests are applied to the special deigns class of CSPRNGs. The proof of security for the other classes (cryptographic primitives and hard mathematical problems) does not rely on passing this type of tests.

This type of tests deal with the case of an attacker compromising a single state of the PRNG, and then using the information gained from that to reconstruct past or future output. Usually performed on the initial seed state or some other vulnerable state where insufficient entropy has been used.

This means that it is an requirement for these type of tests to check, in the case that a state has been revealed or guessed correctly, that it is impossible to reconstruct the CSPRNG stream prior to the revelation. As well as check, in the case of entropy being revealed while running, that it is impossible to predict future values of the stream.

3.4 Dual_EC_DRBG

The U.S. Government releases a standard for DRBGs, where the most recent contained four different approved techniques for implementing this official standard for random number generators [15]. The four approved techniques is based on; hash functions, HMAC (Hash-based message authentication code), block ciphers and elliptic curves, and it is generally a good idea to use one of these few well-trusted cryptographic primitives.

However one of these four, the last called Dual_EC_DRBG for short (Dual Elliptic Curve Deterministic Random Bit Generator), is no longer considered secure. NSA has been involved in the making and breaking of a great deal of cryptography standards, so it is not weird that they have been a part of NISTs (the U.S. Commerce Department's National Institute of Standards and Technology) standard. But there are some additional circumstances that make Dual_EC_DRBG look a bit suspicious [16].

In 2006 a small bias in the random numbers produced were described, a cause for concern, however an optional workaround for this was mentioned in Appendix E of the NIST standard, of which the first draft was released in 2005 [17]. The problems concerning Dual_EC_DRBG has however continued to grow and in an informal presentation at the CRYPTO conference in 2007 [18] it was demonstrated that there exist a possibility of a backdoor.

In Appendix A of the NIST standard we are presented with a set of constants used to define the algorithm's elliptic curve, however the origin of these constants are never explained and what the informal presentation from the 2007 CRYPTO conference shows is that these numbers could have been created with a second set of secret numbers. This second set of secret numbers could work as a skeleton key, enabling anyone in possession of them to predict future output from the RNG after collecting only 32 bytes of output.

The only person that knows if there is a set of secret numbers created in tandem is the person who produced the constants, so it is not known if there exist any at all, but if it does they can be used to break any instantiation of Dual_EC_DRBG. And even if no secret numbers exist it makes Dual_EC_DRBG vulnerable, if one instance of the algorithm's elliptic curve problem were solved it would render every implementation insecure.

This whole problem can be avoided if the optional method of implementation, in Appendix A of the NIST standard, is used to generate new constants using another secure PRNG. However most implementations will probably not.

3.5 Debian Linux RNG bug

In 2008, May 13th, the Debian team reported a vulnerability in the OpenSSL package they distribute. The bug appeared when some lines of code which caused warnings about uninitialized data were removed, accidentally crippling the seeding process for the PRNG. A lot of the random data going into the seed was lost and the only remaining data consisted of the current process ID, limited to 32768, resulting in a very small number of seed values for the PRNG.

As a result of this cryptographic key material may be guessable, and affected keys include SSH keys, OpenVPN keys, DNSSEC keys, SSL/TLS keys, as well as all DSA (Digital Signature Algorithm) keys ever used on an affected Debian system, signing or authenticating, since these rely on a secret random value during signature generation [19].

The bug itself was introduced in September 2006, resulting in the bug propagating to both the testing and the current stable distribution of Debian (etch), the old stable distribution (sarge) however remained unaffected. Systems not based upon Debian should remain unaffected as well, since it is a Debian specific vulnerability, however it is possible that other systems will be indirectly affected if importing weak keys.

4. Discussion

PRNGs are used in a great deal of places when it comes to cryptography, both for short term and for long term applications. And something we can deduce from the examples in this report, is that flaws are easily introduced in PRNGs, both accidentally and purposely, and that these flaws usually are very hard to discover. This is partly because of the difficulty of testing if the output of a PRNG is truly random and partly because of the impossibility of proving that there is no state compromise vulnerability.

These problems are one reason why some vulnerabilities in PRNGs takes a long time to discover. For example in the case of Dual_EC_DRBG, where it took a couple of years from the first draft was made public until the first presentation was held discussion the possibility of a backdoor in the standard. These kinds of situations occur when it is difficult to prove the security of the CSPRNG and when that proof requires specialized tests.

Another thing to note is that a flaw affecting a PRNG often propagates, affecting the systems that are using the PRNG. As with the accidental removal of a couple of lines of code in the Debian OpenSSL package, causing the need for renewal of keys for several cryptographic systems using the implementation. Even if the problem itself was corrected fairly quickly within the package once discovered, the vulnerabilities and insecure keys it created down the chain takes significantly more time before they are solved or exchanged for secure keys, this is also directly affected by the amount of time the vulnerability has been in place.

5. Conclusion

PRNGs and CSPRNGs are pseudo random per definition and their purpose is to speed up creation of random numbers by “stretching” or “extending” the entropy from another source, often hardware in comparison with the PRNGs which are software based, meaning they are running on a deterministic machine. This makes it inherently hard to create a PRNG with high entropy and even harder to create a CSPRNG. For this reason it is also very hard to create tests that “prove” that a PRNG produces a high entropy sequence.

CSPRNGs are even harder to determine if they are suitable for their purpose since they in addition to the criteria for a PRNG also need to be able to withstand state compromise attacks, which is also hard to test and prove, often requiring quite advanced math and a lot of scrutiny. Because of this we conclude that CSPRNGs benefit from open implementations allowing for a more in depth analysis of the CSPRNG to facilitate the discovery of potential vulnerabilities, also avoiding the possibility of intentional nefariously placed vulnerabilities.

If a vulnerability exists in a CSPRNG, making it a weak CSPRNG, this can easily propagate causing a lot of problems, since CSPRNGs are widely used (especially standardized CSPRNGs). Accidental and purposeful weaknesses are easily introduced, and often affect a large amount of users.

References

[1] “Pseudorandom number generator”, [http://en.wikipedia.org/wiki/Pseudorandom_number_generator], [Accessed: 2014-05-14]

[2] John von Neumann, 1949, “Various techniques used in connection with random digits”, [http://www-apr.lip6.fr/~lumbroso/References/VonNeumann51.pdf]

[3] “Entropy”, [http://en.wikipedia.org/wiki/Entropy_(computing)], [Accessed: 2014-05-02]

[4] “Entropy”, [http://www.merriam-webster.com/dictionary/entropy], [Accessed: 2014-05-12]

[5] “Linear congruential generator”, [http://en.wikipedia.org/wiki/Linear_congruential_generator], [Accessed: 2014-05-15]

[6] “Lagged Fibonacci generator”, [http://en.wikipedia.org/wiki/Lagged_Fibonacci_generator], [Accessed: 2014-05-15]

[7] “Linear feedback shift register”, [http://en.wikipedia.org/wiki/Linear_feedback_shift_register#Uses_in_cryptography], [Accessed: 2014-05-15]

[8] M.G. Kendall and B. Babington Smith, 1938, "Randomness and Random Sampling Numbers", Journal of the Royal Statistical Society

[9] “The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness”, [http://www.stat.fsu.edu/pub/diehard/], [Accessed: 2014-05-14]

[10] “Mersenne prime”, [http://en.wikipedia.org/wiki/Mersenne_prime], [Accessed: 2014-05-12]

[11] “Quadratic residuosity problem”, [http://en.wikipedia.org/wiki/Quadratic_residuosity_problem], [Accessed: 2014-05-15]

[12] “Discrete logarithm”, [http://en.wikipedia.org/wiki/Discrete_logarithm_problem], [Accessed: 2014-05-15]

[13] “Decisional Diffie–Hellman assumption”, [http://en.wikipedia.org/wiki/Decisional_Diffie%E2%80%93Hellman_assumption], [Accessed: 2014-05-15]

[14] “Next-bit test”, [http://en.wikipedia.org/wiki/Next-bit_test], [Accessed: 2014-05-02]

[15] Elaine Barker and John Kelsey, January 2012, “Recommendation for Random Number Generation Using Deterministic Random Bit Generators”, [http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf], [Accessed: 2014-04-26]

[16] Bruce Schneier, 2007-11-15, “The Strange Story of Dual_EC_DRBG”, [https://www.schneier.com/blog/archives/2007/11/the_strange_sto.html], [Accessed: 2014-05-06]

[17] Matthew Green, 2013-11-18, “The Many Flaws of Dual_EC_DRBG”,

[http://blog.cryptographyengineering.com/2013/09/the-many-flaws-of-dualecdrbg.html], [Accessed: 2014-05-04]

[18] Dan Shumow and Niels Ferguson, August 2007, "On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng", In: Rump session, CRYPTO, [http://rump2007.cr.yp.to/15-shumow.pdf], [Accessed: 2014-04-28]

[19] "DSA-1571-1 openssl -- predictable random number generator", [http://www.debian.org/security/2008/dsa-1571], [Accessed: 2014-05-04]