Breaking PRNGs for Fun and Profit

Part 1/2: On Linear Congruential Generators

LEONARDO TAMIANO

TABLE OF CONTENTS

INTRODUCTION

Hello.

$ WHOAMI

I'm Leonardo Tamiano, "supposedly" a PhD researcher here at Tor Vergata.

I work with professor Giuseppe Bianchi and I will be your teaching assistant for

Computer Network Security (CNS)

While prof. Bianchi will focus on the **theoretical aspects** of the subject matter, I will instead focus on the more **practical aspects**.

Q: What do you mean with "practical"?

Q: What do you mean with "practical"?

A: In these laboratories we will see

1. **vulnerable implementations** of software constructs
2. **why** they are vulnerable
3. **how to exploit** such vulnerabilities for fun and profit

My philosophy is:

1. To truly understand **how something works**, you have to **implement it**.
2. To truly understand **why something is vulnerable**, you have to **attack it**.

To this end we will use mainly the following programming languages

- Python
- C

Teaching material such as **slides**, **code**, **exercises** and general material can be found at the following URL

https://teaching.leonardotamiano.xyz/cns

For doubts and questions, I'm available after lectures.

Also, feel free to send me emails to the following email address

leonardotamiano95@gmail.com

But, please, put the following in the subject line

[CNS]

Before starting, remember

knowledge is power

use it wisely.

WHAT IS RANDOMNESS?

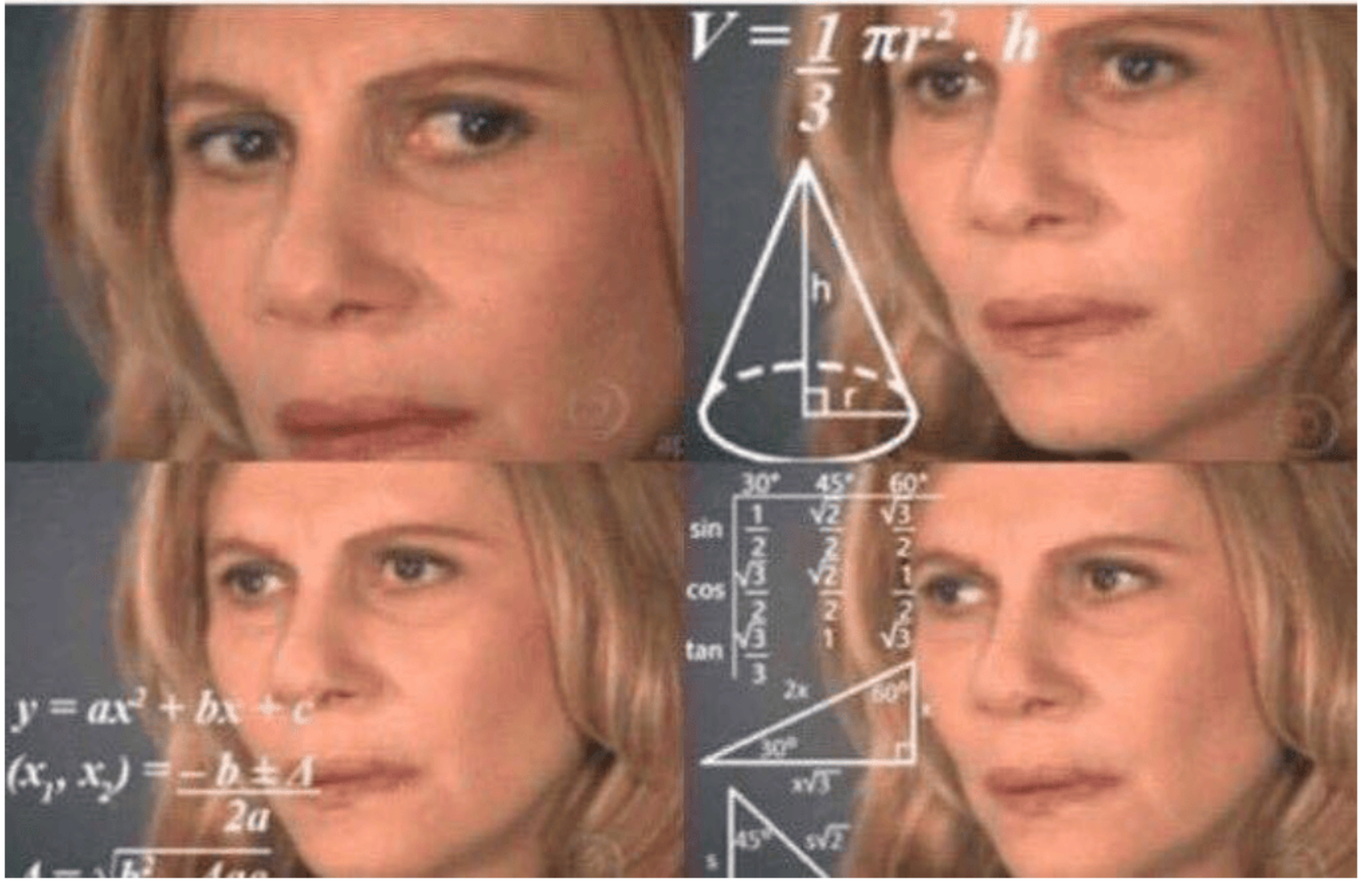
Many applications require the generation of **random numbers** for various purposes:

- Generation of cryptographic material
- Simulation and modelling of complex systems
- Sampling from large data sets

Cool, but…

what exactly is randomness?

what exactly is randomness?



what exactly is randomness?

For example, are these number random?

$$1338 \rightarrow 890 \rightarrow 1632$$
$$\rightarrow 1144 \rightarrow 918 \rightarrow 2068$$
$$\rightarrow 878 \rightarrow 1002 \rightarrow 1386$$
$$\rightarrow ??? \rightarrow ??? \rightarrow ???$$

Are we able to continue the sequence?

Are we able to correctly predict the next number?

Those numbers were generated starting from the names of Metro B subway stations in Rome, from "Laurentina" to "Termini"

From station names to numbers (1/4)

1. First, we go from the metro station name to a sequence of numbers using the underlying **ASCII encoding**.
2. Then, we combine those numbers using basic **mathematical operations**.

From station names to numbers (2/4)

Metro station names $\longrightarrow$ numbers, using the underlying **ASCII encoding**

$$\begin{aligned} \mathrm{T} &\longrightarrow 84 \quad , \quad \mathrm{e} \longrightarrow 101 \quad , \quad \mathrm{r} \longrightarrow 114 \\ \mathrm{m} &\longrightarrow 109 \quad , \quad \mathrm{i} \longrightarrow 105 \quad , \quad \mathrm{n} \longrightarrow 110 \\ \mathrm{i} &\longrightarrow 105 \end{aligned}$$

From station names to numbers (3/4)

Then, we combine those numbers with basic **mathematical operations**.

$$\begin{aligned} 109 \oplus 84 &= (1101101)_2 \oplus (1010100)_2 \\ &= (0111001)_2 \\ &= 57 \end{aligned}$$

From station names to numbers (4/4)

For example,

$$
\begin{aligned}
\mathrm{Hi} &\longrightarrow 72 \;\; 105 \\
&\longrightarrow ((((0 \oplus 72) + 72) \oplus 105) + 105) \\
&\longrightarrow (((72 + 72) \oplus 105) + 105) \\
&\longrightarrow ((144 \oplus 105) + 105) \\
&\longrightarrow (249 + 105) \\
&\longrightarrow 354
\end{aligned}
$$

This is the relevant code

```
#!/usr/bin/env python3
subway_B = ["laurentina", "EUR Fermi", "EUR Palasport", "EUR Magliana
            "Marconi", "Basilica S. Paolo", "Garbatella", "Piramide",
             "Circo Massimo", "Colosseo", "Cavour", "Termini" ]

def station_to_number(station_name):
    result = 0
    for c in station_name:
        result = (result ^ ord(c)) + ord(c) #!
    return result

if __name__ == "__main__":
    for metro_station in subway_B:
        print(station_to_number(metro_station))
```

(code/example_1_subway2seq.py)

```
[leo@ragnar code]$ python3 example_1_subway2seq.py
1338
890
1632
1144
p918
2068
878
1002
1386
1078    <---
824     <---
912     <---
```

We are thus able to complete the sequence

$$
\begin{aligned}
1338 &\to 890 \to 1632 \\
&\to 1144 \to 918 \to 2068 \\
&\to 878 \to 1002 \to 1386 \\
&\to 1078 \to 824 \to 912
\end{aligned}
$$

Weird but completely deterministic pattern

Definitely **not random**!

Q: What is randomness? (1/5)

A1:

> *"Something is random if and only if it happens by chance"*

Reaction: no sh!t, Sherlock.

What do you mean with "chance"?

Q: What is randomness? (2/5)

A2:

> *"scientists use chance, or randomness, to mean that when physical causes can result in any of several outcomes, we cannot predict what the outcome will be in any particular case." (Futuyma 2005: 225)*

Reaction: blah, blah, blah…

Q: What is randomness? (3/5)

Hard to define precisely.

Q: What is randomness? (4/5)

Practical definition:

Randomness is something that is "hard" to predict.

Q: What is randomness? (5/5)

As a consequence,

random numbers are hard to generate!

AND PSEUDO-RANDOMNESS?

So far we have:

1. **Random numbers** are hard to generate
2. Yet, we still need to generate **random numbers**

How to bridge this gap?

How can computers generate randomness?

MAIN IDEA: use an **approximation**!

Consider the following sequence of numbers

$$292616681 \to 1638893262 \to 255706927 \to \dots$$

Do you see any pattern?

$$292616681 \rightarrow 1638893262 \rightarrow 255706927 \rightarrow \ldots$$

While these numbers do look random, they are generated through a completely deterministic process using a **PRNG**

$$\text{PRNG} \longrightarrow \text{Pseudo Random Number Generator}$$

The previous numbers can be generated **deterministically** with the following **C** code

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
  srand(1337);
  int n = 10;

  for (int i = 0; i < n; i++) {
    printf("%d\n", rand());
  }

  return 0;
}
```

(code/example_2_rand_example.c)

$$292616681 \rightarrow 1638893262 \rightarrow 255706927 \rightarrow \ldots$$

```
@ragnar code]$ gcc example_2_rand_example.c -o example_2_rand_example
```

```
[leo@ragnar code]$ ./example_2_rand_example
292616681
1638893262
255706927
995816787
588263094
1540293802
343418821
903681492
898530248
1459533395
```

The sequence generated by a PRNG **is completely determined by internal state of the PRNG and the initial seed value, which initializes the internal state**

$$\text{seed} \longrightarrow \text{PRNG} \longrightarrow \text{output}_0, \text{output}_1, \ldots$$

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

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

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

Some important 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**?

Said in another way…

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

$$x_n \longrightarrow \ ?$$

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

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

We will see why using PRNGs in certain contexts could be dangerous.

Now, there are many PRNGs:

- **Middle-square method** (1946)
- **Linear Congruential Generators** (1958)
- **Linear-feedback shift register** (1965)
- …
- **Mersenne Twister** (1998)
- **xorshift** (2003)
- **xoroshiro128+** (2018)
- **squares RNG** (2020)

To understand how PRNGs work we will analyze two specific implementations:

- **rand()**, implemented in C. (today)

```c
#include <stdlib.h>
seed(1337);
printf("%d\n", rand()); // 292616681
```

- **getrandbits()**, implemented in python. (next lecture)

```python
import random
random = random.Random(1337)
print(random.getrandbits(32))  # 2653228291
```

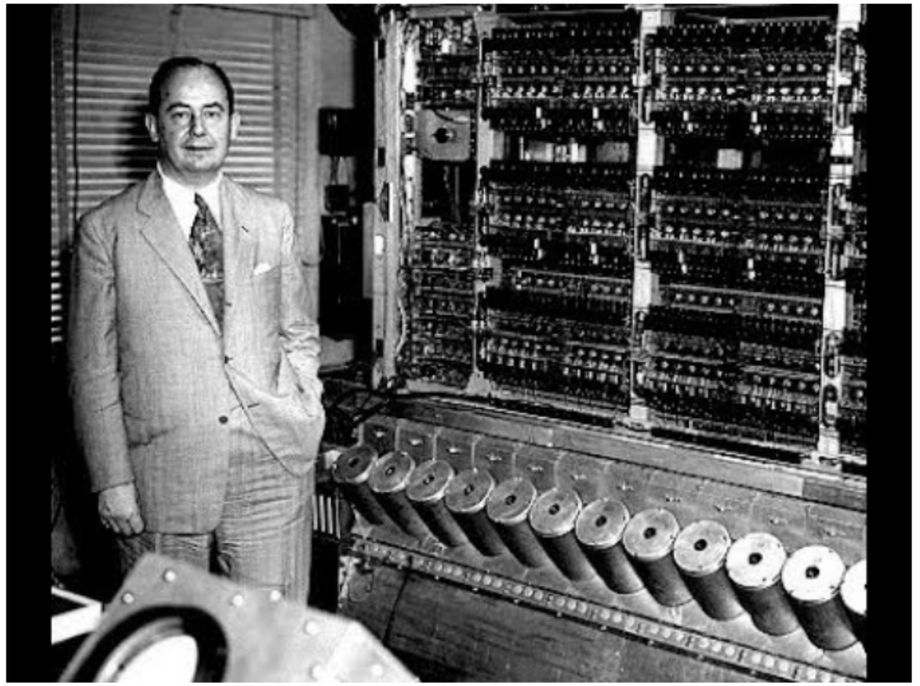
But first, let us consider a simple example.

A FIRST PRNG: MIDDLE SQUARE METHOD

One of the simplest PRNG.

Invented by **John Von Neumann** around 1949.

It is "weak", but it is a good starting point to approach the world of PRNGs.

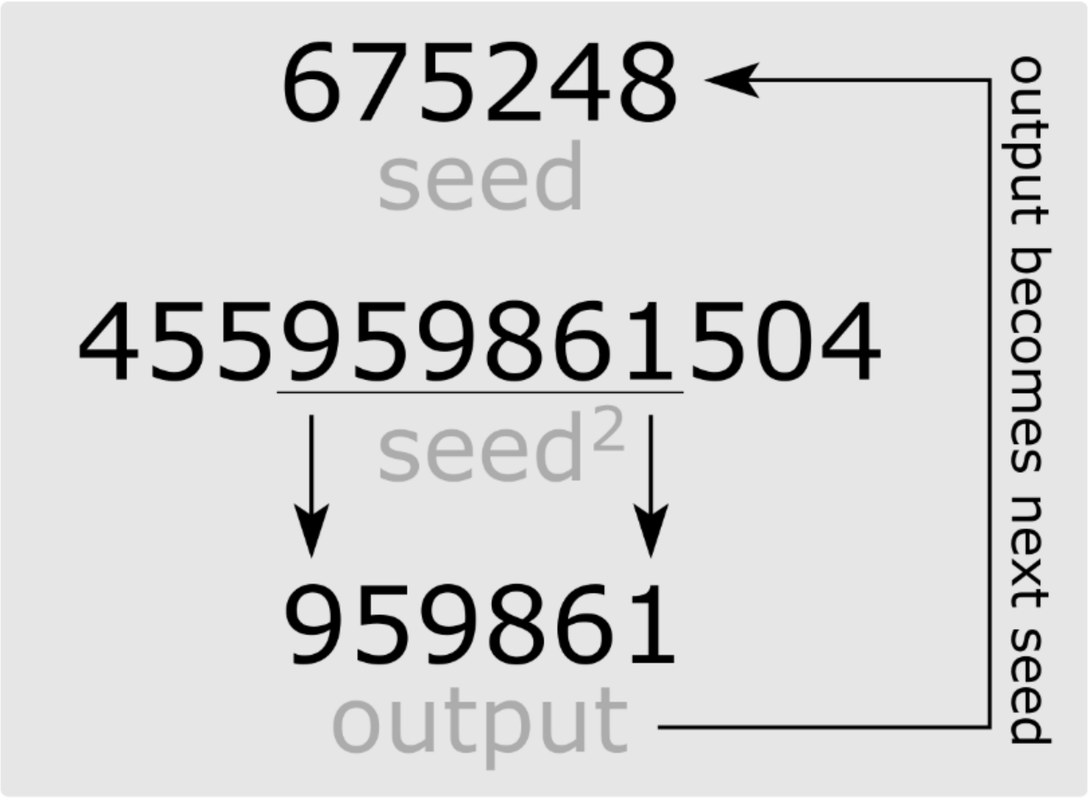


John Von Neumann

It works as follows:

- a n digit number is given in input as a **seed**
- to produce the next number:
  - square the seed
  - add leading zeros to reach a $2n$ digit number
  - return the n middle digits
  - the returned number becomes the new seed

For example,


675248
seed
455959861504
seed²
959861
output
output becomes next seed

Some sequences with different seeds,

$$
\begin{aligned}
675248 &\longrightarrow 959861, & 333139, & & 981593, \ldots \\
1337 &\longrightarrow 7875, & 156, & & 243, \ldots \\
42 &\longrightarrow 76, & 77, & & 92, \ldots
\end{aligned}
$$

Is it statistically useful?

Not really, as it usually has a short **period**.

Also, the value of n must be even in order for the method to work. (can you see why?)

state: total amount of memory that is used internally by the PRNG to generate the sequence of numbers.

Q: How big is the state for the Middle Square Method?

A: The memory necessary to store the n digit number, which is at most…

$$\log_2(10^n - 1)$$

Is it **cryptographically secure**?

no (trivially).

Exercise (optional): implement the Middle Square Method PRNG using a programming language you desire.

Prefered options are **Python** or **C**.

A SECOND PRNG: LINEAR CONGRUENTIAL GENERATOR

Consider the code of before

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
  srand(1337);
  int n = 10;

  for (int i = 0; i < n; i++) {
    printf("%d\n", rand());
  }

  return 0;
}
```

(code/example_2_rand_example.c)

If we execute it, we get

```
@ragnar code]$ gcc example_2_rand_example.c -o example_2_rand_example
```

```
[leo@ragnar code]$ ./example_2_rand_example
292616681
1638893262
255706927
995816787
588263094
1540293802
343418821
903681492
898530248
1459533395
```

How are these numbers generated?

$$292616681, \quad 1638893262, \quad 255706927$$
$$995816787, \quad 588263094, \quad 1540293802$$
$$343418821, \quad 903681492, \quad 898530248$$
$$1459533395, \quad \ldots$$

The **libc** implementation of **rand()** has two distinct behaviors, depending on the value of an internal variable

`buf->rand_type`

- If it is equal to 0, we have a simple

  Linear Congruential Generator

- Otherwise, we have a more complex

  Additive Feedback Generator

By default **rand()** has the more complex behavior of an **Additive Feedback Generator** type of **PRNG**

```
srand(1337)
rand()
```

The **LCG** behavior has to be manually activated

```c
int main(void) {
  // initialize LCG
  char state1[8];                  // !
  initstate(1337, state1, 0); // !
  setstate(state1);                // !

  // use the PRNG
  srand(1337);
  int n = 10;
  for (int i = 0; i < n; i++) {
    printf("%d\n", rand());
  }
  return 0;
}
```

(code/example_3_rand_lcg.c)

Q: How did you figure this out?

A: some research, using:

- search engines
- reading source code
- debugging with **gdb**

(for those interested, at the end of the lecture I will do an **interactive debugging session**).

Let us focus on the first, simpler case.

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.

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

Let's look at the LCG implemented in the **libc**…

LCG IN RAND()'S GLIBC

Initialization in __srandom_r()

```
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 __random_r()

```
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

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

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**)

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:

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.

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

$$\begin{aligned} x_{n+1} &= (1337 \cdot 1103515245 + 12345) \mod 2147483648 \\ &= 78628734 \end{aligned}$$

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 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

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

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

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

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

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

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

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

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

$$\Longrightarrow$$

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

Computing b (1/2)

Code to compute the increment b

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

(code/example_5_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

```
[leo@archlinux code]$ python3 attack_lcg.py
c=2147483648
a=1103515245
b=12345
```

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

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)

```c
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_attack_lcg.py** to extract the parameters

```
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?

Did we break the math somehow?

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

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

Is this the case when working with C?

Someone said… what, overflows?

In C every datatype has a fixed number of bytes.

$$\texttt{uint32\_t} \longrightarrow 4 \text{ bytes}$$
$$\longrightarrow \underbrace{01010101101011100011101010111011}_{32 \text{ bits}}$$

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

$$
\begin{aligned}
4294967295 &\longrightarrow \overbrace{11111111111111111111111111111111}^{32\text{ bits}} \\
4294967296 &\longrightarrow 00000000000000000000000000000000
\end{aligned}
$$

Overflows break our model

The correct model when dealing with overflows is the following one

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

When things break down, asses your models.

(works in all aspects of life, btw)

TOWARDS NEXT LECTURE

Today we have

- Introduced the concept of a PRNG
- Defined two specific types of PRNGs:
  - Middle Square Method
  - Linear Congruential Generator
- Showed how to "break" LCG types of PRNGs

In the next lecture we will

- Define a different PRNG called the **Mersenne Twister**
- Show how it also can be "broken"
- Show how to use cryptographically secure PRNGs

REFERENCES

For the creation of this lecture, the following **resources** were used

---

- Chance versus Randomness (Standford Encyclopedia)
- glibc rand function implementation (StackOverflow)
- How Brittle Are LCG-Cracking Techniques? (StackExchange)
- Cracking a linear congruential generator (StackExchange)
- George Marsaglia – RANDOM NUMBERS FALL MAINLY IN THE PLANES

EXTRA: DEBUGGING RAND()

Let us now describe how to understand the internal behavior of the following code.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
  srand(1337);
  int n = 10;

  for (int i = 0; i < n; i++) {
    printf("%d\n", rand());
  }

  return 0;
}
```

To analyze the implementation we need to understand to understand in particular:

1. Which function is called when initializing the PRNG?

```
srand(42);
```

2. Which function is called when generating the next pseudo-random number?

```
printf("%d\n", rand());
```

Given that the **libc** is an **open-source** project, the idea is to look at the source code.

To this end, we will need to:

1. Understand our **libc** version
2. Download the **source code** for our appropriate version
3. Analyze the source

Step 1/3: Understand our **libc** version

A simple way is to use **ldd** command, which prints the **shared objects** used by a binary.

```
[leo@archlinux code]$ ldd rand
linux-vdso.so.1 (0x00007ffc517fa000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007fbf8577f000)
...
```

```
[leo@ragnar code]$ ldd --version
ldd (GNU libc) 2.36
...
```

Step 2/3: Download the **source code**

Given that our version if 2.36, we can download the source code using **curl** by making an appropriate **HTTP request** to https://ftp.gnu.org

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

After we have downloaded the source, we can decompress it with **tar**

```
tar -xvf glibc-2.36.tar.bz2 && cd glibc-2.36
```

Step 3/3: Analyze the source

To analyze the code we can start by using **grep**

```
[leo@ragnar glibc-2.36]$ grep -nr 'srand(' .
./stdlib/random_r.c:61:   rand()/srand() like interface, ...
./stdlib/random.c:60:   rand()/srand() like interface, ...
```

We can also start do debug the code using **gdb**.

If we want to do this however it is useful to install a **debug build** of the **libc**, which has all the relevant **symbols** information.

Useful script (in archlinux): install-debug-glibc.sh

LIVE DEMO