Network Working Group
Request for Comments: 1750
Category: Informational

D. Eastlake, 3rd
DEC
S. Crocker
Cybercash
J. Schiller
MIT

December 1994

Randomness Recommendations for Security

Status of this Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Abstract

Security systems today are built on increasingly strong cryptographic algorithms that foil pattern analysis attempts. However, the security of these systems is dependent on generating secret quantities for passwords, cryptographic keys, and similar quantities. The use of pseudo-random processes to generate secret quantities can result in pseudo-security. The sophisticated attacker of these security systems may find it easier to reproduce the environment that produced the secret quantities, searching the resulting small set of possibilities, than to locate the quantities in the whole of the number space.

Choosing random quantities to foil a resourceful and motivated adversary is surprisingly difficult. This paper points out many pitfalls in using traditional pseudo-random number generation techniques for choosing such quantities. It recommends the use of truly random hardware techniques and shows that the existing hardware on many systems can be used for this purpose. It provides suggestions to ameliorate the problem when a hardware solution is not available. And it gives examples of how large such quantities need to be for some particular applications.

Acknowledgements

Comments on this document that have been incorporated were received from (in alphabetic order) the following:

David M. Balenson (TIS)
Don Coppersmith (IBM)
Don T. Davis (consultant)
Carl Ellison (Stratus)
Marc Horowitz (MIT)
Christian Huitema (INRIA)
Charlie Kaufman (IRIS)
Steve Kent (BBN)
Hal Murray (DEC)
Neil Haller (Bellcore)
Richard Pitkin (DEC)
Tim Redmond (TIS)
Doug Tygar (CMU)

Table of Contents

1. Introduction

Software cryptography is coming into wider use. Systems like Kerberos, PEM, PGP, etc. are maturing and becoming a part of the network landscape [PEM]. These systems provide substantial protection against snooping and spoofing. However, there is a potential flaw. At the heart of all cryptographic systems is the generation of secret, unguessable (i.e., random) numbers.

For the present, the lack of generally available facilities for generating such unpredictable numbers is an open wound in the design of cryptographic software. For the software developer who wants to build a key or password generation procedure that runs on a wide range of hardware, the only safe strategy so far has been to force the local installation to supply a suitable routine to generate random numbers. To say the least, this is an awkward, error-prone and unpalatable solution.

It is important to keep in mind that the requirement is for data that an adversary has a very low probability of guessing or determining. This will fail if pseudo-random data is used which only meets traditional statistical tests for randomness or which is based on limited range sources, such as clocks. Frequently such random quantities are determinable by an adversary searching through an embarrassingly small space of possibilities.

This informational document suggests techniques for producing random quantities that will be resistant to such attack. It recommends that future systems include hardware random number generation or provide access to existing hardware that can be used for this purpose. It suggests methods for use if such hardware is not available. And it gives some estimates of the number of random bits required for sample

applications.

2. Requirements

Probably the most commonly encountered randomness requirement today is the user password. This is usually a simple character string. Obviously, if a password can be guessed, it does not provide security. (For re-usable passwords, it is desirable that users be able to remember the password. This may make it advisable to use pronounceable character strings or phrases composed on ordinary words. But this only affects the format of the password information, not the requirement that the password be very hard to guess.)

Many other requirements come from the cryptographic arena. Cryptographic techniques can be used to provide a variety of services including confidentiality and authentication. Such services are based on quantities, traditionally called "keys", that are unknown to and unguessable by an adversary.

In some cases, such as the use of symmetric encryption with the one time pads [CRYPTO*] or the US Data Encryption Standard [DES], the parties who wish to communicate confidentially and/or with authentication must all know the same secret key. In other cases, using what are called asymmetric or "public key" cryptographic techniques, keys come in pairs. One key of the pair is private and must be kept secret by one party, the other is public and can be published to the world. It is computationally infeasible to determine the private key from the public key [ASYMMETRIC, CRYPTO*].

The frequency and volume of the requirement for random quantities differs greatly for different cryptographic systems. Using pure RSA [CRYPTO*], random quantities are required when the key pair is generated, but thereafter any number of messages can be signed without any further need for randomness. The public key Digital Signature Algorithm that has been proposed by the US National Institute of Standards and Technology (NIST) requires good random numbers for each signature. And encrypting with a one time pad, in principle the strongest possible encryption technique, requires a volume of randomness equal to all the messages to be processed.

In most of these cases, an adversary can try to determine the "secret" key by trial and error. (This is possible as long as the key is enough smaller than the message that the correct key can be uniquely identified.) The probability of an adversary succeeding at this must be made acceptably low, depending on the particular application. The size of the space the adversary must search is related to the amount of key "information" present in the information theoretic sense [SHANNON]. This depends on the number of different

secret values possible and the probability of each value as follows:

$$\text{Bits-of-info} = \sum_{i} - p_i * \log_2 (p_i)$$

where i varies from 1 to the number of possible secret values and p sub i is the probability of the value numbered i. (Since p sub i is less than one, the log will be negative so each term in the sum will be non-negative.)

If there are 2^n different values of equal probability, then n bits of information are present and an adversary would, on the average, have to try half of the values, or 2^(n-1) , before guessing the secret quantity. If the probability of different values is unequal, then there is less information present and fewer guesses will, on average, be required by an adversary. In particular, any values that the adversary can know are impossible, or are of low probability, can be initially ignored by an adversary, who will search through the more probable values first.

For example, consider a cryptographic system that uses 56 bit keys. If these 56 bit keys are derived by using a fixed pseudo-random number generator that is seeded with an 8 bit seed, then an adversary needs to search through only 256 keys (by running the pseudo-random number generator with every possible seed), not the 2^56 keys that may at first appear to be the case. Only 8 bits of "information" are in these 56 bit keys.

3. Traditional Pseudo-Random Sequences

Most traditional sources of random numbers use deterministic sources of "pseudo-random" numbers. These typically start with a "seed" quantity and use numeric or logical operations to produce a sequence of values.

[KNUTH] has a classic exposition on pseudo-random numbers. Applications he mentions are simulation of natural phenomena, sampling, numerical analysis, testing computer programs, decision making, and games. None of these have the same characteristics as the sort of security uses we are talking about. Only in the last two could there be an adversary trying to find the random quantity. However, in these cases, the adversary normally has only a single chance to use a guessed value. In guessing passwords or attempting to break an encryption scheme, the adversary normally has many,

perhaps unlimited, chances at guessing the correct value and should be assumed to be aided by a computer.

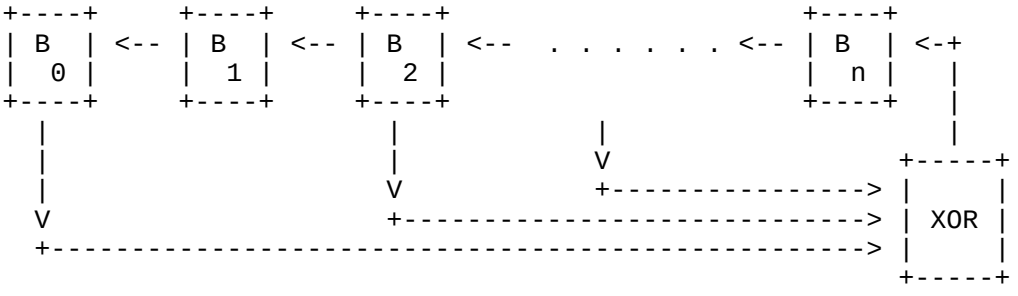
For testing the "randomness" of numbers, Knuth suggests a variety of measures including statistical and spectral. These tests check things like autocorrelation between different parts of a "random" sequence or distribution of its values. They could be met by a constant stored random sequence, such as the "random" sequence printed in the CRC Standard Mathematical Tables [CRC].

A typical pseudo-random number generation technique, known as a linear congruence pseudo-random number generator, is modular arithmetic where the N+1th value is calculated from the Nth value by

$$V_{N+1} = (V_N * a + b)(Mod\ c)$$

The above technique has a strong relationship to linear shift register pseudo-random number generators, which are well understood cryptographically [SHIFT*]. In such generators bits are introduced at one end of a shift register as the Exclusive Or (binary sum without carry) of bits from selected fixed taps into the register.

For example:



$$V_{N+1} = ((V_N * 2) + B_0 .xor. B_2 ...)(Mod\ 2^n)$$

The goodness of traditional pseudo-random number generator algorithms is measured by statistical tests on such sequences. Carefully chosen values of the initial V and a, b, and c or the placement of shift register tap in the above simple processes can produce excellent statistics.

These sequences may be adequate in simulations (Monte Carlo experiments) as long as the sequence is orthogonal to the structure of the space being explored. Even there, subtle patterns may cause problems. However, such sequences are clearly bad for use in security applications. They are fully predictable if the initial state is known. Depending on the form of the pseudo-random number generator, the sequence may be determinable from observation of a short portion of the sequence [CRYPTO*, STERN]. For example, with the generators above, one can determine V(n+1) given knowledge of V(n). In fact, it has been shown that with these techniques, even if only one bit of the pseudo-random values is released, the seed can be determined from short sequences.

Not only have linear congruent generators been broken, but techniques are now known for breaking all polynomial congruent generators [KRAWCZYK].

4. Unpredictability

Randomness in the traditional sense described in section 3 is NOT the same as the unpredictability required for security use.

For example, use of a widely available constant sequence, such as that from the CRC tables, is very weak against an adversary. Once they learn of or guess it, they can easily break all security, future and past, based on the sequence [CRC]. Yet the statistical properties of these tables are good.

The following sections describe the limitations of some randomness generation techniques and sources.

4.1 Problems with Clocks and Serial Numbers

Computer clocks, or similar operating system or hardware values, provide significantly fewer real bits of unpredictability than might appear from their specifications.

Tests have been done on clocks on numerous systems and it was found that their behavior can vary widely and in unexpected ways. One version of an operating system running on one set of hardware may actually provide, say, microsecond resolution in a clock while a different configuration of the "same" system may always provide the same lower bits and only count in the upper bits at much lower resolution. This means that successive reads on the clock may produce identical values even if enough time has passed that the value "should" change based on the nominal clock resolution. There are also cases where frequently reading a clock can produce artificial sequential values because of extra code that checks for

the clock being unchanged between two reads and increases it by one! Designing portable application code to generate unpredictable numbers based on such system clocks is particularly challenging because the system designer does not always know the properties of the system clocks that the code will execute on.

Use of a hardware serial number such as an Ethernet address may also provide fewer bits of uniqueness than one would guess. Such quantities are usually heavily structured and subfields may have only a limited range of possible values or values easily guessable based on approximate date of manufacture or other data. For example, it is likely that most of the Ethernet cards installed on Digital Equipment Corporation (DEC) hardware within DEC were manufactured by DEC itself, which significantly limits the range of built in addresses.

Problems such as those described above related to clocks and serial numbers make code to produce unpredictable quantities difficult if the code is to be ported across a variety of computer platforms and systems.

4.2 Timing and Content of External Events

It is possible to measure the timing and content of mouse movement, key strokes, and similar user events. This is a reasonable source of unguessable data with some qualifications. On some machines, inputs such as key strokes are buffered. Even though the user's inter-keystroke timing may have sufficient variation and unpredictability, there might not be an easy way to access that variation. Another problem is that no standard method exists to sample timing details. This makes it hard to build standard software intended for distribution to a large range of machines based on this technique.

The amount of mouse movement or the keys actually hit are usually easier to access than timings but may yield less unpredictability as the user may provide highly repetitive input.

Other external events, such as network packet arrival times, can also be used with care. In particular, the possibility of manipulation of such times by an adversary must be considered.

4.3 The Fallacy of Complex Manipulation

One strategy which may give a misleading appearance of unpredictability is to take a very complex algorithm (or an excellent traditional pseudo-random number generator with good statistical properties) and calculate a cryptographic key by starting with the current value of a computer system clock as the seed. An adversary who knew roughly when the generator was started would have a

relatively small number of seed values to test as they would know likely values of the system clock. Large numbers of pseudo-random bits could be generated but the search space an adversary would need to check could be quite small.

Thus very strong and/or complex manipulation of data will not help if the adversary can learn what the manipulation is and there is not enough unpredictability in the starting seed value. Even if they can not learn what the manipulation is, they may be able to use the limited number of results stemming from a limited number of seed values to defeat security.

Another serious strategy error is to assume that a very complex pseudo-random number generation algorithm will produce strong random numbers when there has been no theory behind or analysis of the algorithm. There is a excellent example of this fallacy right near the beginning of chapter 3 in [KNUTH] where the author describes a complex algorithm. It was intended that the machine language program corresponding to the algorithm would be so complicated that a person trying to read the code without comments wouldn't know what the program was doing. Unfortunately, actual use of this algorithm showed that it almost immediately converged to a single repeated value in one case and a small cycle of values in another case.

Not only does complex manipulation not help you if you have a limited range of seeds but blindly chosen complex manipulation can destroy the randomness in a good seed!

4.4 The Fallacy of Selection from a Large Database

Another strategy that can give a misleading appearance of unpredictability is selection of a quantity randomly from a database and assume that its strength is related to the total number of bits in the database. For example, typical USENET servers as of this date process over 35 megabytes of information per day. Assume a random quantity was selected by fetching 32 bytes of data from a random starting point in this data. This does not yield 32*8 = 256 bits worth of unguessability. Even after allowing that much of the data is human language and probably has more like 2 or 3 bits of information per byte, it doesn't yield 32*2.5 = 80 bits of unguessability. For an adversary with access to the same 35 megabytes the unguessability rests only on the starting point of the selection. That is, at best, about 25 bits of unguessability in this case.

The same argument applies to selecting sequences from the data on a CD ROM or Audio CD recording or any other large public database. If the adversary has access to the same database, this "selection from a

large volume of data" step buys very little. However, if a selection can be made from data to which the adversary has no access, such as system buffers on an active multi-user system, it may be of some help.

5. Hardware for Randomness

Is there any hope for strong portable randomness in the future? There might be. All that's needed is a physical source of unpredictable numbers.

A thermal noise or radioactive decay source and a fast, free-running oscillator would do the trick directly [GIFFORD]. This is a trivial amount of hardware, and could easily be included as a standard part of a computer system's architecture. Furthermore, any system with a spinning disk or the like has an adequate source of randomness [DAVIS]. All that's needed is the common perception among computer vendors that this small additional hardware and the software to access it is necessary and useful.

5.1 Volume Required

How much unpredictability is needed? Is it possible to quantify the requirement in, say, number of random bits per second?

The answer is not very much is needed. For DES, the key is 56 bits and, as we show in an example in Section 8, even the highest security system is unlikely to require a keying material of over 200 bits. If a series of keys are needed, it can be generated from a strong random seed using a cryptographically strong sequence as explained in Section 6.3. A few hundred random bits generated once a day would be enough using such techniques. Even if the random bits are generated as slowly as one per second and it is not possible to overlap the generation process, it should be tolerable in high security applications to wait 200 seconds occasionally.

These numbers are trivial to achieve. It could be done by a person repeatedly tossing a coin. Almost any hardware process is likely to be much faster.

5.2 Sensitivity to Skew

Is there any specific requirement on the shape of the distribution of the random numbers? The good news is the distribution need not be uniform. All that is needed is a conservative estimate of how non-uniform it is to bound performance. Two simple techniques to de-skew the bit stream are given below and stronger techniques are mentioned in Section 6.1.2 below.

5.2.1 Using Stream Parity to De-Skew

Consider taking a sufficiently long string of bits and map the string to "zero" or "one". The mapping will not yield a perfectly uniform distribution, but it can be as close as desired. One mapping that serves the purpose is to take the parity of the string. This has the advantages that it is robust across all degrees of skew up to the estimated maximum skew and is absolutely trivial to implement in hardware.

The following analysis gives the number of bits that must be sampled:

Suppose the ratio of ones to zeros is 0.5 + e : 0.5 - e, where e is between 0 and 0.5 and is a measure of the "eccentricity" of the distribution. Consider the distribution of the parity function of N bit samples. The probabilities that the parity will be one or zero will be the sum of the odd or even terms in the binomial expansion of (p + q)^N, where p = 0.5 + e, the probability of a one, and q = 0.5 - e, the probability of a zero.

These sums can be computed easily as

$$1/2 * ((p + q)^N + (p - q)^N)$$

and

$$1/2 * ((p + q)^N - (p - q)^N).$$

(Which one corresponds to the probability the parity will be 1 depends on whether N is odd or even.)

Since p + q = 1 and p - q = 2e, these expressions reduce to

$$1/2 * [1 + (2e)^N]$$

and

$$1/2 * [1 - (2e)^N].$$

Neither of these will ever be exactly 0.5 unless e is zero, but we can bring them arbitrarily close to 0.5. If we want the probabilities to be within some delta d of 0.5, i.e. then

$$(0.5 + (0.5 * (2e)^N)) < 0.5 + d.$$

Solving for N yields N > log(2d)/log(2e). (Note that 2e is less than 1, so its log is negative. Division by a negative number reverses the sense of an inequality.)

The following table gives the length of the string which must be sampled for various degrees of skew in order to come within 0.001 of a 50/50 distribution.

Prob(1)	e	N
0.5	0.00	1
0.6	0.10	4
0.7	0.20	7
0.8	0.30	13
0.9	0.40	28
0.95	0.45	59
0.99	0.49	308

The last entry shows that even if the distribution is skewed 99% in favor of ones, the parity of a string of 308 samples will be within 0.001 of a 50/50 distribution.

5.2.2 Using Transition Mappings to De-Skew

Another technique, originally due to von Neumann [VON NEUMANN], is to examine a bit stream as a sequence of non-overlapping pairs. You could then discard any 00 or 11 pairs found, interpret 01 as a 0 and 10 as a 1. Assume the probability of a 1 is 0.5+e and the probability of a 0 is 0.5-e where e is the eccentricity of the source and described in the previous section. Then the probability of each pair is as follows:

pair	probability
00	(0.5 - e)^2 = 0.25 - e + e^2
01	(0.5 - e)*(0.5 + e) = 0.25 - e^2
10	(0.5 + e)*(0.5 - e) = 0.25 - e^2
11	(0.5 + e)^2 = 0.25 + e + e^2

This technique will completely eliminate any bias but at the expense of taking an indeterminate number of input bits for any particular desired number of output bits. The probability of any particular pair being discarded is 0.5 + 2e^2 so the expected number of input bits to produce X output bits is X/(0.25 - e^2).

This technique assumes that the bits are from a stream where each bit has the same probability of being a 0 or 1 as any other bit in the stream and that bits are not correlated, i.e., that the bits are identical independent distributions. If alternate bits were from two correlated sources, for example, the above analysis breaks down.

The above technique also provides another illustration of how a simple statistical analysis can mislead if one is not always on the lookout for patterns that could be exploited by an adversary. If the algorithm were mis-read slightly so that overlapping successive bits pairs were used instead of non-overlapping pairs, the statistical analysis given is the same; however, instead of provided an unbiased uncorrelated series of random 1's and 0's, it instead produces a totally predictable sequence of exactly alternating 1's and 0's.

5.2.3 Using FFT to De-Skew

When real world data consists of strongly biased or correlated bits, it may still contain useful amounts of randomness. This randomness can be extracted through use of the discrete Fourier transform or its optimized variant, the FFT.

Using the Fourier transform of the data, strong correlations can be discarded. If adequate data is processed and remaining correlations decay, spectral lines approaching statistical independence and normally distributed randomness can be produced [BRILLINGER].

5.2.4 Using Compression to De-Skew

Reversible compression techniques also provide a crude method of de-skewing a skewed bit stream. This follows directly from the definition of reversible compression and the formula in Section 2 above for the amount of information in a sequence. Since the compression is reversible, the same amount of information must be present in the shorter output than was present in the longer input. By the Shannon information equation, this is only possible if, on average, the probabilities of the different shorter sequences are more uniformly distributed than were the probabilities of the longer sequences. Thus the shorter sequences are de-skewed relative to the input.

However, many compression techniques add a somewhat predicatable preface to their output stream and may insert such a sequence again periodically in their output or otherwise introduce subtle patterns of their own. They should be considered only a rough technique compared with those described above or in Section 6.1.2. At a minimum, the beginning of the compressed sequence should be skipped and only later bits used for applications requiring random bits.

5.3 Existing Hardware Can Be Used For Randomness

As described below, many computers come with hardware that can, with care, be used to generate truly random quantities.

5.3.1 Using Existing Sound/Video Input

Increasingly computers are being built with inputs that digitize some real world analog source, such as sound from a microphone or video input from a camera. Under appropriate circumstances, such input can provide reasonably high quality random bits. The "input" from a sound digitizer with no source plugged in or a camera with the lens cap on, if the system has enough gain to detect anything, is essentially thermal noise.

For example, on a SPARCstation, one can read from the /dev/audio device with nothing plugged into the microphone jack. Such data is essentially random noise although it should not be trusted without some checking in case of hardware failure. It will, in any case, need to be de-skewed as described elsewhere.

Combining this with compression to de-skew one can, in UNIXese, generate a huge amount of medium quality random data by doing

```
cat /dev/audio | compress - >random-bits-file
```

5.3.2 Using Existing Disk Drives

Disk drives have small random fluctuations in their rotational speed due to chaotic air turbulence [DAVIS]. By adding low level disk seek time instrumentation to a system, a series of measurements can be obtained that include this randomness. Such data is usually highly correlated so that significant processing is needed, including FFT (see section 5.2.3). Nevertheless experimentation has shown that, with such processing, disk drives easily produce 100 bits a minute or more of excellent random data.

Partly offsetting this need for processing is the fact that disk drive failure will normally be rapidly noticed. Thus, problems with this method of random number generation due to hardware failure are very unlikely.

6. Recommended Non-Hardware Strategy

What is the best overall strategy for meeting the requirement for unguessable random numbers in the absence of a reliable hardware source? It is to obtain random input from a large number of uncorrelated sources and to mix them with a strong mixing function.

Such a function will preserve the randomness present in any of the sources even if other quantities being combined are fixed or easily guessable. This may be advisable even with a good hardware source as hardware can also fail, though this should be weighed against any increase in the chance of overall failure due to added software complexity.

6.1 Mixing Functions

A strong mixing function is one which combines two or more inputs and produces an output where each output bit is a different complex non-linear function of all the input bits. On average, changing any input bit will change about half the output bits. But because the relationship is complex and non-linear, no particular output bit is guaranteed to change when any particular input bit is changed.

Consider the problem of converting a stream of bits that is skewed towards 0 or 1 to a shorter stream which is more random, as discussed in Section 5.2 above. This is simply another case where a strong mixing function is desired, mixing the input bits to produce a smaller number of output bits. The technique given in Section 5.2.1 of using the parity of a number of bits is simply the result of successively Exclusive Or'ing them which is examined as a trivial mixing function immediately below. Use of stronger mixing functions to extract more of the randomness in a stream of skewed bits is examined in Section 6.1.2.

6.1.1 A Trivial Mixing Function

A trivial example for single bit inputs is the Exclusive Or function, which is equivalent to addition without carry, as show in the table below. This is a degenerate case in which the one output bit always changes for a change in either input bit. But, despite its simplicity, it will still provide a useful illustration.

input 1	input 2	output
0	0	0
0	1	1
1	0	1
1	1	0

If inputs 1 and 2 are uncorrelated and combined in this fashion then the output will be an even better (less skewed) random bit than the inputs. If we assume an "eccentricity" e as defined in Section 5.2 above, then the output eccentricity relates to the input eccentricity

as follows:

$$e_{output} = 2 * e_{input\ 1} * e_{input\ 2}$$

Since e is never greater than 1/2, the eccentricity is always improved except in the case where at least one input is a totally skewed constant. This is illustrated in the following table where the top and left side values are the two input eccentricities and the entries are the output eccentricity:

e	0.00	0.10	0.20	0.30	0.40	0.50
0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.10	0.00	0.02	0.04	0.06	0.08	0.10
0.20	0.00	0.04	0.08	0.12	0.16	0.20
0.30	0.00	0.06	0.12	0.18	0.24	0.30
0.40	0.00	0.08	0.16	0.24	0.32	0.40
0.50	0.00	0.10	0.20	0.30	0.40	0.50

However, keep in mind that the above calculations assume that the inputs are not correlated. If the inputs were, say, the parity of the number of minutes from midnight on two clocks accurate to a few seconds, then each might appear random if sampled at random intervals much longer than a minute. Yet if they were both sampled and combined with xor, the result would be zero most of the time.

6.1.2 Stronger Mixing Functions

The US Government Data Encryption Standard [DES] is an example of a strong mixing function for multiple bit quantities. It takes up to 120 bits of input (64 bits of "data" and 56 bits of "key") and produces 64 bits of output each of which is dependent on a complex non-linear function of all input bits. Other strong encryption functions with this characteristic can also be used by considering them to mix all of their key and data input bits.

Another good family of mixing functions are the "message digest" or hashing functions such as The US Government Secure Hash Standard [SHS] and the MD2, MD4, MD5 [MD2, MD4, MD5] series. These functions all take an arbitrary amount of input and produce an output mixing all the input bits. The MD* series produce 128 bits of output and SHS produces 160 bits.

Although the message digest functions are designed for variable amounts of input, DES and other encryption functions can also be used to combine any number of inputs. If 64 bits of output is adequate, the inputs can be packed into a 64 bit data quantity and successive 56 bit keys, padding with zeros if needed, which are then used to successively encrypt using DES in Electronic Codebook Mode [DES MODES]. If more than 64 bits of output are needed, use more complex mixing. For example, if inputs are packed into three quantities, A, B, and C, use DES to encrypt A with B as a key and then with C as a key to produce the 1st part of the output, then encrypt B with C and then A for more output and, if necessary, encrypt C with A and then B for yet more output. Still more output can be produced by reversing the order of the keys given above to stretch things. The same can be done with the hash functions by hashing various subsets of the input data to produce multiple outputs. But keep in mind that it is impossible to get more bits of "randomness" out than are put in.

An example of using a strong mixing function would be to reconsider the case of a string of 308 bits each of which is biased 99% towards zero. The parity technique given in Section 5.2.1 above reduced this to one bit with only a 1/1000 deviance from being equally likely a zero or one. But, applying the equation for information given in Section 2, this 308 bit sequence has 5 bits of information in it. Thus hashing it with SHS or MD5 and taking the bottom 5 bits of the result would yield 5 unbiased random bits as opposed to the single bit given by calculating the parity of the string.

6.1.3 Diffie-Hellman as a Mixing Function

Diffie-Hellman exponential key exchange is a technique that yields a shared secret between two parties that can be made computationally infeasible for a third party to determine even if they can observe all the messages between the two communicating parties. This shared secret is a mixture of initial quantities generated by each of them [D-H]. If these initial quantities are random, then the shared secret contains the combined randomness of them both, assuming they are uncorrelated.

6.1.4 Using a Mixing Function to Stretch Random Bits

While it is not necessary for a mixing function to produce the same or fewer bits than its inputs, mixing bits cannot "stretch" the amount of random unpredictability present in the inputs. Thus four inputs of 32 bits each where there is 12 bits worth of unpredicatability (such as 4,096 equally probable values) in each input cannot produce more than 48 bits worth of unpredictable output. The output can be expanded to hundreds or thousands of bits by, for example, mixing with successive integers, but the clever adversary’s

search space is still 2^48 possibilities. Furthermore, mixing to fewer bits than are input will tend to strengthen the randomness of the output the way using Exclusive Or to produce one bit from two did above.

The last table in Section 6.1.1 shows that mixing a random bit with a constant bit with Exclusive Or will produce a random bit. While this is true, it does not provide a way to "stretch" one random bit into more than one. If, for example, a random bit is mixed with a 0 and then with a 1, this produces a two bit sequence but it will always be either 01 or 10. Since there are only two possible values, there is still only the one bit of original randomness.

6.1.5 Other Factors in Choosing a Mixing Function

For local use, DES has the advantages that it has been widely tested for flaws, is widely documented, and is widely implemented with hardware and software implementations available all over the world including source code available by anonymous FTP. The SHS and MD* family are younger algorithms which have been less tested but there is no particular reason to believe they are flawed. Both MD5 and SHS were derived from the earlier MD4 algorithm. They all have source code available by anonymous FTP [SHS, MD2, MD4, MD5].

DES and SHS have been vouched for the the US National Security Agency (NSA) on the basis of criteria that primarily remain secret. While this is the cause of much speculation and doubt, investigation of DES over the years has indicated that NSA involvement in modifications to its design, which originated with IBM, was primarily to strengthen it. No concealed or special weakness has been found in DES. It is almost certain that the NSA modification to MD4 to produce the SHS similarly strengthened the algorithm, possibly against threats not yet known in the public cryptographic community.

DES, SHS, MD4, and MD5 are royalty free for all purposes. MD2 has been freely licensed only for non-profit use in connection with Privacy Enhanced Mail [PEM]. Between the MD* algorithms, some people believe that, as with "Goldilocks and the Three Bears", MD2 is strong but too slow, MD4 is fast but too weak, and MD5 is just right.

Another advantage of the MD* or similar hashing algorithms over encryption algorithms is that they are not subject to the same regulations imposed by the US Government prohibiting the unlicensed export or import of encryption/decryption software and hardware. The same should be true of DES rigged to produce an irreversible hash code but most DES packages are oriented to reversible encryption.

6.2 Non-Hardware Sources of Randomness

The best source of input for mixing would be a hardware randomness such as disk drive timing affected by air turbulence, audio input with thermal noise, or radioactive decay. However, if that is not available there are other possibilities. These include system clocks, system or input/output buffers, user/system/hardware/network serial numbers and/or addresses and timing, and user input. Unfortunately, any of these sources can produce limited or predicatable values under some circumstances.

Some of the sources listed above would be quite strong on multi-user systems where, in essence, each user of the system is a source of randomness. However, on a small single user system, such as a typical IBM PC or Apple Macintosh, it might be possible for an adversary to assemble a similar configuration. This could give the adversary inputs to the mixing process that were sufficiently correlated to those used originally as to make exhaustive search practical.

The use of multiple random inputs with a strong mixing function is recommended and can overcome weakness in any particular input. For example, the timing and content of requested "random" user keystrokes can yield hundreds of random bits but conservative assumptions need to be made. For example, assuming a few bits of randomness if the inter-keystroke interval is unique in the sequence up to that point and a similar assumption if the key hit is unique but assuming that no bits of randomness are present in the initial key value or if the timing or key value duplicate previous values. The results of mixing these timings and characters typed could be further combined with clock values and other inputs.

This strategy may make practical portable code to produce good random numbers for security even if some of the inputs are very weak on some of the target systems. However, it may still fail against a high grade attack on small single user systems, especially if the adversary has ever been able to observe the generation process in the past. A hardware based random source is still preferable.

6.3 Cryptographically Strong Sequences

In cases where a series of random quantities must be generated, an adversary may learn some values in the sequence. In general, they should not be able to predict other values from the ones that they know.

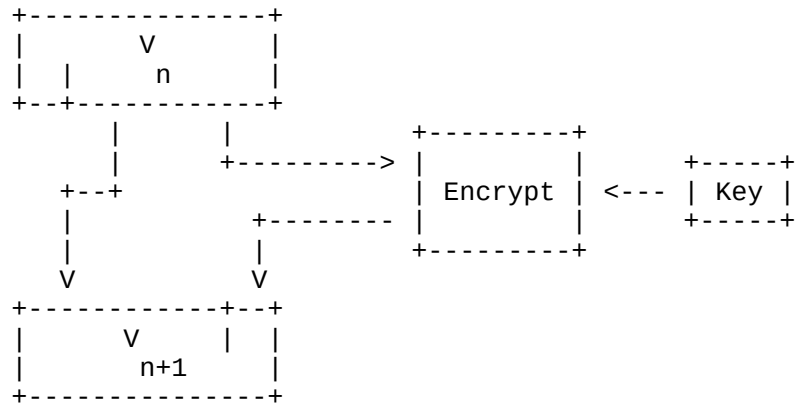
The correct technique is to start with a strong random seed, take cryptographically strong steps from that seed [CRYPTO2, CRYPTO3], and do not reveal the complete state of the generator in the sequence elements. If each value in the sequence can be calculated in a fixed way from the previous value, then when any value is compromised, all future values can be determined. This would be the case, for example, if each value were a constant function of the previously used values, even if the function were a very strong, non-invertible message digest function.

It should be noted that if your technique for generating a sequence of key values is fast enough, it can trivially be used as the basis for a confidentiality system. If two parties use the same sequence generating technique and start with the same seed material, they will generate identical sequences. These could, for example, be xor'ed at one end with data being send, encrypting it, and xor'ed with this data as received, decrypting it due to the reversible properties of the xor operation.

6.3.1 Traditional Strong Sequences

A traditional way to achieve a strong sequence has been to have the values be produced by hashing the quantities produced by concatenating the seed with successive integers or the like and then mask the values obtained so as to limit the amount of generator state available to the adversary.

It may also be possible to use an "encryption" algorithm with a random key and seed value to encrypt and feedback some or all of the output encrypted value into the value to be encrypted for the next iteration. Appropriate feedback techniques will usually be recommended with the encryption algorithm. An example is shown below where shifting and masking are used to combine the cypher output feedback. This type of feedback is recommended by the US Government in connection with DES [DES MODES].



Note that if a shift of one is used, this is the same as the shift register technique described in Section 3 above but with the all important difference that the feedback is determined by a complex non-linear function of all bits rather than a simple linear or polynomial combination of output from a few bit position taps.

It has been shown by Donald W. Davies that this sort of shifted partial output feedback significantly weakens an algorithm compared will feeding all of the output bits back as input. In particular, for DES, repeated encrypting a full 64 bit quantity will give an expected repeat in about 2^63 iterations. Feeding back anything less than 64 (and more than 0) bits will give an expected repeat in between 2**31 and 2**32 iterations!

To predict values of a sequence from others when the sequence was generated by these techniques is equivalent to breaking the cryptosystem or inverting the "non-invertible" hashing involved with only partial information available. The less information revealed each iteration, the harder it will be for an adversary to predict the sequence. Thus it is best to use only one bit from each value. It has been shown that in some cases this makes it impossible to break a system even when the cryptographic system is invertible and can be broken if all of each generated value was revealed.

6.3.2 The Blum Blum Shub Sequence Generator

Currently the generator which has the strongest public proof of strength is called the Blum Blum Shub generator after its inventors [BBS]. It is also very simple and is based on quadratic residues. It’s only disadvantage is that is is computationally intensive compared with the traditional techniques give in 6.3.1 above. This is not a serious draw back if it is used for moderately infrequent purposes, such as generating session keys.

Simply choose two large prime numbers, say p and q, which both have the property that you get a remainder of 3 if you divide them by 4. Let n = p * q. Then you choose a random number x relatively prime to n. The initial seed for the generator and the method for calculating subsequent values are then

$$s_0 = (x^2)(\mathrm{Mod}\ n)$$

$$s_{i+1} = (s_i^2)(\mathrm{Mod}\ n)$$

You must be careful to use only a few bits from the bottom of each s. It is always safe to use only the lowest order bit. If you use no more than the

$$\log_2 (\log_2 (s_i))$$

low order bits, then predicting any additional bits from a sequence generated in this manner is provable as hard as factoring n. As long as the initial x is secret, you can even make n public if you want.

An intersting characteristic of this generator is that you can directly calculate any of the s values. In particular

$$s_i = (s_0 ^{((2^i)(\mathrm{Mod}\ ((p - 1) * (q - 1))))})(\mathrm{Mod}\ n)$$

This means that in applications where many keys are generated in this fashion, it is not necessary to save them all. Each key can be effectively indexed and recovered from that small index and the initial s and n.

7. Key Generation Standards

Several public standards are now in place for the generation of keys. Two of these are described below. Both use DES but any equally strong or stronger mixing function could be substituted.

7.1 US DoD Recommendations for Password Generation

The United States Department of Defense has specific recommendations for password generation [DoD]. They suggest using the US Data Encryption Standard [DES] in Output Feedback Mode [DES MODES] as follows:

use an initialization vector determined from
the system clock,
system ID,
user ID, and
date and time;
use a key determined from
system interrupt registers,
system status registers, and
system counters; and,
as plain text, use an external randomly generated 64 bit quantity such as 8 characters typed in by a system administrator.

The password can then be calculated from the 64 bit "cipher text" generated in 64-bit Output Feedback Mode. As many bits as are needed can be taken from these 64 bits and expanded into a pronounceable word, phrase, or other format if a human being needs to remember the password.

7.2 X9.17 Key Generation

The American National Standards Institute has specified a method for generating a sequence of keys as follows:

s_0 is the initial 64 bit seed

g_n is the sequence of generated 64 bit key quantities

k is a random key reserved for generating this key sequence

t is the time at which a key is generated to as fine a resolution as is available (up to 64 bits).

DES (K, Q) is the DES encryption of quantity Q with key K

```
g    = DES ( k, DES ( k, t ) .xor. s  )
 n                                   n

s    = DES ( k, DES ( k, t ) .xor. g  )
 n+1                                 n
```

If g sub n is to be used as a DES key, then every eighth bit should be adjusted for parity for that use but the entire 64 bit unmodified g should be used in calculating the next s.

8. Examples of Randomness Required

Below are two examples showing rough calculations of needed randomness for security. The first is for moderate security passwords while the second assumes a need for a very high security cryptographic key.

8.1 Password Generation

Assume that user passwords change once a year and it is desired that the probability that an adversary could guess the password for a particular account be less than one in a thousand. Further assume that sending a password to the system is the only way to try a password. Then the crucial question is how often an adversary can try possibilities. Assume that delays have been introduced into a system so that, at most, an adversary can make one password try every six seconds. That's 600 per hour or about 15,000 per day or about 5,000,000 tries in a year. Assuming any sort of monitoring, it is unlikely someone could actually try continuously for a year. In fact, even if log files are only checked monthly, 500,000 tries is more plausible before the attack is noticed and steps taken to change passwords and make it harder to try more passwords.

To have a one in a thousand chance of guessing the password in 500,000 tries implies a universe of at least 500,000,000 passwords or about 2^29. Thus 29 bits of randomness are needed. This can probably be achieved using the US DoD recommended inputs for password generation as it has 8 inputs which probably average over 5 bits of randomness each (see section 7.1). Using a list of 1000 words, the password could be expressed as a three word phrase (1,000,000,000 possibilities) or, using case insensitive letters and digits, six would suffice ((26+10)^6 = 2,176,782,336 possibilities).

For a higher security password, the number of bits required goes up. To decrease the probability by 1,000 requires increasing the universe of passwords by the same factor which adds about 10 bits. Thus to have only a one in a million chance of a password being guessed under the above scenario would require 39 bits of randomness and a password

that was a four word phrase from a 1000 word list or eight letters/digits. To go to a one in 10^9 chance, 49 bits of randomness are needed implying a five word phrase or ten letter/digit password.

In a real system, of course, there are also other factors. For example, the larger and harder to remember passwords are, the more likely users are to write them down resulting in an additional risk of compromise.

8.2 A Very High Security Cryptographic Key

Assume that a very high security key is needed for symmetric encryption / decryption between two parties. Assume an adversary can observe communications and knows the algorithm being used. Within the field of random possibilities, the adversary can try key values in hopes of finding the one in use. Assume further that brute force trial of keys is the best the adversary can do.

8.2.1 Effort per Key Trial

How much effort will it take to try each key? For very high security applications it is best to assume a low value of effort. Even if it would clearly take tens of thousands of computer cycles or more to try a single key, there may be some pattern that enables huge blocks of key values to be tested with much less effort per key. Thus it is probably best to assume no more than a couple hundred cycles per key. (There is no clear lower bound on this as computers operate in parallel on a number of bits and a poor encryption algorithm could allow many keys or even groups of keys to be tested in parallel. However, we need to assume some value and can hope that a reasonably strong algorithm has been chosen for our hypothetical high security task.)

If the adversary can command a highly parallel processor or a large network of work stations, 2*10^10 cycles per second is probably a minimum assumption for availability today. Looking forward just a couple years, there should be at least an order of magnitude improvement. Thus assuming 10^9 keys could be checked per second or 3.6*10^11 per hour or 6*10^13 per week or 2.4*10^14 per month is reasonable. This implies a need for a minimum of 51 bits of randomness in keys to be sure they cannot be found in a month. Even then it is possible that, a few years from now, a highly determined and resourceful adversary could break the key in 2 weeks (on average they need try only half the keys).

8.2.2 Meet in the Middle Attacks

If chosen or known plain text and the resulting encrypted text are available, a "meet in the middle" attack is possible if the structure of the encryption algorithm allows it. (In a known plain text attack, the adversary knows all or part of the messages being encrypted, possibly some standard header or trailer fields. In a chosen plain text attack, the adversary can force some chosen plain text to be encrypted, possibly by "leaking" an exciting text that would then be sent by the adversary over an encrypted channel.)

An oversimplified explanation of the meet in the middle attack is as follows: the adversary can half-encrypt the known or chosen plain text with all possible first half-keys, sort the output, then half-decrypt the encoded text with all the second half-keys. If a match is found, the full key can be assembled from the halves and used to decrypt other parts of the message or other messages. At its best, this type of attack can halve the exponent of the work required by the adversary while adding a large but roughly constant factor of effort. To be assured of safety against this, a doubling of the amount of randomness in the key to a minimum of 102 bits is required.

The meet in the middle attack assumes that the cryptographic algorithm can be decomposed in this way but we can not rule that out without a deep knowledge of the algorithm. Even if a basic algorithm is not subject to a meet in the middle attack, an attempt to produce a stronger algorithm by applying the basic algorithm twice (or two different algorithms sequentially) with different keys may gain less added security than would be expected. Such a composite algorithm would be subject to a meet in the middle attack.

Enormous resources may be required to mount a meet in the middle attack but they are probably within the range of the national security services of a major nation. Essentially all nations spy on other nations government traffic and several nations are believed to spy on commercial traffic for economic advantage.

8.2.3 Other Considerations

Since we have not even considered the possibilities of special purpose code breaking hardware or just how much of a safety margin we want beyond our assumptions above, probably a good minimum for a very high security cryptographic key is 128 bits of randomness which implies a minimum key length of 128 bits. If the two parties agree on a key by Diffie-Hellman exchange [D-H], then in principle only half of this randomness would have to be supplied by each party. However, there is probably some correlation between their random inputs so it is probably best to assume that each party needs to

provide at least 96 bits worth of randomness for very high security if Diffie-Hellman is used.

This amount of randomness is beyond the limit of that in the inputs recommended by the US DoD for password generation and could require user typing timing, hardware random number generation, or other sources.

It should be noted that key length calculations such at those above are controversial and depend on various assumptions about the cryptographic algorithms in use. In some cases, a professional with a deep knowledge of code breaking techniques and of the strength of the algorithm in use could be satisfied with less than half of the key size derived above.

9. Conclusion

Generation of unguessable "random" secret quantities for security use is an essential but difficult task.

We have shown that hardware techniques to produce such randomness would be relatively simple. In particular, the volume and quality would not need to be high and existing computer hardware, such as disk drives, can be used. Computational techniques are available to process low quality random quantities from multiple sources or a larger quantity of such low quality input from one source and produce a smaller quantity of higher quality, less predictable key material. In the absence of hardware sources of randomness, a variety of user and software sources can frequently be used instead with care; however, most modern systems already have hardware, such as disk drives or audio input, that could be used to produce high quality randomness.

Once a sufficient quantity of high quality seed key material (a few hundred bits) is available, strong computational techniques are available to produce cryptographically strong sequences of unpredicatable quantities from this seed material.

10. Security Considerations

The entirety of this document concerns techniques and recommendations for generating unguessable "random" quantities for use as passwords, cryptographic keys, and similar security uses.

References

[ASYMMETRIC] - Secure Communications and Asymmetric Cryptosystems, edited by Gustavus J. Simmons, AAAS Selected Symposium 69, Westview Press, Inc.

[BBS] - A Simple Unpredictable Pseudo-Random Number Generator, SIAM Journal on Computing, v. 15, n. 2, 1986, L. Blum, M. Blum, & M. Shub.

[BRILLINGER] - Time Series: Data Analysis and Theory, Holden-Day, 1981, David Brillinger.

[CRC] - C.R.C. Standard Mathematical Tables, Chemical Rubber Publishing Company.

[CRYPTO1] - Cryptography: A Primer, A Wiley-Interscience Publication, John Wiley & Sons, 1981, Alan G. Konheim.

[CRYPTO2] - Cryptography: A New Dimension in Computer Data Security, A Wiley-Interscience Publication, John Wiley & Sons, 1982, Carl H. Meyer & Stephen M. Matyas.

[CRYPTO3] - Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 1994, Bruce Schneier.

[DAVIS] - Cryptographic Randomness from Air Turbulence in Disk Drives, Advances in Cryptology - Crypto '94, Springer-Verlag Lecture Notes in Computer Science #839, 1984, Don Davis, Ross Ihaka, and Philip Fenstermacher.

[DES] - Data Encryption Standard, United States of America, Department of Commerce, National Institute of Standards and Technology, Federal Information Processing Standard (FIPS) 46-1.
- Data Encryption Algorithm, American National Standards Institute, ANSI X3.92-1981.
(See also FIPS 112, Password Usage, which includes FORTRAN code for performing DES.)

[DES MODES] - DES Modes of Operation, United States of America, Department of Commerce, National Institute of Standards and Technology, Federal Information Processing Standard (FIPS) 81.
- Data Encryption Algorithm - Modes of Operation, American National Standards Institute, ANSI X3.106-1983.

[D-H] - New Directions in Cryptography, IEEE Transactions on Information Technology, November, 1976, Whitfield Diffie and Martin E. Hellman.

[DoD] - Password Management Guideline, United States of America, Department of Defense, Computer Security Center, CSC-STD-002-85. (See also FIPS 112, Password Usage, which incorporates CSC-STD-002-85 as one of its appendices.)

[GIFFORD] - Natural Random Number, MIT/LCS/TM-371, September 1988, David K. Gifford

[KNUTH] - The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Chapter 3: Random Numbers. Addison Wesley Publishing Company, Second Edition 1982, Donald E. Knuth.

[KRAWCZYK] - How to Predict Congruential Generators, Journal of Algorithms, V. 13, N. 4, December 1992, H. Krawczyk

[MD2] - The MD2 Message-Digest Algorithm, RFC1319, April 1992, B. Kaliski
[MD4] - The MD4 Message-Digest Algorithm, RFC1320, April 1992, R. Rivest
[MD5] - The MD5 Message-Digest Algorithm, RFC1321, April 1992, R. Rivest

[PEM] - RFCs 1421 through 1424:
- RFC 1424, Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services, 02/10/1993, B. Kaliski
- RFC 1423, Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers, 02/10/1993, D. Balenson
- RFC 1422, Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management, 02/10/1993, S. Kent
- RFC 1421, Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures, 02/10/1993, J. Linn

[SHANNON] - The Mathematical Theory of Communication, University of Illinois Press, 1963, Claude E. Shannon. (originally from: Bell System Technical Journal, July and October 1948)

[SHIFT1] - Shift Register Sequences, Aegean Park Press, Revised Edition 1982, Solomon W. Golomb.

[SHIFT2] - Cryptanalysis of Shift-Register Generated Stream Cypher Systems, Aegean Park Press, 1984, Wayne G. Barker.

[SHS] - Secure Hash Standard, United States of American, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180, April 1993.

[STERN] - Secret Linear Congruential Generators are not Cryptograhically Secure, Proceedings of IEEE STOC, 1987, J. Stern.

[VON NEUMANN] - Various techniques used in connection with random digits, von Neumann's Collected Works, Vol. 5, Pergamon Press, 1963, J. von Neumann.

Authors' Addresses

Donald E. Eastlake 3rd
Digital Equipment Corporation
550 King Street, LKG2-1/BB3
Littleton, MA 01460

Phone: +1 508 486 6577(w) +1 508 287 4877(h)
EMail: dee@lkg.dec.com

Stephen D. Crocker
CyberCash Inc.
2086 Hunters Crest Way
Vienna, VA 22181

Phone: +1 703-620-1222(w) +1 703-391-2651 (fax)
EMail: crocker@cybercash.com

Jeffrey I. Schiller
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139

Phone: +1 617 253 0161(w)
EMail: jis@mit.edu