Potential Weaknesses In Pseudorandom Number Generators

Andrew Li
andrew.li@tufts.edu
Instructor: Ming Chow

Section 1

Abstract

Recent concerns have arisen over the security of pseudorandom number generators (PRNGs). Cryptographically secure PRNGs (CSPRNGs) are verified mathematically to produce a sufficiently random series of numbers. However, some PRNGs previously thought to be secure have suffered from a range of issues, including improper implementation and backdoors. These flaws become apparent when PRNGs are depended upon for cryptography or encryption. Without a sequence of sufficiently random numbers, an attacker can potentially subvert the software security. In this paper, we will explore the problems found in PRNGs and highlight recent examples of vulnerabilities and consequences. We will also demonstrate how an attacker might take advantage of such weaknesses.

Pseudorandom Number Generators

Simulating true randomness in a computing environment is a surprisingly difficult task. Computers cannot operate on chance. They require a set of instructions to execute and complete tasks. In order to mimic randomness, computers use PRNGs to choose numbers that appear to be picked at random. Each algorithm is given a seed, or starting point. From that seed, a mathematical formula generates a predetermined list of numbers (Random.org). This series of numbers is predetermined by the initial state, making the PRNG a deterministic system which simulates randomness.

Random number generators are commonly used in encryption and other security applications. Cryptographically secure PRNGs are referred to as CSPRNGs. These are verified by researchers in several aspects, including the robustness of their entropy source and their

resilience to predictions in both forward and backward security (Dodis et al.). An entropy source injects randomness into the system, and is typically taken from hardware or peripherals such as the mouse and keyboard. Forward security implies that if the system fails in the future, past keys are not compromised while in backward security, past keys do not compromise ones generated in the future. What results is a number generator which produces seemingly random numbers and that an attacker can break only break temporarily.

The Bad Guys

When a CSPRNG is compromised, the attacker gains some insight into defeating the randomness of the number generator. Since PRNGs are deterministic, it is comparable to knowing the poker hands before they are dealt. Due to the vast cryptographic applications of CSPRNGs, many different parties may be working against their security. Some might target the keys generated by CSPRNGs and used in symmetric-key algorithms, while still others may attempt to break passwords, nonces, or one-time pads (Dodis et al.). In every case, the bad guy is ultimately after the encrypted communication. An intercepted message is useless, but with inside knowledge of the random number generator, an attacker can potentially decrypt the information and steal vital data.

To the Community

Why do we need to care about the security of CSPRNGs? CSPRNGs are the foundation of security. When they are cracked, the security of every service which depends on it is compromised. This could lead to a loss of private information, including passwords, encrypted conversations, and personal records and information. It is far too easy to take CSPRNGs for

granted. Unlike other software vulnerabilities, weaknesses in the number generator are more difficult to detect without spending the time to observe possible patterns after many trials. Thus, we must avoid taking the security of CSPRNGs for granted.e

Section 2

Breaking the PRNG

The source of CSPRNG weaknesses vary between each vulnerability. Some arise due to the improper implementation of algorithms and bugs in the code. Others are flawed from the start and contain intentionally built in weaknesses for an outside party to exploit. Even without these implementation issues, CSPRNGs can be lacking enough entropy to generate sufficiently random numbers. These three cases have all appeared at one time in real world applications. We will examine each case separately in this paper.

Cryptocat

In June of 2013, a user discovered a bug in the algorithm used to generate random numbers for the open-source encrypted web chat client Cryptocat. The service promised users a secure system through which online chatting would be encrypted using Off-the-Record Messaging (OTR) (Faucon & Kobeissi). OTR uses AES symmetric-key encryption and the Diffie-Helman exchange to share keys. In symmetric-key encryption, the same key is used in both encryption and decryption of a file. The key is shared secret between two parties. Since it must be kept secret in order to maintain its integrity, Cryptocat uses the Diffie-Helman key exchange. In such an exchange, both parties begin with the same key. Each adds their own secret to the key, generating two different values. The new keys are then swapped, and each

receiver adds their secret to the other party's key. The result is a common key which cannot be obtained by an eavesdropper. Furthermore, all Cryptocat messages were encrypted client-side, and the server which handled communications between two users only received encrypted messages.

The Cryptocat software used a PRNG created by its developers, and contained the following bug:

```
var x, o = '';
while (o.length < 16) {
    x = state.getBytes(1);
    if (x[0] <= 250) {
        o += x[0] % 10;
    }
}
```

Since Cryptocat uses JavaScript, the PRNG needed to return a number which fit within the 8-byte IEEE-754 floating point numbers (Ducklin). Integer precision in IEEE-754, however, only extended to 53 bits. To insert as much randomness as possibles, the programmers of the PRNG built the number using a text string starting with the character '0'. Sixteen ASCII digits were then added onto the string, which was then converted into an 8-byte float. The numbers were selected using an algorithm known as Salsa20, or a stream cipher which combines plaintext digits with pseudorandom digits using XOR. Salsa20 generates a random byte which is then converted to a character between 0 and 9. Since the byte could take 256 different variations and could only be mapped to ten different characters, the programmers then restricted the outcome by throwing out the 6 extra variations and accepting the first 250 values, a number divisible by 10.

However, the implementation contained an off-by-one error. The comparison *x[0] <= 250* allows for 251 possible values from 0 to 250 inclusive. The result of this innocuous bug

produced a set of numbers that favored a certain outcome, in other words, the spread of results was not sufficiently random. With an extra value available, the character '0' was seen 10% more than expected. As a result, nearly a year's worth of messages were easier to crack.

Dual_EC_DRGB

The Dual Elliptic Curve Deterministic Random Bit Generator, or Dual_EC_DRGB, is a CSPRNG which uses elliptical curve cryptography to generate random bits. It was **approved** by the National Institute of Standards and Technology (NIST) as a cryptographically secure number generator (United States: NIST).

Shortly after its release in 2006, researchers Dan Shumow and Niels Ferguson suggested that the Dual_EC_DRGB algorithm contained a possible backdoor inserted by the National Security Agency (NSA) (Green). The algorithm relied on finding a point P on an elliptical curve called a generator using a secret constant Q (Shumow & Ferguson). There also exists some value e by which $Q^e=P$. However, Shumow and Ferguson found that an attacker could compute the initial state of the CSPRNG if they found some value d such that $P^d=Q$. With Q and P both known, e could then be calculated, giving the attacker access to the initial state as well as the list of future values that the CSPRNG would generate.

After Shumow and Ferguson revealed the vulnerability, the NIST retracted its approval for the algorithm and warned developers to avoid any usage of Dual_EC_DRGB. Recently, the backdoor was confirmed to be inserted by the NSA with Edward Snowden's release of NSA classified documents in 2013 (United States: NIST). With this disclosure, the trust in several

well known PRNGs utilizing this algorithm was lost, notably RSA BSAFE libraries used in both Java and C/C++ (Green).

/dev/random

Recently, Dodis, Pointcheval, Ruhault, Vergnaud, and Wichs discovered that the Linux standard PRNGs lacked the entropy to produce truly random results in some cases. Their research focused on the robustness, or randomness, of /dev/random and /dev/urandom. These PRNGs collect randomness, or entropy, from keyboard presses, mouse movements, and other system processes (Dodis et al., 2013). However, the research found that entropy at certain points was especially low, such as the period immediately following start-up. At these points, it is possible that the internal state of the PRNG could be compromised. Due to the collection of entropy, the PRNG should recover from such a compromise when enough entropy is added into the algorithm.

When Dodis et al. examined /dev/random and /dev/urandom, they found that the PRNGs did not recover sufficiently as they were not accumulating enough entropy due to interactions with the built-in entropy estimator and mixing function. The entropy estimator is used to estimate the entropy of the input source while the mixing function refreshes the state of the PRNG after each input (Dodis et al., 2013). However, in cases of high input, Dodis et al. found that the estimator estimated very high entropy while in cases of arbitrarily high entropy, the estimator mistakenly estimated near zero entropy. With this information, an attacker who has compromised the PRNG might construct a situation with one of these two conditions and compare the results of the PRNG to the ideal conditions of zero or arbitrarily high entropy

estimated. Although the researchers admit that no known exploits from these weaknesses have been documented, they recommend a newer PRNG which avoids utilizing an entropy estimator.

Section 3
Prevention

Prevention of CSPRNG failure is difficult due to differing reasons for their weaknesses. In every case, there is a level of trust that must be maintained between the user and underlying algorithm. The implementation of each CSPRNG must be sound. In most cases, this can be guaranteed to be true as CSPRNGs must conform to certain standards before they are approved for general usage. Yet, the Cryptocat incident shows that not every CSPRNG is perfect. Even properly implemented number generators may have intentional vulnerabilities that are waiting to be exploited by outside parties, such as the Dual_EC_DRGB. Sufficient entropy collection is harder still to detect. Prevention of CSPRNG weaknesses becomes a continuous task of analysis and verification.

Though each vulnerability stemmed from different sources, all were found with a combination of research and analysis. Biased CSPRNGs are easily identified when examining a graph of their outputs and identifying results that are more heavily weighted than others. Other vulnerabilities require more study to find. When new CSPRNGs are proposed, they must first be approved and then verified by study. Thus, well-established CSPRNGs are generally reliable to use.

The best practice for current developers is to use approved CSPRNGs, verified by both the NIST and security researchers. Java's SecureRandom is one such example. It is also safer to

use a prewritten CSPRNG rather than implement a custom number generator. There are also hardware random number generatorsA final option may be avoid deterministic PRNGs altogether and use True Random Number Generators, or TRNGs. These number generators choose numbers based on external variability, including radioactive decay, atmospheric noise, or even snapshots of lava lamps (Random.org). TRNGs offer a nondeterministic sequence of numbers which will never repeat at the cost of difficult setup and inefficiency. The best choice for number generators is based on the application.

Conclusion

CSPRNGs are a vital component in security. They are used by programmers in all fields building secure systems to create cryptographic keys, encrypt communications, and defend anonymity. Yet, many are based on deterministic algorithms which, if broken, can reveal all numbers that will be generated to the attacker. While many of these number generators have not been compromised, there are some that have had flaws undermining their security. From improper implementation, backdoors, and entropy collection, the difficulties facing CSPRNGs threaten to crack the foundation of cryptography. It is important to be vigilant. One should never take randomness for granted when working on a secure system. By verifying the cryptographic standards of CSPRNGs, we can maintain the integrity of random number generation in future applications.

References:

Dodis, Yevgeniy, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs. "Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust." Cryptology ePrint. (2013). Web. 12 Dec. 2013. <http://eprint.iacr.org/2013/338.pdf>.

Ducklin, Paul. "Anatomy of a pseudorandom number generator - visualising Cryptocat's buggy PRNG." Sophos: Naked Security. (2013). Web. 13 Dec. 2013.

Faucon, Daniel, and Nadim Kobeissi. "OTR Encrypted File Transfer Specification." . Cryptocat, 27 Nov 2013. Web. 13 Dec 2013.

Green, Matthew. "The Many Flaws of Dual_EC_DRGB." A Few Thoughts on Cryptographic Engineering, 18 Sep 2013. Web. 13 Dec. 2013.

"Introduction to Randomness and Random Numbers." Random.org. Web. 12 Dec 2013.

Shumow, Dan, and Niels Ferguson. "On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng." Microsoft. Lecture.

United States. National Institute of Standards and Technology. Supplemental ITL Bulletin. 2013. Web. <http://csrc.nist.gov/publications/nistbul/itlbul2013_09_supplemental.pdf>.

Demonstration:

This paper includes a demonstration of the bug that Cryptocat experienced with its CSPRNG and the resulting weakness. The bug, caused by an off-by-one error, lead to a certain digit being favored over others when constructing the random number. The Cryptocat.random() function produces a random number between 0 and 1 by constructing each digit using a Salsa20 byte and modulo. With each byte having 256 possibilities, the programmers attempted to restrict number to one that was divisible by 10 in order to produce a series of digits 0-9. However, the code used <= 250 in a conditional check, which allowed for 251 possible numbers (from 0 to 250 inclusive). Thus, this lead to the digit 0 being chosen more frequently than other digits, and this property of the CSPRNG will be highlighted in the example.

The code provided in this example was taken from the open source software Cryptocat. The original source code can be found at this location: https://github.com/cryptocat/cryptocat.

1. Download the source code included with this paper from Github (example.html and example.js).
2. With both files in the same directory, open example.html in a browser (excluding Firefox or Opera). For best results, use Google Chrome.
3. When the web page loads, a seed will be generated and 100,000 random numbers will be generated using the bugged CSPRNG. Then a new seed will be generated and 100,000 random numbers will be generated using the fixed CSPRNG.
4. The results will be written to the web page.

One will notice that in all cases, the digit '0' will be favored over the other digits (picked ~4% more of the time). This shows the weakness in the Cryptocat CSPRNG before it was discovered.

Visualizations by Paul Ducklin of Sophos:

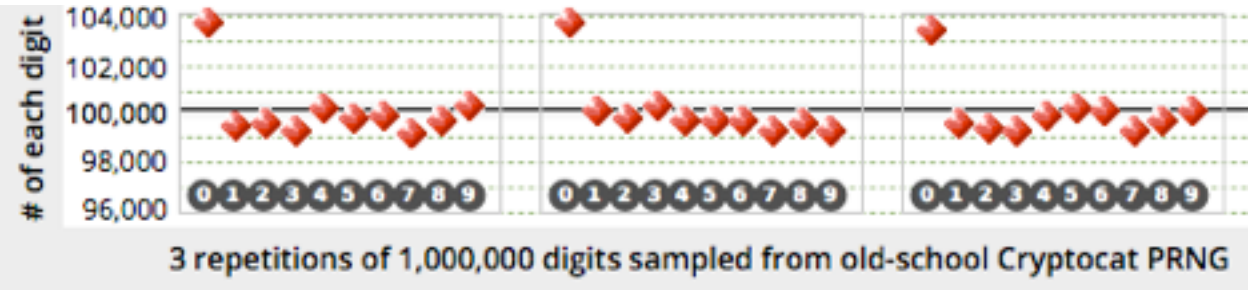

Figure 1: Similar test to demonstration run with one million digit samples. The digit 0 is chosen roughly 4% more of the time (Ducklin).

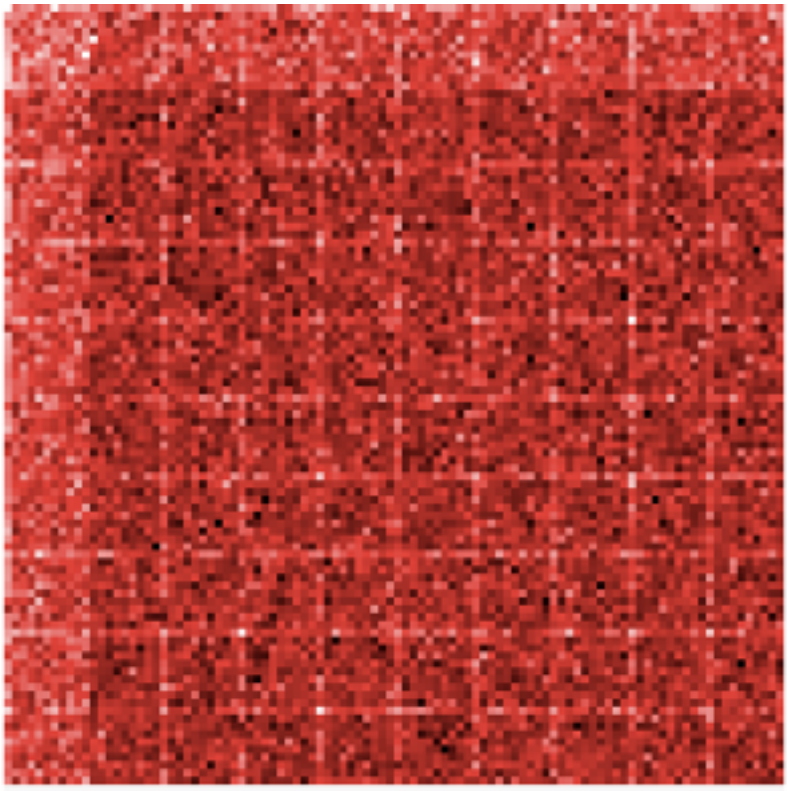

Figure 2: Numbers chosen by the bugged Cryptocat CSPRNG. The system shows clear biases towards certain numbers represented by darker squares (Ducklin).