Pseudo-Randomness

(breaking) LCG

TABLE OF CONTENTS

BREAKING PSEUDO-RANDOMNESS?

C `rand()` with different **seeds**

$$
\begin{aligned}
1337 &\longrightarrow 292616681, \quad 1638893262, \quad 255706927, \ldots \\
5667 &\longrightarrow 1971409024, \quad 815969455, \quad 1253865160 \\
42 &\longrightarrow 71876166, \quad 708592740, \quad 1483128881
\end{aligned}
$$

This is the idea behind **PRNGs** and, more in general, **pseudo-randomness** and **pseudo-random sequences**:

- having a sequence of numbers that **looks** random
- yet it is completely determined by
  - an underlying **algorithm**
  - the initial **seed** value

Meaningful terms in the context of PRNGs

- **state**: total amount of memory that is used internally by the PRNG to generate the sequence of numbers.
- **period**: after how many numbers the PRNG resets to its **initial state**.

Not all about **looks**, even for PRNGs.

Good PRNGs satisfy specific **statistical properties**.

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

Said in another way…

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

Said in another way…

given an output of the PRNG, are we able to predict the next number?

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

Said in another way…

given an output of the PRNG, are we able to predict the next number?

$$x_n \longrightarrow \quad ?$$

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

- **Short answer**: No.

Q: Do basic PRNGs also satisfy security related **cryptographic properties**?

- **Short answer**: No.
- **Long answer**: No, and this is problematic…

LINEAR CONGRUENTIAL GENERATOR

A **Linear Congruential Generator** is defined by the following set of equations

$$\begin{cases} x_0 &= \text{seed} \\ x_n &= (x_{n-1} \cdot a + b) \mod c \end{cases}$$

where

- a, b, c are typically fixed
- seed changes on every restart

The state is initialized with the given seed, and it is then updated for generating each subsequent number.

$$\mathrm{seed} = x_0 \longrightarrow x_1 \longrightarrow x_2 \longrightarrow x_3 \longrightarrow \ldots \longrightarrow x_n$$

LCG IN RAND()'S GLIBC

Let's look at the LCG implemented in the code of the **standard C library** (libc), which is inserted into most binaries compiled with **gcc**.

The code can be download with

```
curl https://ftp.gnu.org/gnu/libc/glibc-2.36.tar.bz2 > glibc-2.36.tar.bz2
```

Initialization in _srandomr()

```c
int __srandom_r (unsigned int seed, struct random_data *buf) {
  int type;
  int32_t *state;
  // ...
  state = buf->state;
  // ...
  state[0] = seed;
  if (type == TYPE_0)
    goto done;
  // ...
}
```

(glibc/stdlib/random$_{r.c}$:161)

State update in _randomr()

```c
int __random_r (struct random_data *buf, int32_t *result) {
  // ...
  if (buf->rand_type == TYPE_0) {
    int32_t val = ((state[0] * 1103515245U) + 12345U) & 0x7fffffff;
    state[0] = val;
    *result = val;
  }
  // ...
}
```

(glibc/stdlib/random$_{r.c}$:353)

The main equation of the glibc LCG is

$$x_n = ((x_{n-1} \times 1103515245) + 12345) \quad \& \quad \texttt{0x7fffffff}$$

The main equation of the glibc LCG is

$$x_n = ((x_{n-1} \times 1103515245) + 12345) \quad \& \quad \texttt{0x7fffffff}$$

where

The main equation of the glibc LCG is

$$x_n = ((x_{n-1} \times 1103515245) + 12345) \quad \& \quad \texttt{0x7fffffff}$$

where

$$\texttt{0x7fffffff} = 2147483647$$

The main equation of the glibc LCG is

$$x_n = ((x_{n-1} \times 1103515245) + 12345) \quad \& \quad \texttt{0x7fffffff}$$

where

$$\texttt{0x7fffffff} = 2147483647$$

$$= \underbrace{01111111111111111111111111111111}_{32\ \text{bit}}$$

Note that

$$x \quad \& \quad 2147483647$$

is equivalent to

$$x \quad mod \quad 2147483648$$

(see **code/example-4-rand-equivalence.c**)

Remember the concepts of **period** and **state**…

- The LCG state in C **rand()** is made up of a single **32 bit** integer
- Thus it has a period of

$$2^{31} - 1 = 2147483647$$

(see **code/example-5-rand-lcg-period.c**)

Remember the concepts of **period** and **state**…

- The LCG state in C **rand()** is made up of a single **32 bit** integer
- Thus it has a period of

$$2^{31} - 1 = 2147483647$$

(see **code/example-5-rand-lcg-period.c**)

NOTE: why only $2^{31} - 1$ and not $2^{32} - 1$? Because the last bit is thrown away (ask the devs).

HOW TO BREAK LCG

Now that we know how a LCG works, we can begin to understand how to "break" it.

Remember that by "breaking a PRNG" we simply mean

being able to predict what's the next number in the sequence given some outputs obtained from the PRNG

$$x_1, x_2, \ldots, x_n \xrightarrow{?} x_{n+1}$$

Remember the main equation of the LCG

$$x_n = (x_{n-1} \cdot a + b) \mod c$$

and consider the following attack scenarios:

Remember the main equation of the LCG

$$x_n = (x_{n-1} \cdot a + b) \mod c$$

and consider the following attack scenarios:

1. We know all the parameters a, b and c

Remember the main equation of the LCG

$$x_n = (x_{n-1} \cdot a + b) \mod c$$

and consider the following attack scenarios:

1. We know all the parameters a, b and c
2. We know some of the parameters a, b and c

Remember the main equation of the LCG

$$x_n = (x_{n-1} \cdot a + b) \mod c$$

and consider the following attack scenarios:

1. We know all the parameters a, b and c
2. We know some of the parameters a, b and c
3. We don't know any of the parameters a, b and c

Remember the main equation of the LCG

$$x_n = (x_{n-1} \cdot a + b) \mod c$$

and consider the following attack scenarios:

1. We know all the parameters a, b and c
2. We know some of the parameters a, b and c
3. We don't know any of the parameters a, b and c

We'll cover how to deal with scenarios 1 and 3.

SCENARIO 1: WE KNOW ALL THE PARAMETERS

Scenario 1: We know all the parameters a, b and c

This scenario is easy.

Scenario $\mathbf{1}$: We know all the parameters a, b and c

This scenario is easy.

Why?

Scenario 1: We know all the parameters a, b and c

Let $x_1, x_2, \ldots, x_n$ be a sequence of observed outputs from the PRNG. Then the next output is obtained by simply using the main LCG equation

$$x_{n+1} = (x_n \cdot a + b) \mod c$$

For example, assuming

$$a = 1103515245 \quad , \quad b = 12345 \quad , \quad c = 2147483648$$

if we get an output $x_n = 1337$ the next output will be

$$\begin{aligned} x_{n+1} &= (1337 \cdot 1103515245 + 12345) \mod 2147483\text{6} \\ &= 78628734 \end{aligned}$$

SCENARIO 2: WE DON'T KNOW ANY OF THE PARAMETERS

Scenario 2: We don't know the parameters a, b and c

This scenario is a bit more involved.

The attack we'll discuss is based on a cool property of **number theory**.

There are also other roads to attack LCGs, following the research published by **George Marsaglia** in 1968

RANDOM NUMBERS FALL MAINLY IN THE PLANES

By George Marsaglia

MATHEMATICS RESEARCH LABORATORY, BOEING SCIENTIFIC RESEARCH LABORATORIES, SEATTLE, WASHINGTON

Communicated by G. S. Schairer, June 24, 1968

Virtually all the world's computer centers use an arithmetic procedure for generating random numbers. The most common of these is the multiplicative congruential generator first suggested by D. H. Lehmer. In this method, one merely multiplies the current random integer I by a constant multiplier K and keeps the remainder after overflow:

$$\text{new } I = K \times \text{old } I \quad \text{modulo } M.$$

Article

We can sketch the general idea behind the attack:

We can sketch the general idea behind the attack:

- We first observe an output sequence $x_0, x_1, \ldots, x_n$.

We can sketch the general idea behind the attack:

- We first observe an output sequence $x_0, x_1, \ldots, x_n$.
- Then we compute the modulus c

We can sketch the general idea behind the attack:

- We first observe an output sequence $x_0, x_1, \ldots, x_n$.
- Then we compute the modulus c
- Then we compute the multiplier a

We can sketch the general idea behind the attack:

- We first observe an output sequence $x_0, x_1, \ldots, x_n$.
- Then we compute the modulus c
- Then we compute the multiplier a
- Then we compute the increment b

Step 1/3: Computing the modulus c

Computing c (1/11)

Let $x_0, x_1, \ldots, x_n$ be the observed sequence of outputs. We define

$$t_n := x_{n+1} - x_n \qquad , \quad n = 0, \ldots, n-1$$

$$u_n := |t_{n+2} \cdot t_n - t_{n+1}^2| \quad , \quad n = 0, \ldots, n-3$$

Computing c (2/11)

Then with **high probability** we have that

$$c = \gcd(u_1, u_2, u_3, \ldots, u_{n-3})$$

where

$$\gcd \longrightarrow \text{Greatest Common Divisor}$$

Computing c (3/11)

Code to compute the modulus c

```
def compute_modulus(outputs):
    ts = []
    for i in range(0, len(outputs) - 1):
        ts.append(outputs[i+1] - outputs[i])

    us = []
    for i in range(0, len(ts)-2):
        us.append(abs(ts[i+2]*ts[i] - ts[i+1]**2))

    modulus =  reduce(math.gcd, us) #!
    return modulus
```

(code/example-6-attack-lcg.py)

Computing c (4/11)

Q: Why does that even work?

Computing c (5/11)

Remember how we defined t_n

$$
\begin{aligned}
t_n &= x_{n+1} - x_n \\
&= (x_n \cdot a + b) - (x_{n-1} \cdot a + b) \mod c \\
&= x_n \cdot a - x_{n-1} \cdot a \mod c \\
&= (x_n - x_{n-1}) \cdot a \mod c \\
&= t_{n-1} \cdot a \mod c
\end{aligned}
$$

Computing c (6/11)

Thus we get

$$t_{n+2} = t_n \cdot a^2 \mod c$$

Computing c (7/11)

This means that

$$
\begin{aligned}
t_{n+2} \cdot t_n - t_{n+1}^2 &= (t_n \cdot a^2) \cdot t_n - (t_n \cdot a)^2 \mod c \\
&= (t_n \cdot a)^2 - (t_n \cdot a)^2 \mod c \\
&= 0 \mod c
\end{aligned}
$$

Computing c (8/11)

Therefore $\exists k \in \mathbb{Z}$ such that

$$u_n = |t_{n+2} \cdot t_n - t_{n+1}^2| = |k \cdot c|$$

Computing c (8/11)

Therefore $\exists k \in Z$ such that

$$u_n = |t_{n+2} \cdot t_n - t_{n+1}^2| = |k \cdot c|$$

Said in another way

Computing c (8/11)

Therefore $\exists k \in Z$ such that

$$u_n = |t_{n+2} \cdot t_n - t_{n+1}^2| = |k \cdot c|$$

Said in another way

u_n is a multiple of c!

Computing c (9/11)

Ok, with this we now know we can compute a bunch of multiples of c starting from a sequence of outputs

$$\begin{aligned} x_0, x_1, \ldots, x_n &\longrightarrow t_0, t_1, \ldots, t_{n-1} \\ &\longrightarrow \underbrace{u_0, u_1, \ldots, u_{n-3}}_{\text{multiples of } c} \end{aligned}$$

Computing c (10/11)

And here comes the cool **number theory fact**:

Computing c (10/11)

And here comes the cool **number theory fact**:

The gcd of two random multiples of c will be c with probability

Computing c (10/11)

And here comes the cool **number theory fact**:

The gcd of two random multiples of c will be c with probability

$$\frac{6}{\pi^2} \approx 0.61$$

Computing c (11/11)

By taking the gcd of many random multiples of c, there is a very high probability that such gcd will be exactly c.

$$c = \gcd(u_1, u_2, u_3, \ldots, u_{n-3})$$

The more multiples we have, the higher the probability!

Step 2/3: Computing the multiplier a

Computing a (1/3)

Once we have the modulus c, we can obtain the multiplier a by observing that

$$\begin{cases} x_1 & = (x_0 \cdot a + b) \mod c \\ x_2 & = (x_1 \cdot a + b) \mod c \end{cases}$$

gives us

$$x_1 - x_2 = a \cdot (x_0 - x_1) \mod c$$

Computing a (2/3)

And from

$$x_1 - x_2 = a \cdot (x_0 - x_1) \mod c$$

we get

$$a = (x_1 - x_2) \cdot (x_0 - x_1)^{-1} \mod c$$

Computing a (3/3)

Code to compute the multiplier a

```
def compute_multiplier(outputs, modulus):
    term_1 = outputs[1] - outputs[2]
    term_2 = pow(outputs[0] - outputs[1], -1, modulus) #!
    a = (term_1 * term_2) % modulus
    return a
```

(code/example-6-attack-lcg.py)

Step 3/3: Computing the increment b

Computing b (1/2)

Finally, once we know c and a, we can easily obtain b

$$x_1 = (x_0 \cdot a + b) \mod c$$

$$\implies$$

$$b = (x_1 - x_0 \cdot a) \mod c$$

Computing b (1/2)

Code to compute the increment b

```python
def compute_increment(outputs, modulus, a):
    b = (outputs[1] - outputs[0] * a) % modulus
    return b
```

(code/example-6-attack-lcg.py)

Putting it all together

```
def main():
    prng = LCG(seed=1337, a=1103515245, b=12345, c=2147483648)
    n = 10
    outputs = []
    for i in range(0, n):
        outputs.append(prng.next())
    # -----------------------------
    c = compute_modulus(outputs)
    a = compute_multiplier(outputs, c)
    b = compute_increment(outputs, c, a)
    print(f"c={c}")
    print(f"a={a}")
    print(f"b={b}")
```

(code/example-6-attack-lcg.py)

We get

```
$ python3 example-6-attack-lcg.py
c=2147483648
a=1103515245
b=12345
```

$$c = 2147483648 \quad , \quad a = 1103515245 \quad , \quad b = 12345$$

LIVE DEMO

WAIT A SEC…

Let us implement a custom LCG in C with custom parameters

$$a = 2147483629$$
$$b = 2147483587$$
$$c = 2147483647$$

Custom LCG implementation (1/3)

```c
uint32_t a = 2147483629;
uint32_t b = 2147483587;
uint32_t c = 2147483647;
uint32_t state;

uint32_t myrand(void) {
  uint32_t val = ((state * a) + b) % c;
  state = val;
  return val;
}

void mysrand(uint32_t seed) {
  state = seed;
}
```

(code/example-7-custom-lcg.c)

Custom LCG implementation (2/3)

```
int main(void) {
  mysrand(1337);
  int n = 10;
  for (int i = 0; i < n; i++) {
    printf("%d\n", myrand());
  }

  return 0;
}
```

(code/example-7-custom-lcg.c)

Custom LCG implementation (3/3)

By executing it we get

```
gcc example-7_custom_lcg.c -o example-7_custom_lcg
```

```
[leo@archlinux code]$ ./example-7_custom_lcg
2147458185
483737
2138292585
174630137
976994632
764454763
507744979
1090263579
759828418
595645533
```

Now if we use **example-6**$_{\textbf{attacklcg.py}}$ to extract the parameters

```python
outputs = [2147458185, 483737, 2138292585, 174630137,
           976994632, 764454763, 507744979, 1090263579,
           759828418, 595645533]

c = compute_modulus(outputs)
a = compute_multiplier(outputs, c)
b = compute_increment(outputs, c, a)

print(f"c={c}")
print(f"a={a}")
print(f"b={b}")
```

We get

```
[leo@archlinux code]$ python3 example-6_attack_lcg.py
c=1
a=0
b=0
```

We get

```
[leo@archlinux code]$ python3 example-6_attack_lcg.py
c=1
a=0
b=0
```

Why did it fail?

We get

```
[leo@archlinux code]$ python3 example-6_attack_lcg.py
c=1
a=0
b=0
```

Why did it fail?

Did we break the math somehow?

The mathematical model on which our attack is based assumes to be working with the standard LCG formula

$$\begin{cases} x_0 & = \text{seed} \\ x_n & = (x_{n-1} \cdot a + b) \mod c \end{cases}$$

The mathematical model on which our attack is based assumes to be working with the standard LCG formula

$$\begin{cases} x_0 & = \text{seed} \\ x_n & = (x_{n-1} \cdot a + b) \mod c \end{cases}$$

Is this the case when working with C?

The mathematical model on which our attack is based assumes to be working with the standard LCG formula

$$\begin{cases} x_0 & = \text{seed} \\ x_n & = (x_{n-1} \cdot a + b) \mod c \end{cases}$$

Is this the case when working with C?

Someone said… what, overflows?

In C every datatype has a fixed number of bytes.

$$\begin{aligned}\text{uint32\_t} &\longrightarrow 4 \text{ bytes} \\ &\longrightarrow \underbrace{01010101101011100011101010111011}_{32 \text{ bits}}\end{aligned}$$

When all bytes of a given datatype (`uint32_t`) are used, an **overflow** happens.

$$4294967295 \longrightarrow \overbrace{11111111111111111111111111111111}^{\text{32 bits}}$$
$$4294967296 \longrightarrow 00000000000000000000000000000000$$

Overflows break our model

The correct model when dealing with overflows is the following one

$$
\begin{cases}
x_0 & = \text{seed} \ \wedge \ \texttt{0xFFFFFFFF} \\
x_n & = (((x_{n-1} \cdot a) \ \wedge \ \texttt{0xFFFFFFFF} + b) \\
 & \qquad\qquad \wedge \ \texttt{0xFFFFFFFF}) \mod c
\end{cases}
$$

When things break down, analyze your models.

(works in all aspects of life)