tics, clock, and various files. Now the next RC2 key generated will be based on new seed material as well as the old.

Conclusion

Done properly, random number generation in software can provide the security necessary for most cryptographic systems. Using a good PRNG and choosing good seed material are the two critical points.

Developers may wish to create a set of routines to pull random and unique information from the operating system, which can then be used in any applications requiring cryptography. It may be desirable to save encrypted seed state for use in subsequent sessions.

Over time, as the need for cryptography in software increases, hardware and operating system vendors may provide more tools and hooks for random information. In the meantime, however, the techniques described can be used.

Further Reading

For more information on random numbers and cryptography, take a look at the following:

- Donald Eastlake, Steve Crocker, Jeff Schiller, "Randomness Recommendations for Security," IETF RFC 1750, 1994
- Ian Goldberg and David Wagner, "Randomness and the Netscape Browser," *Dr. Dobb's Journal*, January 1996
- Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1981
- Colin Plumb, "Truly Random Numbers," *Dr. Dobb's Journal*, November 1994
- RSA Data Security, Inc., *BSAFE User's Manual*, Version 3.0, 1996
- Bruce Schneier, *Applied Cryptography*, John Wiley & Sons, Inc., New York, 1995

For more information on this and other recent developments in cryptography, contact RSA Laboratories at one of the addresses below.

RSA Laboratories
100 Marine Parkway, Suite 500
Redwood City, CA 94065 USA
415/595-7703
415/595-4126 (fax)
rsa-labs@rsa.com
http://www.rsa.com/rsalabs/

Copyright © 1996 RSA Laboratories, a division of RSA Data Security, Inc. All rights reserved.

RSA Laboratories' **Bulletin**

News and advice from RSA Laboratories

Suggestions for Random Number Generation in Software

Tim Matthews
RSA Data Security

Introduction

The generation of random numbers is critical to cryptographic systems. Symmetric ciphers such as DES, RC2, and RC5 all require a randomly selected encryption key. Public-key algorithms — like RSA, Diffie-Hellman, and DSA — begin with randomly generated values when generating prime numbers. At a higher level, SSL and other cryptographic protocols use random challenges in the authentication process to foil replay attacks.

But truly random numbers are difficult to come by in software-only solutions, where electrical noise and sources of hardware randomness are not available (or at least not convenient). This poses a challenge for software developers implementing cryptography. There are methods, however, for generating sufficiently random sequences in software that can provide an adequate level of security. This bulletin offers suggestions on generating random numbers in software, along with a bit of background on random numbers.

Random vs. Pseudo-Random Numbers

What is a truly random number? The definition can get a bit philosophical. Knuth speaks of a sequence of independent random numbers with a specified distribution, each number being obtained by chance and not influenced by the other numbers in the sequence. Rolling a die would give such results. But computers are logical and deterministic by nature, and fulfilling Knuth's requirements is not something they were designed to do. So-called random number generators on computers actually produce pseudo-random numbers. Pseudo-random numbers are numbers generated in a deterministic way, which only appear to be random.

Most programming languages include a pseudo-random number generator, or "PRNG." This PRNG may produce a sequence adequate for a computerized version of blackjack, but it is probably not good enough to be used for cryptography. The reason is that someone knowledgeable in cryptanalysis might notice patterns and correlations in the numbers that get generated. Depending on the quality of the PRNG, one of two things may happen. If the PRNG has a short period, and repeats itself after a relatively short number of bits, the number of possibilities the attacker will need to try in order to deduce keys will be significantly reduced. Even worse, if the distribution of ones and zeros has a noticeable pattern, the attacker may be able to predict the sequence of numbers, thus limiting the possible number of resulting keys. An attacker may know that a PRNG will never produce 10 binary ones in a row, for example, and not bother searching for keys that contain that sequence.

The detail of what makes a PRNG cryptographically "good" is a bit beyond the scope of this paper. Briefly stated, a PRNG must have a high degree of unpredictability. Even if nearly every bit of output is known, those that are unknown should remain hard to predict. The "hardness" is in the sense of compu-

Tim Matthews is a cryptographic systems engineer at RSA Data Security. He can be contacted at tim@rsa.com.

RSA LABORATORIES
CRYPTOGRAPHIC RESEARCH AND CONSULTATION

tational difficulty — predicting the bits should require an infeasible amount of computation. A true random number generator, like a hardware device, will have maximum unpredictability. A good PRNG will have a high degree of unpredictability, making the output unguessable, which is the goal.

One essential ingredient in producing good random numbers in software, then, is to use a good PRNG. Important to note is that although the PRNG may produce statistically good looking output, it also has to withstand analysis to be considered strong. Since the one included with your compiler or operating system may or may not be, we recommend you don't use it. Instead, use a PRNG that has been verified to have a high degree of randomness. RSA's BSAFE toolkit uses the MD5 message digest function as a random number generator. BSAFE uses a state value that is digested with MD5. The strength of this approach relies on MD5 being a one-way function — from the random output bytes it is difficult to determine the state value, and hence the other output bytes remain secure. Similar generators can be constructed with other hash functions, such as SHA1.

System Unique	Variable and Unguessable	External Random
Configuration files	Contents of screen	Cursor position with time
Drive configuration	Date and time	Keystroke timing
Environment strings	High resolution clock samples	Microphone input (with microphone connected)
	Last key pressed	Mouse click timing
	Log file blocks	Mouse movement
	Memory statistics	Video input
	Network statistics	
	Process statistics	
	Program counter for other processes or threads	

Less Entropy ⟷ More Entropy

Table 1
Seed Sources

The Seed

The other component in producing good random numbers is providing a random seed. A good PRNG like BSAFE's will produce a sequence that is sufficiently random for cryptographic operations, with one catch: it needs to be properly initialized, or "seeded." Using a bad seed (or no seed at all) is a common flaw in poorly implemented cryptographic systems. A PRNG will always generate the same output if started with the same seed. If you are using MD5 with the time of day as the seed, for example, an attacker has a high likelihood of re-creating your sequence of pseudo-random bytes by guessing the exact seeding time. Once he has the pseudo-random bytes, he can re-create your keys. The security issue becomes one of making sure an attacker cannot determine your seed.

You may be wondering why use a random number generator to generate random bytes, if to use it, you need to first generate random bytes. Seeding is a bootstrap operation. Once done, generating subsequent keys will be more efficient. Another important point is that the information collected for the seed does not need to be truly random, but unguessable and unpredictable. Once the seed is fed into MD5, the output becomes pseudo-random. If attackers cannot guess or predict seeds, they will be unable to predict the output.

There are two aspects to a random seed: quantity and quality. They are related. The quality of a random seed refers to the entropy of its bits. Cryptographers use the word entropy a lot, so it is worth knowing. In a system that produces the same output each time, each bit is fixed, so there is no uncertainty, or zero entropy per bit. If every possible sequence of outputs is equally likely (i.e. truly random) then there is total uncertainty, or one bit of entropy per output bit. There are precise mathematical formulas for entropy, but the short summary is the more entropy per bit, the better. Since the quality may vary, it is a good idea to account for this with quantity. Sufficient quantity makes it impractical for an attacker to exhaustively try all likely seed values. Let's start with quality.

Table 1 shows a list of potential sources for building the initial seed pool. External random events are the best, but harder to get than variable or unique information. Sources that are variable, while not random, are very difficult for an attacker to guess. Quantities that are unique to a system are also hard to guess and usable if more bytes are needed.

In general, collect as much external random information as possible. Supplement this with sources from the two other columns if more bytes are needed. Using a composite of many items makes the attacker's task more difficult. In an application where several keys will be generated, it may make sense to collect enough seed bytes for multiple keys, even before the first is generated. Be careful of information that moves across a network that could be intercepted by a dedicated attacker. Mouse movements on X-terminals, for example, may be available to anyone listening on the wire.

Now we get to the issue of quantity. A developer cannot assume that all of the bits collected are truly random, so a useful rule of thumb is to assume that for every byte of data collected at random, there is one bit of entropy. This may either be a bit conservative, or a bit generous, depending on the source. To illustrate this rule of thumb, take the example of user keystrokes, which many consider to be a good source of randomness. Assuming ASCII keystrokes, bit 7 will always be zero. Many of the letters can be predicted: they will probably all be lowercase, and will often alternate between left and right hand. Analysis has shown that there is only one bit per byte of entropy per keystroke.

To guard against this kind of analysis, the idea is to collect one byte of seed for each bit required. This information will be fed into the PRNG to produce the first random output.

As an example, if the seed will be used to produce a random symmetric encryption key, the number of random bytes in the seed should at least equal the number of effective bits in the key. In the case of DES, this would be 56 random bits culled from a seed pool of 56 bytes. Any less and the number of possible starting keys is reduced from 2^{56} to something smaller, reducing the amount of effort required by an attacker in searching the seed space by brute force. Attacks like this have recently been widely publicized on the Internet and in the press. For public-key algorithms, the goal is to make searching for the seed at least as difficult as the hard mathematical problem at their core. This will discourage attackers from searching for seeds instead of attacking problems like factoring composite numbers and calculating discreet logarithms. A seed of 128 bits (taken from a seed pool of 128 bytes) should be more than enough for the modulus sizes being used today.

One last thing that should be mentioned is updating the seed, or "re-seeding." It makes sense to allow an application to add seed bits as they become available. User events often provide additional sources of randomness, but obviously have not taken place when an application starts. These should be included as they occur. Re-seeding also frustrates attackers trying to find the seed state using a brute force attack. Since the seed will be change, say, every thirty seconds, the seed state becomes a moving target and makes the brute force attack infeasible. The idea is to take the existing seed and mix it together with the new information as it becomes available.

Example

A brief example is in order. The diagram in Figure 1 illustrates how functions in BSAFE would be used to generate random keying material.

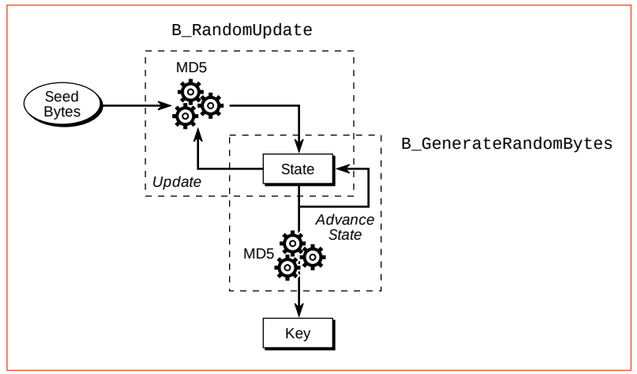

Figure 1
Random Seed Process

The first step is to supply the pool of random seed bytes. Let's assume that the application needs a random 80-bit RC2 key. Using the rule of thumb that one byte of data yields one bit of randomness, a minimum of 80 bytes will be needed for the pool. This pool would be gathered from the sources listed in Table 1. The B_RandomUpdate function in BSAFE takes the seed pool and runs it through the MD5 message digest algorithm to create the state.

The state is then used by the function B_GenerateRandomBytes, which runs it through MD5 to produce the key. This is the key that would be used for RC2. As an added measure, BSAFE automatically advances the state after random bytes are generated.

Notice the arrow labeled "*Update*" within B_RandomUpdate. This is where re-seeding is done. By calling B_RandomUpdate again, the state can be mixed with more seed information. Random information like key timing and mouse movement can be used here, along with changes in the system statis-