True Random Number Generation for Cryptography, on the Cheap

Paul Nixon

pauljnixon@gmail.com

Ming Chow
Abstract:

This paper describes an open system for building a True Random Number Generator and using it for cryptographic purposes. The main components of this system are an Arduino Uno and OpenSSL.

Introduction:

Random number generation is an important part of cryptography, because flaws in random number generation can be used by attackers to compromise encryption systems that are algorithmically secure. True Random Number Generators, or TRNGs, use an external source of random information outside of the computer in order to gather entropy. Pseudorandom number generators use mathematical functions in order to generate a sequence of random numbers. PRNGs exist which have the property that even knowing a particular PRNG output it is impossible to statistically predict the next random number, but all PRNGs must be "seeded" with an initial value, and the sequence is predictable based on this seed value. So it's common to use TRNG output as seed data for a PRNG.

To the Community:

Recently, the NSA's long-rumored capability to intercept any Internet communications was revealed to be real. This increases the importance of cryptography because a wide-scale man-in-the-middle attack is actually being carried out. In the past it was a best practice to assume that all data would be intercepted, but now it appears to be probable. This attack is being carried out with help from such giants as Google and Verizon, so individuals cannot trust others to keep their data secure. The glimmer of hope in the NSA revelations is that NSA agents asked websites to hand over their private RSA keys. A depressing majority of them did so, but it suggests the NSA is still unable to crack the RSA algorithm. Since the NSA has deliberately sabotaged encryption algorithms in the past via their crippling of the Dual_EC_DRBG random number generator, it can be assumed that they would like to be able to crack RSA encryption. Since they asked for keys, it can be hoped that they have not
yet done so. The NSA is not the only enemy, but they have devoted considerable resources to hiring
the best codebreakers they can find, so their inability to break RSA makes it less likely that RSA is
broken.

RSA is an asymmetric algorithm, meaning that the sender and recipient of the encrypted data
use different keys, one public and one private. The same keypair is often used many times, so a
compromised private key can be used to spy on many conversations. RSA encryption cannot encode
large quantities of data, but it is often used to encode AES keys. AES is a symmetric algorithm, so the
same key is known by the sender and the recipient, and hopefully only those two systems. The practice
of using an RSA keypair to share an AES key is used in SSL/TLS, the encryption underlying HTTPS.
While RSA encryption may be the best bet for individuals hoping to keep data secret, it alone does not
guarantee security. Any encryption can be brute-forced by an attacker who takes the time to guess and
check possible encryption keys. This attack is guaranteed to work given enough time, but longer keys
mean that more possibilities have to be checked, to the point that brute-force attacks on some keys have
expected runtimes longer than the expected lifetime of the universe. It is for this reason that RSA
recommends using keys of at least 2048 bits.

Known attacks on RSA mostly focus on ways to speed up guessing the key. Both private and
public key are generated from a pair of primes, with the private key containing both numbers but the
public key containing only their product. This means that not all numbers of a certain size are suitable
for use as RSA keys. Because the key is supposed to be randomly generated from among the
possibilities, many attacks on RSA are actually attacks on the random number generation. One such
attack was allegedly perpetrated by the NSA when they sabotaged the Dual_EC_DRBG standard for
random number generation. The standard was included in a larger NIST document describing four
different algorithms for pseudorandom number generation at the NSA’s insistence, but it was later
shown that the algorithm had a flaw allowing an attacker with knowledge of the relationship between
some of the algorithm's constants to break encryption using the PRNG. The NSA never explained how
they generated the constants which the algorithm used, so it was suspected the NSA had such a backdoor.

However, not all flaws in random number generation are deliberately created by malicious attackers. The Debian distribution of Linux and its derivatives (including Ubuntu) were revealed in 2008 to have a flaw involving insufficient entropy as seed data for their random number generation. The systems had initially used the generator’s process ID and a block of uninitialized data as seed values, but the line of code reading the uninitialized data generated a compiler warning and was removed. This meant that the number of possible keys generated by these systems was limited to the number of possible process IDs, less than 65535. Guessing a key was now within reach of an attacker. To avoid making it easy to guess keys, it is important to use a good random number generation system. Avoiding flawed PRNGs and using enough random seed data are both important. Good pseudorandom number generators exist, but they generate predictable sequences of random numbers based on their initial seed value. This is where True Random Number Generators come in. They can be used on their own for random number generation, but are more often used to seed cryptographically-strong PRNGs.

**Action Items:**

Cryptographically-strong PRNGs are widely available, including the SHA-1 cryptographic hash function included in OpenSSL. OpenSSL is in turn freely available on a variety of platforms. However, there is still the problem of seeding. Linux includes /dev/random which is a PRNG incorporating random data such as mouse movements, keyboard timings, and hard disk timing data. However, a headless server running an SSD does not have these sorts of entropy, so the current system was built to allow key generation using radio noise as seed data.

This system uses an Arduino, which uses an open source hardware design in addition to open source software. The Arduino has an analog read feature which allows the reading of atmospheric radio noise. A breadboard wire was used as an antenna. The Arduino reads integer values from the pin, but
only the least significant bit of each read value is actually used, to prevent the wider fluctuations of noise from creating predictable data.

Bits read from the Arduino's analog read capability are subjected to a transformation known as von Neumann randomness extraction. This is designed to correct the problem of a bitstream having more 1s than 0s or vice versa. If two successive bits are the same, no output is generated for those two bits, but if they differ, the first of the two is output. This results in an even distribution of bits, given that the bitstream was otherwise unpredictable.

After the von Neumann randomness extraction is applied, the resulting bitstream is sent over the Arduino's USB connection to a PC, via serial emulation. The bitstream is read by a C program which uses the bitstream from the Arduino to seed OpenSSL's PRNG. Then this program calls OpenSSL's key generation library to generate an RSA key. OpenSSL's PRNG consists of reading the seed data and hashing it with SHA-1, but the key generation process beyond that is outside of the scope of this paper.

**Conclusion:**

The system outlined in this paper is not unique in using atmospheric radio noise to create a TRNG. The most well-known of these systems is probably random.org, a service which dispenses random numbers over the internet for free, generated from FM radio noise. However, there are security concerns involved when random numbers are sent over the Internet, especially when those numbers are going to be used to hide data from the NSA. Statistical tests run on random.org have proven that its numbers are indeed random, but a man-in-the-middle attack could change the numbers. The random.org system isn't open source, so there's no hope for cloning it exactly.

SSL cards exist as add-ins for servers, but they are expensive, and implementation details are unavailable. The VIA C3 line of CPUs included a TRNG based on voltage fluctuations, but VIA no longer makes processors and VIA's competitors did not copy the feature.

The random.org system's statistically-proven randomness is an advantage over the current
system, however. Using the entire output of the Arduino's analog read unsurprisingly failed most statistical tests, but so did the low-order byte and even the low-order nibble. Von Neumann randomness extraction on the lowest-order bit helped greatly, but even with the extraction, the Arduino TRNG fails the "diehard" test suite. Encouragingly, applying the SHA-1 hash to the Arduino TRNG results in a bitstream which passes the diehard suite.

Sources:


http://www.random.org/faq/

http://stat.fsu.edu/pub/diehard/


https://www.schneier.com/blog/archives/2008/05/random_number_b.html

http://www.openssl.org/docs/crypto/rand.html

http://csrc.nist.gov/groups/ST/toolkit/rng/batteries_stats_test.html

http://linux.die.net/man/4/random