Using and Creating Cryptographic-Quality Random Numbers

Jon Callas, jon@worldbenders.com

3 June 1996

There’s an old mathematician’s joke that few decades ago, statisticians were concerned about the random number tables they had to use. They were difficult, ungainly, and just a pain. So a number of people got together and decided that if they could find out what the most random number is, they could just use that and lose all those thick books of random numbers. After much work, they found out that the random number is 17. I won’t bore you with the proof — much. The proof shows that The Random Number cannot be larger than 17, and then shows that it cannot be less than 17. Therefore it must be 17. The flaw in this proof is, of course, obvious.

Most of us know what random numbers are traditionally used for: games, simulations, testing, numerical analysis, etc. However, a number of us are now starting to work with privacy software, and privacy software needs to use random numbers for generating public / private key pairs, producing session keys, creating some digital signatures and message authentication codes, and many other uses. But we can’t just use 17, and we can’t even use the functions that come with our favorite development system. Creating good random numbers is a hard problem, so hard that there isn’t a library we can just use.

In this paper, I will talk about randomness, what it means, how you get it, how you use it, and how to misuse it. This is a hard problem. Most of the mistakes I’ll talk about I’ve made myself. Given the nature of the study, there’s likely at least one comment in this paper that I’ll regret in five years. The good news is that with a little bit of study, humility, and cleverness a good hacker can do some good work.

What does it mean to be random?

This is a good question. In fact, it’s the crux of the issue. Random numbers, at some sense can be thought to be arbitrary, unknowable, unpredictable. Because we’re working on computers, we can think of random numbers as being random bits, or streams of random bits, which we group together, form modulos on, use in massaged forms.

The canonical mathematical definition is that a string is random if there is no shorter way to state the string than to state the string itself. In plain English, it's random if you can't compress it. Note that this means that compression is a randomizing function.

Statistics tells us a few things about random bits. Zeros ought to occur as often as ones. A pair of zeros should occur half as often as single zero, and as often as a pair of ones. A triplet ought to occur half as often as a pair, a quarter as often as a single bit. If you take samples in pairs, a graph shouldn’t show clumps. There are other statistical tests such as the chi-squared test you can perform on a stream of numbers to show how random (or how much they stray from random) they are. The degree to which a stream of bits follows this statistical form is the degree to which it is said to be entropic. This is the strict mathematical definition. Please note that there is no well-defined term for something that is entropic and unknowable. Many people (including me) casually use the term "entropic" to mean statistically random and unknowable.

These are good enough for traditional purposes (although some traditional random number generators have some amusing flaws — most implementations of rand() show clumps if you plot pairs on an x-y graph), but not in privacy and cryptography. The notion that a random number is unknowable is central to cryptographic work. If you construct a zillion digit prime number, and then post it on your web page, you might as well just use 17. It’s no less safe, and you can probably remember it.

Cryptographers use the notion of “The Adversary” to help them focus their thought. The adversary is a hypothetical person who is trying to break the system. Depending on what you are trying to do, the adversary has certain abilities. The adversary can always disassemble your code and understand it. The adversary can always use brute force. The adversary can’t read your mind. For us on Macintoshes, it’s safe to assume our user can (sometimes) use the system without the adversary watching. For our purposes, a perfectly random number is one that the adversary has to guess, that is that there is no strategy for determining it that is better than brute force. Note that this is equivalent to the mathematical definition. If there were a shorter way to state the string, then this would form the basis of some way to guess it that's more efficient than brute force. Strong cyphers are also randomizing functions; the cyphertext output of a function is random, and ideally perfectly random. It follows all the statistical tests, and in effect looks like noise. Much of cryptanalysis centers around breaking codes by discovering and exploiting imperfections in a function’s ability to randomize.

Random numbers may be strong or weak (perfectly random numbers are the strongest), and also knowable or unknowable. Knowable random numbers are also called pseudo-random numbers. Pseudo-random numbers are knowable because they come from a mathematical function. For example, the number produced by summing the digits of your date of birth is completely pseudo-random and knowable because it’s a fixed formula, but it may be secret. Flipping a coin is truly random, assuming the right coin, but pretty easy for the adversary to guess. A lopsided coin may be statistically skewed, and thus weakly random, but also truly random.

Collecting, Cooking, and Feeding Entropy

Collecting truly random numbers is hard. Collecting perfect random numbers is hard. Collecting perfect, truly random numbers is non-trivial. The best way to do it is to treat each problem separately. Figure out how to get entropy, or randomness, or what I’ve called how truly random the number is, and then use that entropy to produce something that is statistically (ideally perfectly) random. In short, we want to get some entropic numbers, and then use them to seed a pseudo-random number generator. Better still, set up a process whereby we continually sample entropic numbers and continually and cumulatively seed a pseudo-random number generator.

It’s a good idea to collect as much entropy as possible before using a system. If you are creating a key pair, it’s a good idea to have a tenth to a fifth as much entropy as the length of the key (if you’re generating a 1024-bit key, it’s a good idea to have 100 to 200 bits of entropy). More is of course always better. Also note that by one bit of entropy I mean that it’s as random as a coin flip. A coin marked with 17 on one side and 23 on the other side has one bit of entropy, but anyone foolish enough to assume it’s the low bit is going to be in for a rude surprise. This is, in fact, a common mistake! Don’t assume that the low bit of a random sample is the random bit! In fact, the entropy may (as in the coin above) be smeared across the whole of the sample, or aggregated across multiple samples.

There are many ways to get truly random numbers. Some outré methods include making hardware devices that generate noise, observing cosmic ray flux, and observing light emissions from trapped mercury atoms. They’re great in theory — and in some sorts of practice, as there are some very high-quality random generator chips out there — but we have to work with existing equipment. Here is a list of ways to get entropic numbers, and my biased opinions on their pros and cons:

Use your user. Users are among the most entropic things there are. The obvious drawbacks are that it takes time to get entropic responses from the user, and if you have no user (if you’re a server), this doesn’t help. Nonetheless, if you do have one, you should use user inputs, as these are the hardest for the adversary to acquire or spoof. You can:

Look at the keyboard. Timings between keystrokes aren’t bad. People tend to type rather regularly, except when they don’t. Ticks between keystrokes are probably good for one bit per stroke. If you use the value of the keystroke itself there are two dangers. One is that you might leak information about the person’s typing, which can be bad, and could ruin the whole game if it’s the passphrase for their key that you leak. The other is that English text contains only about an average of 1.3 bits of entropy per character (or so say Cover and King from an IEEE paper). My paranoia meter tells me to avoid using keystroke content because I fear leaking valuable information, unless I use a good distiller (which I’ll discuss below) and am careful to use other streams of entropy too.

Look at the mouse. Timings between clicks are probably better than timings between keystrokes, but they happen less often. There’s easily a bit there. The position of the mouse is also a good one, but there is obvious skew. Mice spend a lot of time in the menu bar. There’s a lot of white space in the menu bar, and menus aren’t used evenly. Dialog boxes tend to go in the same places, and lay out their components regularly. If you measure the mouse position in an event, it’s a mediocre sample. My gut feel is that there’s a bit of entropy in the whole mouse position. If you sample it in a VBL, null event, or some other free-running time, I feel confident at guessing that it’s got two bits of entropy, when moving.

If you have no user, you have to use your hardware for a stream of entropy. This is annoying, but not impossible. Here are some methods you might use:

Look at the clock. Simply looking at the clock provides some, but not a lot of entropy. If you use the time() function in the C library or the GetDateTime() function from the toolbox, it’s pretty easy for the adversary to search outward from now, and home in on your value. Finer resolution clocks are better, but the same principle holds. If you seeded a PRNG with TickCount() and nothing else, a search will find it quickly. There are only 31.5 million ticks in a year. However, a stream of observations that look at the clock at the primary interrupt time (and use the microsecond clock) can get a lot of entropy quickly. Of course, there are other considerations, such as the fact that there is at least one regular interrupt on the system, and it’s possible for the adversary to do things to generate interrupts (like ping you).

Use the disk. At the Crypto ‘94 conference, Davis, Ihaka, and Fenstermach showed that air turbulence inside a disk drive creates enough randomness to make cryptographic-strength random numbers. Obviously, there are a number of ways to do this badly, but with a good enough clock, and properly flushing (or disabling) the caches, this will work. A unix implementation of this scheme has generated over 100 bits per minute.

Use a microphone. Reading the microphone that is built into many Macs (like the PowerBook I’m writing this on), gives a source of apparently random data. I say apparently because this technique gives me the willies. White noise frequently isn’t. As I listen to things in the room here with me, there is 60Hz hum from power sources, A whine from my disk drive, fans in the room which are noisy but still have a regular component from their rotation, whine from nearby monitors, and so on. If I were mixing disk timings with sound samples, I would worry about how the disk's chatter would make the two dependent on each other. Second, signal analysis is one of the most developed disciplines there is, and that makes me worry. Lastly, I would worry that two servers in the same room would be too tied to each other as the environment they are sampling is similar. I wouldn't use sound alone, and I wouldn't use it and disk timings alone. Other people I respect recommend compressing audio information as a way of removing skew, and then using it. As long as audio data is distilled with some suitable distiller (as I'll discuss below), it can be a fine source of randomness, in spite of the willies it sometimes gives me.

Use video. Reading the video memory of your computer can give another source of randomness. A number of people recommend reading the video buffer under the mouse cursor, too.

Use the network. There are all sorts of things on the net that are unpredictable. However, they are as accessible to other people as they are to you. Packet timings could be observed and then guessed about, so that they'd be only weakly random. Other interesting things that seem random may not be. Certainly it is arbitrary what the checksum of all of yesterday's netnews postings are, but if the adversary used the same netnews provider, its entropy value could be as low as zero. There are other ways a determined adversary could observe your readings, or even feed them to you! As long as you consider network readings to be weakly entropic, they're probably okay.

Use your computer's state. These observations can be anywhere from worthless to excellent. The number of page faults that have happened on a VM system might increase wildly, or sit static. Checksumming memory might be worthless if a helpful program like Ram Doubler is zeroing it to make it more compressible! Some values, like your ethernet address are arbitrary, but not random (since a good adversary already has that along with your SSN and your mother's maiden name).

Distilling Randomness

Once you have a big bag of random observations, or a stream of them that you can generate over time, what do you do with them? You want to mix them together, and get a value that is as random as all of them together; we want some entropy-preserving additive function.

The simplest, and one of the more useful is the XOR. It's cheap, it's fast. The drawback of using XOR is that if the random samples are in some way correlated, the correlation will actually remove randomness from the XORed streams.

Cryptographic functions can distill entropy. For example, you might take the first 8 bytes of sample, and repeatedly DES encrypt it with further samples, you'd get something that would have up to 56 bits of randomness, to the degree with which DES is a randomizer (differential cryptanalysis studies show that DES is non-random to a degree of approximately 10^{-8} bit per operation). Unfortunately, this technique is quite slow.

Checksums are also excellent ways to distill entropy, so long as you pick a good one. A checksum designed for error recovery (examples are any CRC) is a poor function to use. CRCs in particular are designed for the exact opposite of what we want. We want differences in a data stream to compound on each other, while a CRC is specifically designed to flag differences and allow some of them to be recovered.

If checksums are the right idea but the wrong function, then what is the right function?

A class of functions called “message digests,” “cryptographic checksums,” etc. have a number of useful properties. They are designed for producing digital signatures, integrity checks, and so on, and to do so in the face of an adversary who wants to forge one of these. Digesters are designed in Ron Rivest’s words to be secure in that, “It is computationally infeasible to find two messages that hashed to the same value. No attack is more efficient than brute force.” Thus, if we hash a series of random samples, it is not practical to learn what those samples were. Note that this is the same thing as my definition of the output’s being perfectly random, and thus the hash preserves the entropy found in the samples. This has the additional useful effect that many weakly entropic samples can be combined to carry entropy equivalent to the whole of the entropy of all the samples. Note that while these hashes distill entropy, they cannot hold more entropy than their size. They are bottles that you can shake up entropy in, but a 128-bit hash cannot hold 129 bits worth of entropy.

These characteristics make them excellent functions for our purposes. Common functions used today are MD2, MD4, and MD5 (done by Ron Rivest), SHA (the US government’s Secure Hash Algorithm, also known as SHS, the Secure Hash Standard), GOST (the Russian hasher), RIPE-MD, and RIPE-MD160 (which are European standards for hashes).

Here is a short discussion of a few commonly used hash functions:

MD4 is a 128-bit hash. It was broken in late 1995. By broken, I mean that a message like, “I promise to pay Mary $100” has been found to have the same hash as, “I owe Mary $10,000, please pay it to her.” It is therefore completely unusable for producing signatures, but probably still has some use in distilling randomness. It has the advantage that it is very fast. If you use it, be prepared for tongue-clucking. I stopped using it several years ago.

MD5 is an improved version of MD4. It isn’t as fast as MD4, but was designed to improve its weaknesses. MD5 is widely used for computing digital signatures. Recently, some cryptanalysts have found weaknesses in MD5 similar to weaknesses found in MD4 before it was broken. This makes many people nervous about it.

Since I started writing this article, Hans Dobbertin of the German Information Securoty Agency (and one of the inventors of RIPE-MD160) claims to have found a new weakness in MD5's compress stage. Without going into details, I note that you can find a copy of his paper at <http://www.cs.ucsd.edu/users/bsy/dobbertin.ps> or write to him at <mailto:dobbertin@skom.rhein.de>.

SHA is another improvement on MD4, developed at the NIST. It has all the improvements of MD5, with some others, including those that make it a larger hash. It is a 160-bit hash, and has no known weaknesses.

RIPE-MD is another variation of MD4, also designed to improve it. It has also been broken recently, and a 160-bit version called RIPE-MD160 has been produced.

MD2 is another of Rivest’s hash algorithms. It is also a 128-bit hash, and no weaknesses have been found in it. Its drawbacks are that it is slower than most other hashes (one-tenth the speed of MD4, one-seventh the speed of MD5 and RIPE-MD, and one-third the speed of SHA) and that when it was originally released, it was released with a license saying it was freely usable in privacy enhanced mail systems only.

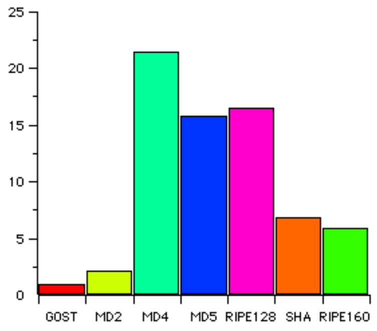
There are also a number of hashes that are based upon using cyphers to mix up the hash values, including the Russian

GOST hash, a 256-bit hash. Unfortunately, they're all slow — GOST runs at less than half the speed of MD2. Speed, is of course relative. GOST can hash about 20 megabytes per second on a Quadra 950, but if you want to set up a system that hashes something every interrupt, this is quite slow. Also, for US software developers, while it is legal to export a cypher that you're only using in a hash function, many people don't want to take the chance. This is what is known to the lawyers as a "chilling effect."

Some people in the crypto world are a little worried about the current state of hashes, as the fast, exportable ones (which are also the basis of the approved signature standards of both the US and the EC) all come from a "monoculture" of MD4 variants. The 128-bit strains have been broken or are showing signs of strain. The 160-bit versions seem good, but people worry.

I recommend using MD5 or SHA. MD5's weaknesses are disturbing, but we're not producing signatures, we're distilling entropy. The generator I include with this paper uses SHA, but mostly because it is a bigger bottle.

Following is a graph of the comparative speeds of the hash functions I have discussed here. The data come from Bruce Schneier and Wei Dei.



Using a distiller

An advantage of using a hash function as a distiller is that hash functions give a different answer if you hash things in a different order. Thus, if the adversary knew that you hashed A and B, but did not know what order you hashed them in, at least you get one bit of entropy, while you'd get none if you simply XORed them.

Another is that there can be an advantage to hashing things with little or no entropy. For example, suppose you had a VBL task that hashed the mouse position on every interrupt. Obviously, if the mouse hasn't moved, there is no additional entropy gained, but since H(A,A,B) != H(A,A,A,B) this second strategy has the advantage of introducing some entropy based not only on the mouse movement position, but mouse movement timing. I and many other people recommend hashing a lot of things because they might be useful particularly in a generator that continuously updates its store of randomness.

I recommend thoughtfully deciding what are good things to hash, constructing a process by which samples are regularly added to the distiller, and then using the distiller output as the seed to a secondary pseudo-random number generator (PRNG).

Pseudo Random Generators

There are a variety of functions you can use as PRNGs that are random to cryptographic strengths. A simple one is to use chained hashes. For example, after hashing your seed values, produce the final hash, take one byte of the hash as a random value and then hash the whole checksum back into the soup. If you are continually adding entropy to the system, this is effectively a perfect, truly random generator.

Another scheme is to use a cypher or a cypher-variant that is seeded by a hash function.

Depending on your tastes, either scheme works. Repeated hashing is less well-studied than a hash-seeded cypher (breaking such a PRNG is equivalent to breaking the cypher using a hash-derived session key), but hashes are designed to be non-reversible. On the other hand, people have had more luck breaking hashes than cyphers recently. On the third hand, repeated hashing is significantly faster than most cyphers. On the fourth hand, if you picked a fast cypher with few problems known, you could get the best of both worlds.

In fact, the generator I have selected for this paper combines a SHA distiller with a “grinder” that is designed along the same lines as high-performance stream cyphers.

The goals of this generator are that it be high-quality, allow for continuous update, and be reasonably fast. To achieve this, I've created it with two stages. The first stage is a distiller. I chose SHA as my distilling function, but you can easily replace this with some other function. My intent is that some background process will continuously update it with new observations. Also, there is a rating scheme for the observations. Whenever you enter an observation, you can rate the entropy value of the observation in tenths of bits. I picked this so you can easily add in many weakly entropic observations, and even observations that are arbitrary, but not at all entropic (like the ethernet address of the machine you're running on).

Every time the first stage gets enough observations to fill its entropic pool, it overflows into the second stage. The second stage of the generator is a stream generator, similar to an algorithm compatible with the stream cypher RC4. This algorithm differs from RC4 in that its list of bytes vary from 255 to 0 instead of 0 to 255, and by starting its stream from the 0th element of the array, rather than the first. Each time the first stage overflows into the second stage, I use a process similar to the key preparation for a stream cypher except that I generate 256 random bytes and discard them, so as to further mix up the pool.

This has a number of useful features. Using a distiller makes it easy and desirable to continuously update an entropy pool with a variety of observations of varying strength. By using unrelated functions in both stages, the generator capitalizes on the strengths of each stage. The second stage holds a vast amount of randomness when it is iteratively updated; the byte array can be in a total of 256! * 256 * 256 states (this is roughly 2^1700 states — note that this is somewhat smaller than the obvious guess, that of 2^2048 total bits of entropy, but certainly an excellent pool). It is also extremely non-linear, and resistant to linear and differential analysis. Lastly, this technique has advantage that new observations update the whole pool, rather than restarting it, and that using the random number generator interacts with the updates in a way that actually adds to the total entropy of the system.

Could We Do Better?

On the one hand, it's certainly arrogant to think that we couldn't. On the other hand, if I had a better generator lying around, I would have featured it in this article.

If I wanted to do better, I might look at the Blum, Blum, Shub (named for its inventors) generator. Glossing over its features, it has a number of useful properties, not the least of which are that it is provably as strong as factoring. On the other hand, it requires generating two large prime numbers, it is slow, and factoring is only conjectured to be difficult. Also, because it is based upon knowing prime numbers, it can't be the final step of a system like this one that collects large amounts of entropy. I could imagine an extension to this system that used a Blum, Blum, Shub generator along with others.

Conclusions

Generating good random numbers is difficult, but not overly so. If you use a distilling function, make many observations of entropy, and make a habit of guessing low about their valued entropy, you can quickly acquire enough entropy to run a good generator.

If you use a generator designed for cumulative update and actually update it, then you can get further good results.

If you want to read more about cryptographic systems, randomness, entropy, and pseudo-random generators, here are some sources:

Internet RFC 1750, by Eastlake, Crocker, and Shiller. Available from fine web sites everywhere. This includes discussions of randomness, other distilling functions that I didn't discuss, and some other security considerations. It also contains a good bibliography.

There is also a draft appendix to RFC1750 written by Carl Ellison (of Cybercash) and Burt Kaliski (of RSADSI). This draft contains a lot of good information about physical ways to acquire randomness for those who like building hardware. It also contains a good discussion of places not to get randomness from. Carl also has a good bit of source code for playing with and measuring random numbers.
<http://www.clark.net/pub/cme/P1363/ranno.html>

Applied Cryptography, Second Edition, by Bruce Schneier. Published by John Wiley and Sons, ISBN 0-471-11709-9. If you only have one book on cryptography and privacy, it should be this book. It is simple enough for an intelligent layperson to read, contains enough meat to let you see some of the innards of systems, and an extensive bibliography.

Foundations of Cryptography (Fragments of a Book), by Oded Goldreich. Oded Goldreich is a mathematician and cryptographer the Weizmann Institute of Science. He is presently visiting MIT. Several years ago, he wrote four chapters of a planned ten-chapter book. He hasn't made progress on them in a while, but in late 1995, he put his fragments on the Internet in both Postscript and HTML forms. The work is mathematical, but contains proofs for things I have merely asserted here concerning distillers, hash functions, and PRNGs. If you want to see the theorems behind the theory, this is the place. Available from <http://theory.lcs.mit.edu/~oded/>.

Random.c:

```
//
// Two-Stage Cryptographic Random Number Generator
//
// Copyright © 1996, Jon Callas, Eldacur Technologies. All Rights Reserved.
//
// You have permission to use this code in any commercial or non-commercial
// programs provided you do the following:
//
// (1) Somewhere in your documentation, about box, etc. you state that you use Jon
//     Callas's Two-Stage Random Number Generator and that it is copyright by him.
// (2) If you use it in a commercial product, you send me email at
//     jon@worldbenders.com telling me that you're using it. I don't have any
//     nefarious purposes, I'm not going to sell your name to some mailing list, I
//     just want to know who is using it, so I can send updates to you and be able
//     to brag about the bizillion people that use this.
// (3) If you find bugs, make substantive improvements, etc. that you make a best-
//     faith effort to send them to me so I can incorporate it into the code.
//
// That's it. That's all you have to do. It's not much to ask. It's not as odious
// as a copyleft so why not do it?
//

#include "Random.h"

#ifdef COMPILE_COMMENTS

What this is:

This is a high-quality random number generator that can create large amounts of good random
numbers. When used properly, it can hold some 1700 bits of randomness, or entropy in it. This
is more than enough for any purpose you need to use it for. However, like any piece of
precision machinery, it must be used properly and handled with care.

Please take the care to read these instructions and follow them.

HOW TO USE IT
--- -- --- --

The base object type is a Randomizer. There are also objects here for the two stages, the
Distiller and the Grinder. They'll be explained below. Usually you don't ever have to use
them.

(1) Make a Randomizer. The statement:

       Randomizer *r = new Randomizer();

is good enough.

(2) Find a source of random inputs. Real random inputs. Best things to use are user-driven
inputs like mouse positions, timings of keystrokes, and stuff like that. Each observation you
seed the Randomizer with should have a guess of how random it is, in decibits. Yeah,
decibits. Tenths of bits. If you think it's as good as a coin toss (like the clock time),
then it should be 10. If it's a really mediocre observation, use 1. If it's junk that you're
using because you like it (like your ethernet address, contents of files, etc.) throw it into
the mix, but
rate it 0. It all gets mixed up into the first stage, the Distiller. Hint: Always Guess Low,
Especially If You Are Writing Privacy Software! Once the Distiller is filled with enough
entropy, it will empty itself into the second stage. You can force the Distiller to empty by
using a negative number for your entropy guesss.

Add in seed values with:
```

```
r->AddSeed(sourcePtr, length, decibits);
```

For example, you'd add in a timing observation with:

```
r->AddSeed(&timer, 4, 10);
```

You can find out how much entropy is in the system with the call:

```
r->GetEntropy();
```

(3) You should make a habit of periodically adding stuff to the mix. Sample the mouse and put X,Y position in. Time how long it takes you to create, write garbage into, and delete a 1MB file. Have fun. If you are in the habit of using a variety of observations, and mixing them in to the Randomizer, the accumulated entropy will help your system be random. And remember, when you guess about your randomness, guess low, especially in the things you gather in the background!

(4) Get random values with these functions:

```
r->Byte(); r->Word(); r->Long();
```

These return a random 8, 16, and 32 bit quantity respectively. The function:

```
r->String(char *position, long size);
```

will fill the memory 'size' long pointed to by 'position' with random bytes.

(5) Saving State:

If you need to save the state of the Randomizer, you can call:

```
r->GetState(RandomState *state);
r->SetState(RandomState *state);
```

to save and restore the state. You should consider the state of the randomizer to be sensitive data! Treat it with the same respect you would treat password files, private keys, and other sensitive data. Don't let an adversary break your privacy system by scamming your random numbers!

HOW IT WORKS
--- -- -----

Stage 1 is the Distiller. It is a cryptographic checksum, specifically the Secure Hash Algorithm. The implementation of SHS I use here was written by Paul Kocher, given to me by Paul, and used with his kind permission.

A cryptographic checksum like this can be used to accumulate entropy. There is a problem when you toss together things you think are random. The problem is that they probably aren't as random as you think they are, and they probably aren't random in the way you think they are. The wonderful thing about a checksum like this is that it can distill all the entropy in your samples. It doesn't increase entropy, but it holds it up to its maximum (in the case of SHS, 160 bits). It can even be used in things that are weakly random.

An example of a weakly random source follows: Suppose you were sampling white noise from a microphone in the lab you put your HoozieServer 2000 in. Let's also suppose that while it just sounds like hiss to you, if we do fourier analysis on the samples we find that it's all just a bunch of harmonics except for some small bit of noise that makes your sample be X mod Y one time in four, and A mod B the other three times in four. Depressing, huh? It's not hopeless, though. It does suck that you thought you were getting 16 bits of white noise on each sample of the microphone, but there is still some value here. There's actually a half-bit of randomness in any sample you make. Yeah, fractional bits sound weird, but they really represent the flip of an off-kilter coin. You can still use them, so long as you don't overestimate their value.

This is why I tell you to make many samples, and always guess low. Murphy *is* out to get you.

Once you have accumulated enough entropy in the Distiller, we move it to the second stage of the generator. I trust your guesses, when you tell me there's enough, I go for it. That's why I'm being such a nag.

The second stage of the random number generator is a non-linear, arithmetic streamer. Its guts are mathematically similar to some of the non-linear streams used in some high-performance stream cyphers, but adapted to the needs of a random number generator, as opposed to a stream cypher.

The combination of the two of them give you a way to hold up to 2048 bits of entropy and use it to generate crypto-quality random numbers. If you use it properly, it will serve you well. If you misuse it, don't come complaining to me.

If you need to know more about random numbers, what makes a good random number, why the rand() function isn't good enough, or why things you think are random aren't, look at some of the following sources:

Internet RFC 1750, available on a web site near you.
Bruce Schneier's "Applied Cryptography", available in fine bookstores everywhere.

```
#endif

//
// The Distiller object, the first stage generator
//
Distiller::Distiller()
{
       Init();                         // Initialize the object
}

void Distiller::Init()
{
       // The only reason this routine exists, as opposed to
       // keeping it in the constructor is that I thought it might be
       // useful to be able to re-init an existing distiller.

       SetEntropy(0);
       shsInit(&shs);
}

int Distiller::AddData(void *start, long length, int decibits)
{
       totalEntropy += decibits;       // Update the guess
       shsUpdate(&shs, (unsigned char *) start, length);
                                       // Hash in the observation

       return totalEntropy;
}

void Distiller::GetHash(unsigned char hash[HASHLEN])
{
       SHS_CTX context = shs;                                              // Make a
copy of the hash state
       shsFinal(&context, hash);       // Finalize the copied hash.
}

//
// This is the Grinder object, the second stage of the generator
```

```
//

Grinder::Grinder()
{
	numberPlace = 0;
	newPlace = 0;
	InitNums();
}

Grinder::Grinder(void *theStuff, long stuffLen)
{
	// We don't actually ever use this constructor, but here it is, in case
	// you want to make a Grinder and give it some stuff.

	numberPlace = 0;
	newPlace = 0;
	InitNums();
	AddStuff(theStuff, stuffLen);
}

void Grinder::InitNums()
{
	for (int i = 0; i < 256; i++) // put the numbers 255 to 0 into the array
		numbers[i] = ~i;
}

void Grinder::AddStuff(void *theStuff, long stuffLen)
{
	int			i;
	int			j = 0;
	int				stuffPlace = 0;

	for (i = 0; i < 256; i++)
	{
		j = (numbers[i] + ((unsigned char *) theStuff)[stuffPlace] + j);
		j &= 0xff;

		unsigned char t = numbers[i];					// Swap the
appropriate number
		numbers[i] = numbers[j];
		numbers[j] = t;

		stuffPlace = (stuffPlace + 1) % stuffLen;
	}

	for (i = 0; i < 256; i++)  // Turn the crank through a complete
		Turn();            // revolution to thoroughly mix up the
				   // numbers. Studies show that this lessens
				   // the effect of weakly random stuff.
}

unsigned char Grinder::Turn()
{
	// The actual work is done here for the second stage. This is what
	// pops off random bytes. We spin this when we add more stuff to
	// stir it up.

	register unsigned char t;

	// We're going to swap two cells of stuff and numbers. We walk around the
	// array, picking the next spot. The place we're going to swap it with
	// is selected with the statements below:

	newPlace = newPlace + numbers[numberPlace];
```

```
        newPlace &= 0xff;   // mod 256

        t = numbers[numberPlace];                       // Swap number cells
        numbers[numberPlace] = numbers[newPlace];
        numbers[newPlace] = t;

        t += numbers[numberPlace];                      // Add in the other cell
        t &= 0xff;

        numberPlace = (numberPlace + 1) & 0xff; // Find the next cell to play with

        return numbers[t];
}

void Grinder::GetRandom(void *thePlace, long length)
{
        unsigned char *x = (unsigned char *) thePlace;

        for (int i = 0; i < length; i++)                // Just step out N bytes
                *x++ = Turn();
}


unsigned char Randomizer::Byte()
{
        return Turn();
}

unsigned short Randomizer::Word()
{
        return (Turn() << 8) | Turn();
}

unsigned long Randomizer::Long()
{
        return (Turn() << 24) |
                     (Turn() << 16) |
                     (Turn() <<  8) |
                     Turn();
}

void Randomizer::AddSeed(void *seed, int seedLen, int entropyGuessInDecibits)
{
        if (entropyGuessInDecibits < 0)
                entropyGuessInDecibits = TOTALENTROPY;

        AddData(seed, seedLen, entropyGuessInDecibits);

        if (GetEntropy() > TOTALENTROPY)
        {
                unsigned char hash[HASHLEN];

                GetHash(hash);
                AddStuff(hash, HASHLEN);

                SetEntropy(GetEntropy() - TOTALENTROPY);
        }
}

Randomizer::Randomizer()
{
        busy = 0;
}
```

```
void Randomizer::GetState(RandomState *state)
{
	state->distillerEntropy	= GetEntropy();
	state->numberPlace		= numberPlace;
	state->newPlace			= newPlace;
	state->still			= shs;

	for (int i = 0; i < 256; i++)
		state->numbers[i] = numbers[i];
}

void Randomizer::SetState(RandomState *state)
{
	SetEntropy(state->distillerEntropy);
	numberPlace		= state->numberPlace;
	newPlace		= state->newPlace;
	shs			= state->still;

	for (int i = 0; i < 256; i++)
		numbers[i] = state->numbers[i];
}
```

Random.h

```
#ifndef RANDOM_H
#define RANDOM_H 1
//
/ Copyright © 1996, Jon Callas, Eldacur Technologies. All Rights Reserved.
//
// You have permission to use this code in any commercial or non-commercial
// programs provided you do the following:
//
// (1) Somewhere in your documentation, about box, etc. you state that you use Jon
//     Callas's Two-Stage Random Number Generator and that it is copyright by him.
// (2) If you use it in a commercial product, you send me email at
//     jon@worldbenders.com telling me that you're using it. I don't have any
//     nefarious purposes, I'm not going to sell your name to some mailing list, I
//     just want to know who is using it, so I can send updates to you and be able
//     to brag about the bizillion people that use this.
// (3) If you find bugs, make substantive improvements, etc. that you make a best-
//     faith effort to send them to me so I can incorporate it into the code.
//
// That's it. That's all you have to do. It's not much to ask. It's not as odious
// as a copyleft so why not do it?
//

#include "shs.h"

const int HASHLEN = 20;
const int TOTALENTROPY = (HASHLEN * 8 * 10);

typedef struct
{
	long		distillerEntropy;
	long		numberPlace;
	long		newPlace;
	long		beenFullySeeded;
	SHS_CTX		still;
	unsigned char	numbers[256];

} RandomState;

class Distiller
{
protected:
	SHS_CTX		shs;
	long		totalEntropy;

public:
	Distiller();

	int		AddData(void *start, long length, int decibits);
	void		GetHash(unsigned char hash[HASHLEN]);
	void		Init();
	void		SetEntropy(long decibits)	{ totalEntropy = decibits; };
	int		GetEntropy(void)	{ return totalEntropy; };
};

class Grinder
{
protected:
	unsigned char	numbers[256];
	long		numberPlace;
	long		newPlace;
	void		InitNums();

public:
	Grinder();
	Grinder(void *theStuff, long stuffLen);

	unsigned char	Turn();
	void		AddStuff(void *theStuff, long stuffLen);

	void		GetRandom(void *thePlace, long length);
};


class Randomizer : public Distiller, public Grinder
{
protected:
	long		busy;		// Not presently used -- for future expansion
	void		Busy()		{ busy = 1; };
	void		NotBusy()	{ busy = 0; };

public:
	Randomizer();
	Randomizer(void *seed, int seedLen);

	void		AddSeed(void *seed, int seedLen, int entropyGuessInDecibits);
	unsigned char	Byte();
	unsigned short	Word();
	unsigned long	Long();
	void		String(void *start, long byteLen) { GetRandom(start, byteLen); };
	long		isBusy() { return busy; };
	void		GetState(RandomState *state);
	void		SetState(RandomState *state);
};

#endif
```