Exploitation of Random Generators and Secure Implementation

Samuel [redacted]
Matriculation number: [redacted]
[redacted]@student.dhbw-mannheim.de

Abstract

This paper is intended to provide a very basic understanding of how a random number generator (RNG) works. We discuss the different applications of the most common types of RNGs. It is an important decision which RNG we should use in a particular environment. When does it make sense to use a common software based RNGs and when should special hardware be used? We examine various vulnerabilities of the RNGs and demonstrate them with practical examples. Finally, we work out from the previous attacks how such attacks can be defended against and how RNGs are best implemented.

I. Motivation

This topic is particularly important because the security of random numbers is often forgotten. In terms of security vulnerabilities, it is rare to hear about vulnerabilities in RNGs. Although these are already configured securely by default in software terms, any random number can be predicted by an incorrect implementation. Assuming, for example, that this is used to generate SSH keys for employees of a company, an attacker can reconstruct these keys and gain full access to compromise the system. The associated risk is simply underestimated by many. Therefore, in this paper we will examine some threats as well as a correct implementation.

II. Introduction

The basic function of a conventional Pseudo Random Number Generators (PRNGs) is to generate a sequence of number based on a given seed. If the seed is the same, it outputs the same sequence of numbers every time. We can predict every output from the generator if we know the initial seed and used algorithm. These algorithms are not considered truly random because the values generated depend on the seed. It is also called a deterministic random number generator because its output appears random from the outside, but from the inside it is just a repetitive calculation based on the seed.

In untrusted areas like client-side computations, this does not matter. If we choose a different seed each time and the implementation if the algorithm is correct, we have a sufficient randomness and can profit from very fast generation of random numbers.

In applications such as gambling, cryptography or special scientific fields, conventional arithmetic methods reach their limits, as kindly mentioned by the mathematician John van Neumann: *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."* [1].

True random number generators (TRNGs) or hardware random number generators (HRNGs), on the other hand, can generate random output without a seed. Whereas a conventional PRNG is built upon software, a HRNG uses electronic circuits and physical phenomena as described later. A Pi 3 Model B+, for example, can be purchased for about $35 and has a generator built in. External HRNGs can be purchased for between $50 and $1000. They can be plugged into a computer via USB or PCIe. They can then be very easily integrated into the system.

III. Background

Principle of a typical random number generator

The basic principle is the same for all RNGs.

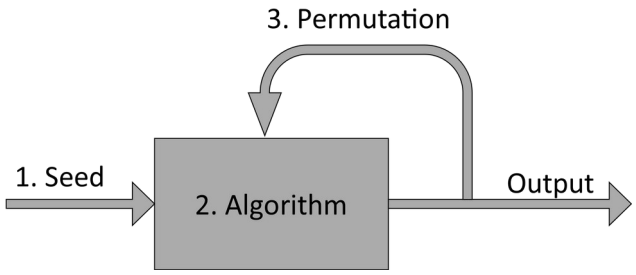

Figure 1. Basic principle of RNG

The generator needs a seed as input. The randomness of the sequence is then exclusively dependent on the seed. As can be seen in Fig. 1, the seed serves as the starting value. Depending on the algorithm, a new value is now calculated with the input. This number is then used again as input for the next generation. This process repeats itself and the calculation always remains the same.

To prevent the output from inferring the next numbers, only part of the result of the output is returned at a time. Thus, information is missing to predict the output. We explain this with a very simple example called the middle-square method suggested by John von Neuman.

We take the number 121 as seed. We square this number and get 14,641 as a result. Now we take the middle 3 numbers and get 464. The output 464 represents our first random number. The seed cannot be reconstructed because this number could also have been created from seeds 157 or 738. The next step would be to square 464 again and take the middle 3 numbers. If we repeat this process, we get these random numbers: 121 (seed), 464, 529, 984, 825 ...

However, this method is not suitable for real applications. Either the number repeats after a given period or the number approaches zero. Therefore, modern random

number generators of e.g., Python, PHP or Ruby are based on the Mersenne Twister algorithm, which is more complex, but can generate much more numbers without repeating or approaching zero. However, these algorithms should also not be used for cryptographic purposes, as explicitly pointed out in the documentation of PHP, for example [2]. In the exploitation section, we therefore show the weaknesses of such algorithms.

Entropy

To make statistical statements about the quality of a random number generator, we need a measured value. With the entropy, we can tell how predictable our randomness is. If it is high, it is hard to find any patterns or frequencies. The opposite is true for low entropy. For example, a message written in English has a low entropy, because letters like ‘e’ or ‘t’ are occurring more often that others like ‘q’ or ‘z’. To illustrate this, let us take any sentence. From this, we remove all punctuation spaces and make it lowercase: *“anyonewhoconsidersarithmeticalmethodsofproductingrandomdigitsisofcourseinastateofsin”*.

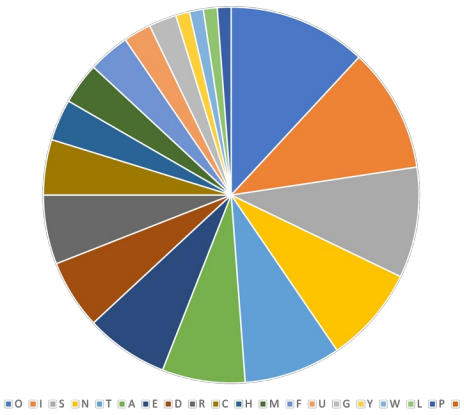

Figure 2. Frequency of letters.

If we now count and graph the number of each letter, we can directly see the distribution of the most frequent and rarest letters as seen in Fig. 2.

To determine the entropy of this set, we use Shannon entropy. The entropy H is defined as follows:

$$H(X) = -\sum_{i=1}^{n} P(x_i) * \log_2 P(x_i)$$

Where X is a set of all occurring values and n is the total number of values. $P(x_i)$ is the frequency of the i value. We calculate the entropy of the above theorem with the formula, using the original quote with upper- and lower-case letters. As a result, we get $H \approx 4{,}09$. Since we can't do much with this number, we need to normalize it by dividing $\log_2 P(x_i)$ by the largest possible value $\log_2 P(x_{max})$. Since we work with ASCII characters and the characters are represented by numbers, the highest value is 255. Then we get a number between 0 and 1, where 1 is the best possible entropy. A high entropy is always better, because less information can be drawn from a text. In our case we get this time $H \approx 0.51$.

To improve this entropy, we could use a hashing algorithm on our text. After hashing our sentence with HMAC-SHA512, we get an entropy of $H \approx 0.72$. Since SHA-512 is a proven cryptographic hash algorithm and cannot be reconstructed, it is often used in RNGs to increase Entropy.

Testing Entropy

The quality of the entropy is hardly visible to the naked eye. Even if one calculates the Shannon entropy, for example, fundamental problems with the algorithm can occur. Therefore, various software programs have been developed to test the algorithm for different cases.

The Statistical Test Suite (STS) developed by NIST consists of 15 different tests. To use it, it must first be compiled and can be run on both Linux and Windows. It can be used for both PRNGs and TRNGs outputs. If the generator has flaws, some other tests are likely to fail too [3].

Another test suite is DIEHARD, which was published in 1995 by George Marsaglia. It contains 16 different tests and can be used on Linux based systems and Windows. Many tests are like those from NIST's STS [4]. Here are some interesting tests included in DIEHARD test suite:

"The squeeze test" multiplies 2 to the power of 31 by a decimal number between 0 and 1 until the result is 1 and counts the number of trials. If this calculation is performed 100,000 times, the respective number of attempts is examined.

"Monkey tests" interprets a series of random numbers as words. Then it counts how many times each word occurs. Finally, the number of words that do not occur is examined.

"The craps test" simulates the dice game "craps" 200,000 times. For each game it is counted how often the dice were rolled and won. These numbers are then examined.

Some tests take longer than others or are more indicative of quality. So, it is a good idea to start with a simple test like the STSs "frequency test". If this test is passed, all other tests should still be performed, as each test examines different properties. If the first test fails, the probability is very high that the remaining tests also fail.

Entropy Source

The fundamental component of a TRNG is the entropy source. This is the main difference to conventional PRNGs. A recommended model from NIST is shown in Fig. 3 and consists of three interrelated components [5].

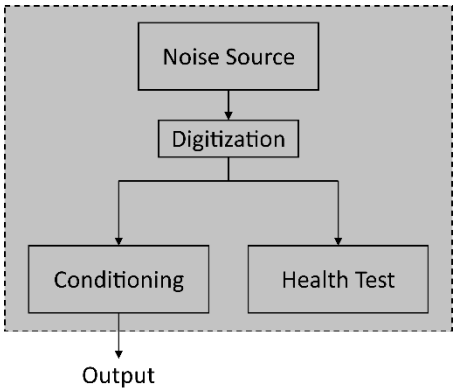

Figure 3. Model of entropy source

The most important component is the noise source. It must be non-deterministic and usually consists of an electrical circuit. The randomness is generated by physical states. This can be, for example, a ring oscillator, which generates noise through its electronic components [6]. This analog noise is then converted into digital signals, which can then be interpreted as binary digits.
At this point, our source of randomness is unpredictable (if implemented correctly) but can be biased. We can optionally implement a conditioning component that can reduce our bias. This can be achieved by distributing the input entropy. This is only an additional component and cannot compensate any lack of security by the noise source.
The last component performs health tests on the digitized noise source to protect from uncaught failures of the entropy. This is very important, because the system is still functioning and there are nearly no indications for this behavior for the human eye. It is highly recommended performing these tests on startup and continuous.

Conditioning
A conditioning function can be used to distribute entropy over the entire output. It is the last element after the noise source, as shown in Fig. 3. For example, a hash function, encryption, or compression can be used for conditioning. It is important to use an approved algorithm. However, this procedure cannot compensate for incorrect implementation or insufficient entropy. The algorithms for conditioning must be able to work with large input and output sets and be based on well-known fundamentals. Suitable keyless algorithms for conditioning are, for example, SHA-3 hash functions or algorithms with keys such as HMAC, CMAC or CBC-MAC [7].

Ring Oscillators
A common implementation in TRNGs are ring oscillators (ROs). These have many different designs with their advantages and disadvantages. A simple approach as shown in Fig. 4 is an independent oscillator, which generates a phase. This phase is not periodic but contains deviations and noise. The occurrence of deviations of signals is called jitter. This is a problem in transmission technology because transmitted signals can contain errors as a result. This phenomenon is used here to generate randomness. The output is then fed to an electronic component together with a slower clock. This can be an XOR gate or a D-flipflop, for example. This component is then referred to as the noise source as shown in Fig. 3.

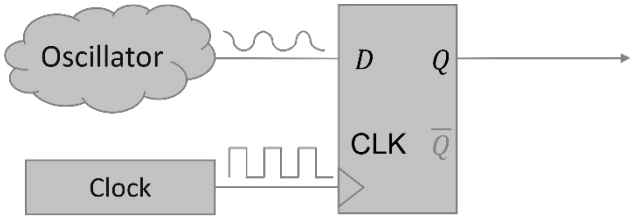

Figure 4. Basic Concept of Ring Oscillator Circuit

However, this scheme brings out some problems. First, synchronous sampling of both oscillators requires a specially designed circuit, which is very complex. Secondly, the accumulation time of the jitter must be waited for because the entropy is much lower than required. This reduces the speed significantly and makes it ineffective for frequent use [8].
An alternative design proposed in a paper by Samsung Electronics is a TRNG based on metastability rather than jitter [8]. In digital systems, the state of a circuit moves between a 1 and a 0. However, an intermediate state can occur in that the system remains in a metastable state and is neither 1 nor 0. This behavior is unpredictable and similar to the jitter phenomenon, can lead to errors in digital technology [9].
Such metastability can be perfectly achieved with a CMOS (complementary metal-oxide-semiconductor) inverter as seen in Fig. 5.

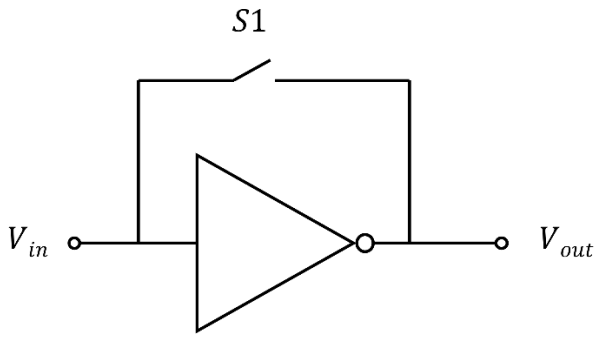

Figure 5. CMOS inverter

The inverter is switched in a loop. When the loop is closed by the switch, the output converges to metastability level. The output can be used as a noise source.

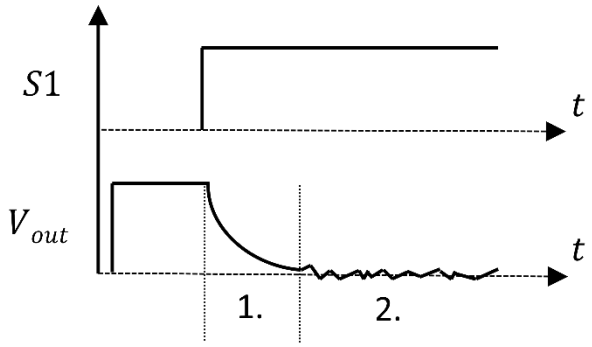

Figure 6. Timing diagram of CMOD inverter (1. convergence time 2. metastable state)

The procedure for transition to metastability can be seen in Fig. 6. Since the output can be used much earlier here, this method is a great advantage over the jitter accumulation time.

IV. Exploitation

Prediction of basic arithmetic algorithm

As mentioned, conventional arithmetic PRNGs are predictable if the initial seed and algorithm is known.

Built-in algorithms of many programming languages tend to initialize their random class with the current time and sometimes together with the memory address of a variable. To illustrate this with an example, we have a look at the math class of the very basic object-oriented language Lua. The source code can be obtained on their official webpage [10].

```
static void randseed (lua_State *L, RanState *state)
{
  lua_Unsigned seed1 = (lua_Unsigned)time(NULL);
  lua_Unsigned seed2 = (lua_Unsigned)(size_t)L;
  setseed(L, state->s, seed1, seed2);
}
```

Figure 7. Lua source code for seed generation

To make their input seed random, they use the current Unix time and a random address in memory as seen in Fig. 7. The Unix time is the number of seconds since 1970 and is represented by a 32-bit number. The address in memory is random, because modern systems use address space layout randomization (ASLR) which is used to randomize memory addresses to lower the chance of an attacker guessing a position in memory [11]. If the computer running the algorithm does not use ASLR, the security of the algorithm would decrease significantly, since each component of the seed can be reconstructed.

Guessing seed

Let's look at another algorithm from the statically typed general-purpose language C# by Microsoft. Its random library relies on a similar approach but is harder to predict. The source code (for the .NET Framework) can be obtained at the reference sources from Microsoft [12].

Mostly, we initialize the random class without having to specify a seed. If this is the case, the class uses an algorithm based on Numerical Recipes in C [13] to generate a seed when the class is initialized. The seed is always a 32-bit number. This means that there are 2.147.483.647 possible seeds. Suppose we have a machine that gives us 20 random numbers between 1 and 10. Theoretically, we could take those 20 random numbers and try to generate the same result with each of the 2 billion seeds. With today's computing power, that's not much.
To verify that this is possible in the real world, we need to assume two things. First, the type of numbers generated must always be the same (data type and range of the output). Second, we need a sequence of numbers generated one after the other and occurring at the beginning if possible.

The proof-of-concept code that was used can be found at my GitHub [14]. Firstly, we generate 20 random numbers between 1 and 10. These 20 numbers represent the random output of the system we want to attack. Our goal is to be able to predict all following random numbers of the system. We start by iterating over all possible seeds. With each seed we generate 20 numbers and check if they are the same as the ones from the victim system. Once they match, we have found the seed. This process can take a long time. Therefore, we divide the seeds on different threads, which run in parallel. This shortens the computation time considerably. It is important to verify that this is the right seed, as we could have generated 20 same numbers by coincidence.
When running this application on a 40-core system (Intel Xeon Silver 4210R (40) @ 2.899GHz), we can predict the seed in about 5 minutes. On a desktop environment with 6 cores (Intel Core i7-4790 CPU @ 3.60GHz), this process can take up to 15 minutes. We can confirm that under the given conditions, an attack would be quite feasible. For large seeds of, say, 64 bits, the attacker would additionally need to know the approximate range of the seed used, since the computation takes about 4 billion times longer than for 32-bit seeds. As the complexity of the system increases, an attack also becomes more and more expensive. For example, if numbers are generated continuously and the attacker only knows one or more distributed sequences of them, the attack can no longer be carried out reliably.

Vulnerabilities in Software

Insufficient seeding was also a problem with widely distributed software versions. In May 2008, a vulnerability was discovered in Debian's OpenSSL package and assigned CVE-2008-0166 [15]. All 65,536 possible SSH keys generated with this vulnerable version were predictable. Originally, a mixture of addresses from memory and the PID (process ID) was used for the seed. However, due to a bug, this implementation was limited to the PID only. This meant that an attacker could simply brute force all possible combinations and try to authenticate on an SSH server with them. Eventually, all keys that were considered affected were blacklisted by a

security patch and had to be regenerated. This exploit was given a severity rating of 8/10 (high).

Attacks on TRNGs

Ring oscillators are often used for TRNGs as described earlier. Since these random numbers originate in anomalous behaviors, it is impossible to make predictions with a proper implementation. Therefore, one attack vector in TRNGs based on jitter accumulation is external influence on the system. In this case, foreign frequencies are injected into the system to influence the phase behavior. Most often, this frequency is injected through the device's power supply. This allows the phase difference between the oscillators to be brought to 0 and locked there. By this method the output of a TRNG can be influenced [16].

A practical application for this attack is, for example, a bank's smart card. The affected smartcards are older models but shows very well the procedure of such an attack. First, the operating frequency of the card under power must be determined. Then, a frequency is selected for injection, which should be approximately between the smallest and largest measured value. While this frequency is injected with different voltages, statistical tests are performed. These usually turn out very poorly with optimal settings of voltage and frequency. Thus, the effectiveness of the built-in TRNG can be significantly influenced [16].

V. Secure Implementation

Implementation of PRNGs

The security of a PRNG depends on the length and indistinguishability of the seed. First, the length of the seed determines how many sequences of random numbers can be generated. As we have already seen, we could guess the one seed used from 2 billion others. If we had used a 64-bit seed instead, we would have had nine quintillion possible seeds. In comparison to a 32-bit seed, this would be about 4 billion times more seeds. Secondly, we need an indistinguishable seed. Depending on the use case, this can either be a combination of the current time and a random memory address, as shown previously, or a TRNG could be used just for the seed.

If we were to use an RNG for a very fast, but also secure purpose, we should consider using a long seed (e.g., 64-bit) in combination with a fast arithmetic PRNG. The seed should then be generated by a slower TRNG, as it's just needed one time [17]. This prevents the seed from being reconstructed, since it does not depend on addresses or times of the system. It also cannot be guessed using a brute force method, since the number space is clearly too large.

Implementation of TRNGs

A TRNG must be subjected to statistical tests before it is put into operation. These can additionally rule out incorrect implementation or a defect in the system. A TRNG should be implemented when the numbers generated cannot be predicted under any circumstances and speed is not a major concern.

Since TRNGs are used especially at security-critical levels, they are a lucrative target for attackers and are therefore particularly worth protecting.

To operate a hardware random number generator (HRNG) securely, it is first and foremost important to be aware of applicable standards and recommendations.

Attacks on ring oscillators can be prevented in different ways. If the system is physically accessible to an attacker, it should be shielded. If the power connection is open, electronic protective components must be processed, which smooth and filter the input. The goal is to allow an attacker to gain as little information as possible. Likewise, the attacker should not influence the behavior of the system. Thus, in the worst case, the system can block all communication during an attack [16].

Tests on TRNGs

A TRNG is significantly more complex and safety-relevant than a PRNG. For example, with a built-in random number generator of a programming language, hardly anything needs to be taken care of and can be used directly. As mentioned in the Background section, testing is strongly recommended for a TRNG. According to the NIST recommendation, a health test should detect when there is a decrease in the quality of the entropy, failure of the noise source, or other damage to the hardware. These incidents should not be underestimated and should be permanently controlled.

In addition, there are 3 types of tests. First, start-up health tests ensure that the generator is running as expected even before its output is used. Second, continuous health testing should be conducted during operation. Even if the tests were successful at startup, errors can still occur during runtime. Even if it is very unlikely in a correctly implemented system, these tests should be automated. Third, health tests can be performed on demand. However, this does not necessarily have to happen with productive systems [7].

VI. Conclusion

In this paper, we have focused particularly on the basic principles of RNGs. These are particularly important to understand why a proper implementation is so important. A random number generator should only be used for its intended purpose. For example, the Lua random number generator used in the previous example should not be used to generate key pairs. Also, HRNG should not be used for fast real-time applications, as it is too slow and expensive. We have learned about the Shannon method, which allows us to determine the entropy of a text. To understand conditioning, we used HMAC-SHA512 to hash the text, thereby distributing the entropy better. For a better understanding of what factors the security of a PRNG depends on, we studied the implementation of Lua and C#. Using a practical example by guessing the seed using brute

force, we were able to determine that the seed is the largest attack surface for an attacker. Through the vulnerability in Debian's OpenSSL library, we were able to see that these attacks also have a foothold in practice and can have severe security-critical consequences.

To understand an attack on a TRNG, we looked at the entropy source model and the function of a simple ring oscillator. Using the smart card example, we saw that the output can be altered by external influences.

With this knowledge, some solutions emerged directly. With a PRNG the use of a large seed, which cannot be reconstructed. Since a TRNG is in the form of hardware, it should also be physically protected.

Both types of generators should be subjected to various tests as best as possible, as we have also shown.

Finally, we have collected the recommendations for the correct implementation and use of suitable and recognized algorithms. These were not all-inclusive, but were intended to cover, in particular, the attacks we conducted. Before implementing an RNG, one should be aware of the applicable standards.

All the aspects we have worked out in this paper should give a good overview of possible threats or wrong implementations to know about.

References

[1] J. v. Neumann, "Various techniques used in connection with random digits" in *Monte Carlo Method*, Washington, DC, US Government Printing Office, 1951, p. 36.

[2] Documentation of PHP random library https://www.php.net/manual/en/function.rand.php

[3] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray and S. Vo. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22 Revision 1a. *National Institute of Standards and Technology*, 2010. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf. Accessed 07.01.2022.

[4] The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. *Florida State University*, 1995. https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/. Accessed 07.01.2022.

[5] E. Barker and J. Kelsey. NIST DRAFT Special Publication 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation. *National Institute of Standards and Technology*, 2016. https://csrc.nist.gov/csrc/media/publications/sp/800-90b/draft/documents/draft-sp800-90b.pdf. Accessed 27 December 2021.

[6] N. Hashim, J. Loong, A. Ghazali and F. Hamid. Memristor based ring oscillators true random number generator with different window functions for applications in cryptography. *Indonesian Journal of Electrical Engineering and Computer Science*, 2019. http://ijeecs.iaescore.com/index.php/IJEECS/article/view/16771. Accessed 29 December 2021.

[7] M. Turan, E. Barker, K. Kelsey, K. McKay, M. Baish and M. Boyle. Recommendation for the Entropy Sources Used for Random Bit Generation. NIST Special Publication 800-90B. *National Institute of Standards and Technology*, 2018. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf. Accessed 07.01.2022.

[8] I. Vasyltsov, E. Hambardzumyan, Y. Kim and B. Karpinskyy. Fast Digital TRNG Based on Metastable Ring Oscillator. *Samsung Electronics, SoC R&D Center, System LSI, Korea*, 2008. https://link.springer.com/content/pdf/10.1007%2F978-3-540-85053-3_11.pdf. Accessed 04.01.2022.

[9] T. Chaney and C. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, pages 421–422, 1973.

[10] Official webpage of Lua with source code https://www.lua.org/source/

[11] C. Ruoho. ASLR: Leopard Versus Vista. *Laconic Security*, 2008. https://web.archive.org/web/20100222214224/http://www.laconicsecurity.com/aslr-leopard-versus-vista.html. Accessed 29 December 2021.

[12] Source code of the .NET Frameworks random library https://referencesource.microsoft.com/#mscorlib/system/random.cs

[13] W. Press, S. Teukolsky, W. Vetterling and B. Flannery. Numerical Recipes in C. The Art of Scientific Computing Second Edition. *CAMBRIDGE UNIVERSITY PRESS*, page 307, 1992 https://www.cec.uchile.cl/cinetica/pcordero/MC_libros/NumericalRecipesinC.pdf. Accessed 10.01.2022.

[14] Proof of Concept code for brute forcing the seed of the random class in C# https://github.com/Freilichtbuehne/RandomSeedGuesser

[15] F. Weimer. New openssl packages fix predictable random number generator. *Security announcements*, 2008. https://lists.debian.org/debian-security-announce/2008/msg00152.html. Accessed 04.01.2022.

[16] A. Markettos and S. Moore. The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators. *Computer Laboratory, University of Cambridge, UK*, 2009. https://link.springer.com/content/pdf/10.1007%2F978-3-642-04138-9_23.pdf. Accessed 05.01.2022.

[17] L. Dongfang, L. Zhaojun, Z. Xuecheng and L. Zhenglin. PUFKEY: A High-Security and High-Throughput Hardware True Random Number Generator for Sensor Networks. *School of Optical and Electronic Information, Huazhong University of Science and Technology*, 2015. https://www.mdpi.com/1424-8220/15/10/26251/htm. Accessed 07.01.2022.