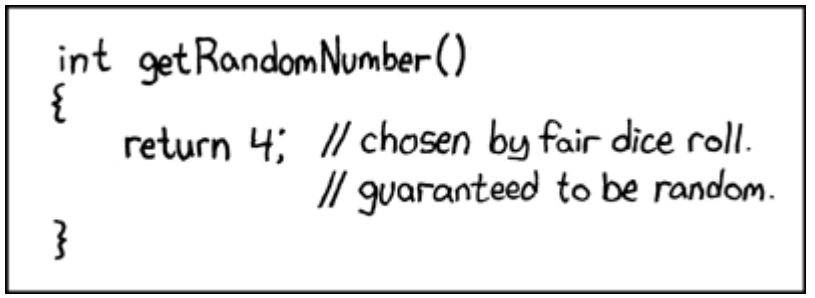
IMMUNITY

The Java soothsayer

A practical application for insecure randomness vulnerabilities



Alejo «Alex» Popovici
27 de Septiembre de 2017
alejo@immunityinc.com

:/# whoami

Alejo «Alex» Popovici

- 26 Years old
- Been working as a pentester for 3 years.
- Currently working at Immunity Inc.

A Pentester's shameful tactics

IMMUNITY

- Broken SSL/TLS…
- Old software/framework versions without real life applications…
- ClickJacking... (The cringe is strong with this one)
- Useless/pointless account enumeration…
- AND
- *Insecure randomness.*

We've all been there

IMMUNITY

- But, as a wise friend once said:

"Whatever lets you deface a site is a critical, the rest is low and everything in between is chamuyo"

The main point of this talk

IMMUNITY

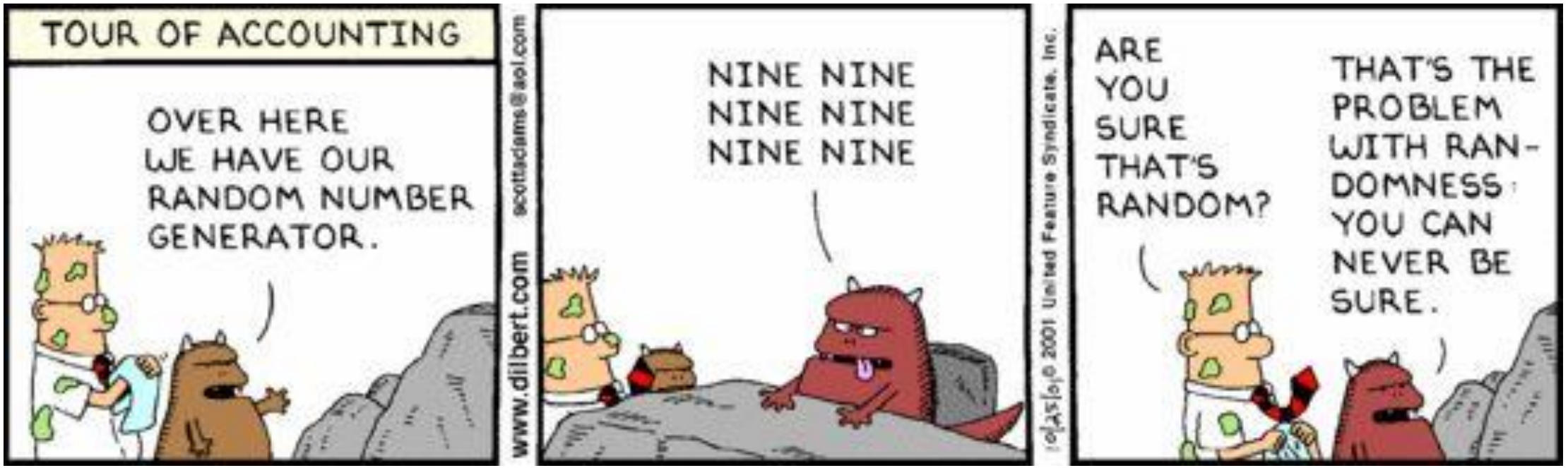
- Go the extra mile for your lows.
- They believe in you, believe in them.
- AND if you see something like this:

- You go like this…

The main point of this talk

What's a PRNG

- A Pseudo Random Number Generator, is an algorithm for generating a sequence of numbers whose properties approximate the properties of random numbers.



Apache Commons Lang RandomStringUtils

- Very useful to generate random strings.
- Can produce secure strings, if used properly.
- Uses java.util.Random() by default.

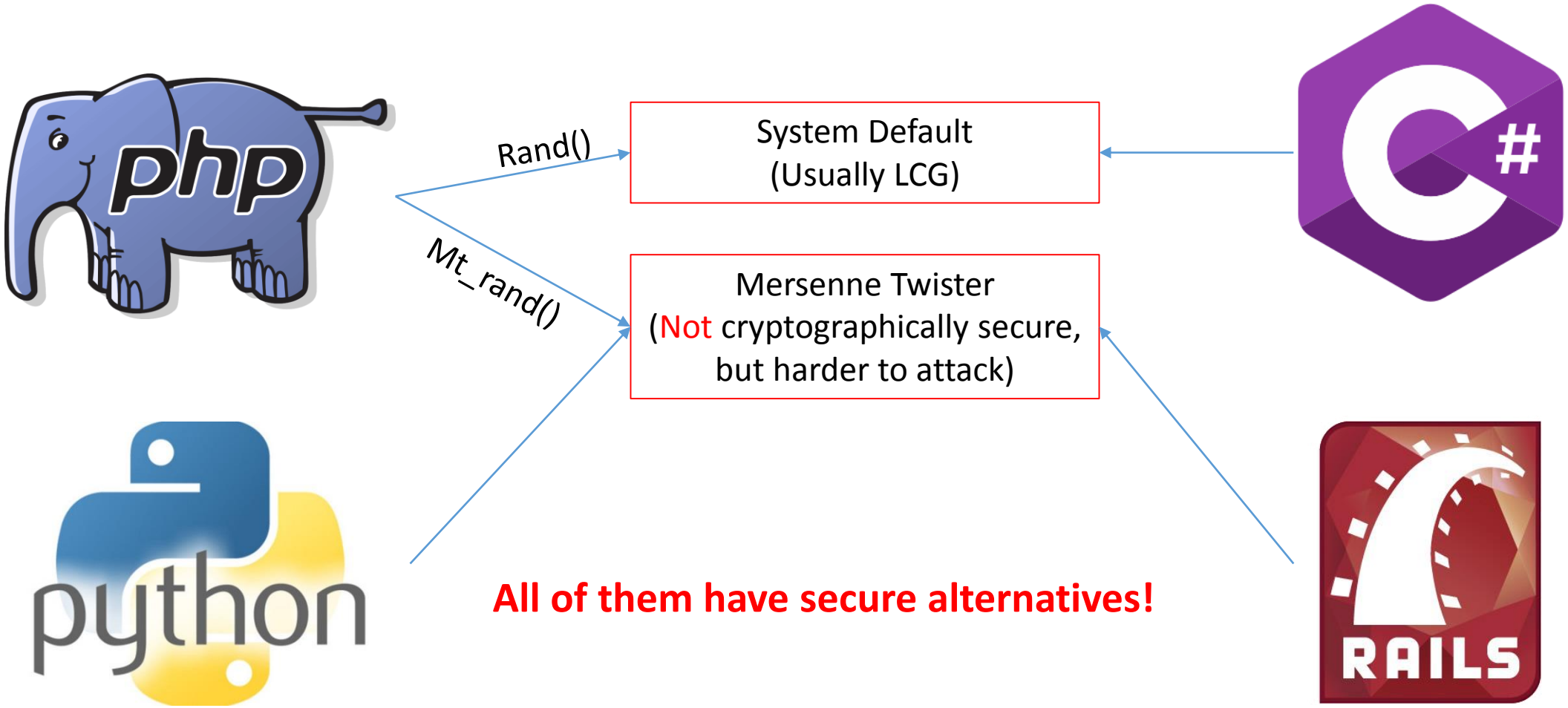
```
41  public class RandomStringUtils {
42
43      /**
44       * <p>Random object used by random method. This has to be not local
45       * to the random method so as to not return the same value in the
46       * same millisecond.</p>
47       */
48      private static final Random RANDOM = new Random();
49
50      /**
51       * <p>{@code RandomStringUtils} instances should NOT be constructed in
52       * standard programming. Instead, the class should be used as
53       * {@code RandomStringUtils.random(5);}.</p>
54       *
55       * <p>This constructor is public to permit tools that require a JavaBean instance
56       * to operate.</p>
57       */
58      public RandomStringUtils() {
59        super();
60      }
```

Java’s Random() class

IMMUNITY

- Uses a Linear Congruential Generator (LCG) for randomness.
- Implemented on RandomStringUtils as a static member, so it acts like a singleton across the server instance (Usually tomcat).
- Never reseeded unless explicitly done.

PRNGs used by other web servers



Lineal Congruential Generator

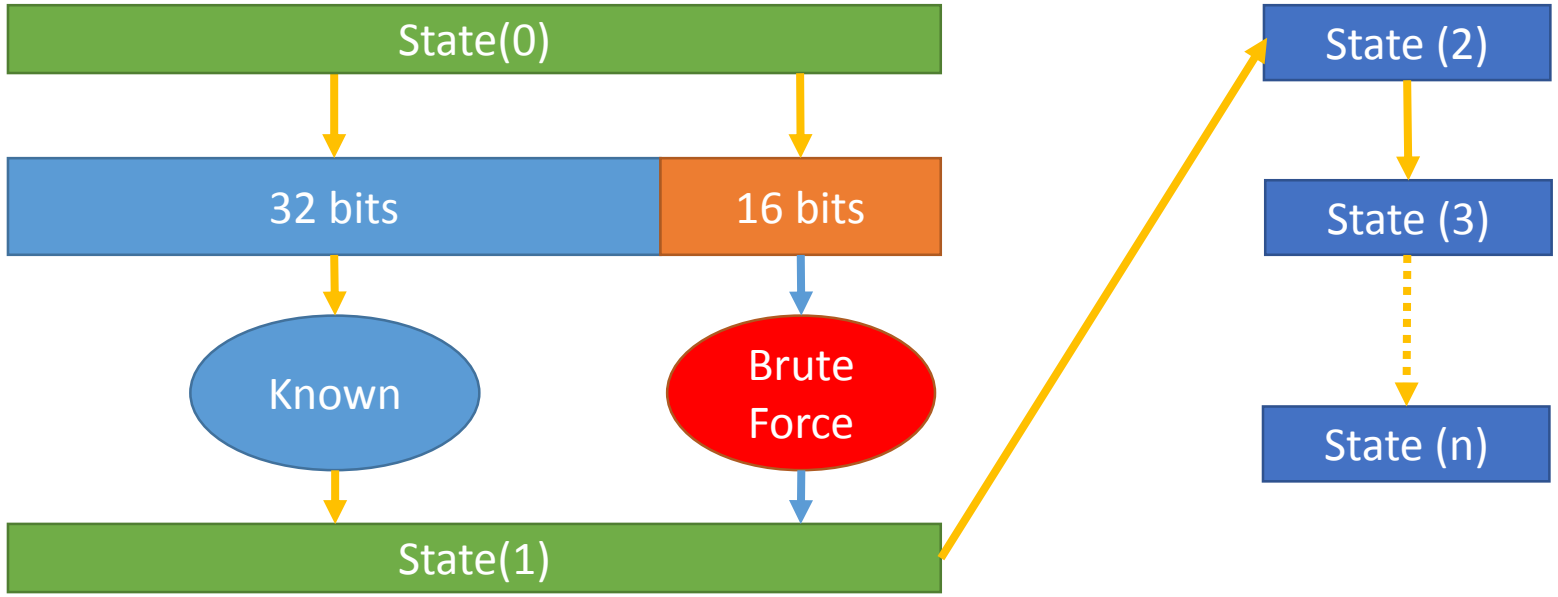
IMMUNITY

- Very (very!) performant, as it only consists of 4 operations (MULT, SUM, AND and ROL)
- Internal state 48 bits long.
- It is Not cryptographically secure, can predict future output with just one state.
- Implemented in Java as:

```
169:     * @param bits the number of random bits to generate, in the range 1..
170:     * @return the next pseudorandom value
171:     * @since 1.1
172:     */
173:    protected synchronized int next(int bits)
174:    {
175:      seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
176:      return (int) (seed >>> (48 - bits));
177:    }
```

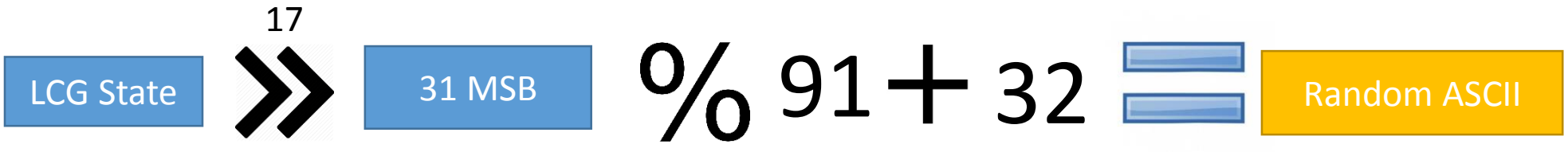
Cracking java.util.Random()

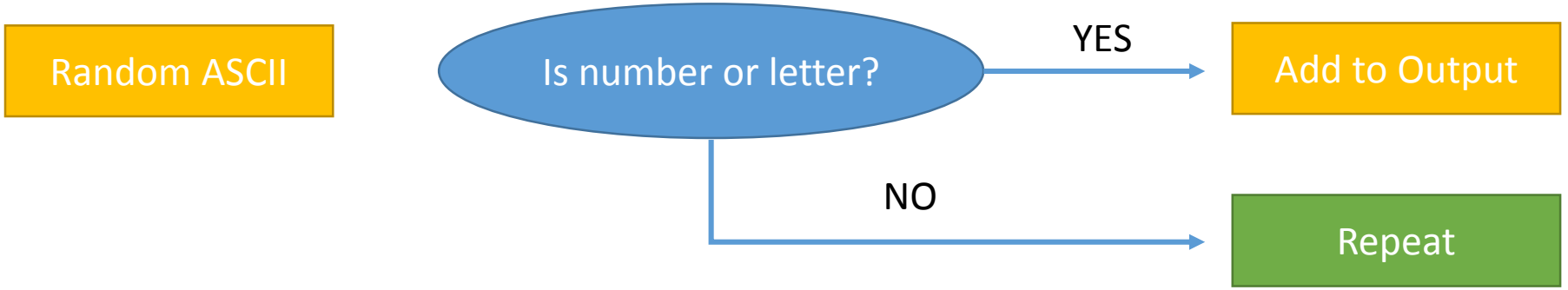
- Random.nextInt() outputs 32 bits.
- Need to crack the other 16 bits.
- Brute forcing on a regular desktop PC would take literally milliseconds.



RandomAlphaNumeric

1) Gets an Int from java.util.Random()
2) Grabs the most significant 31 bits (ROR)
3) Calculates (bits % 91)+32 to get a random ascii character.
4) Check if the ascii character is a number or a letter.





Problems for this attack

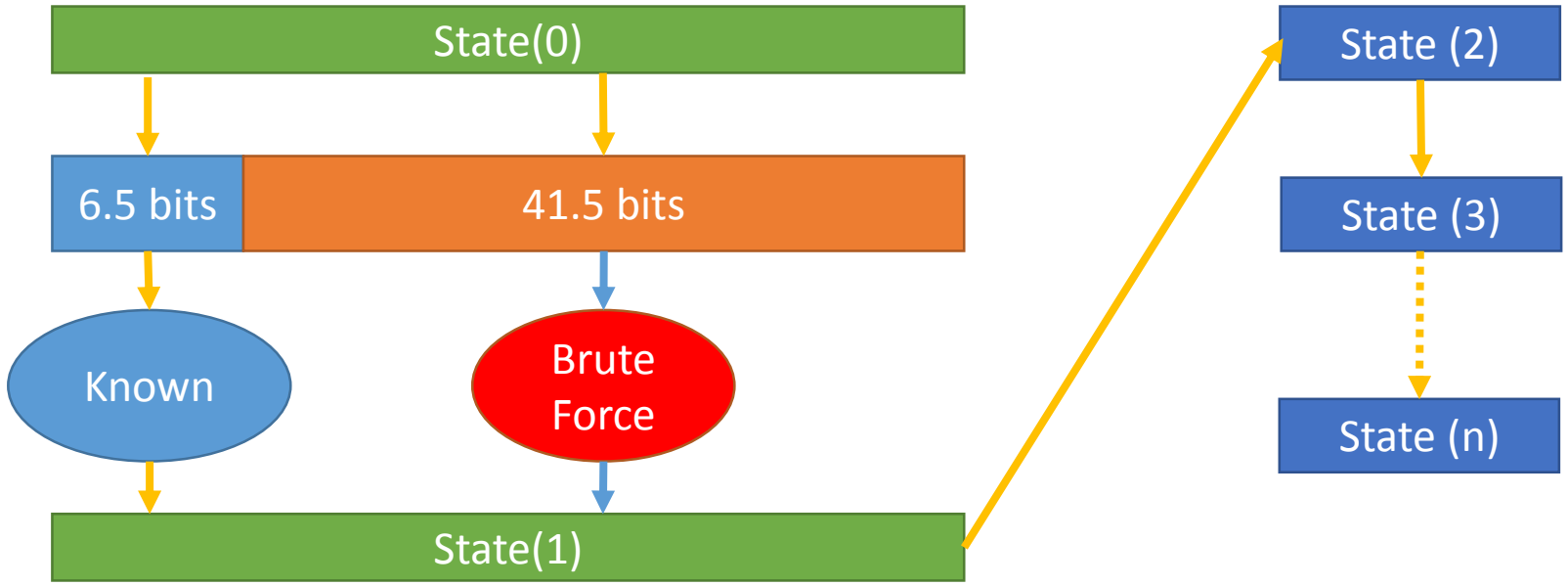
Problems for this attack

IMMUNITY

- A big chunk of the state is discarded.
- In theory, brute forcing the whole state of 48 bits would take in my i5 around 130 hours or 5 days.
- Which is still feasible, but a pain in the ass…

Problems for this attack

- It is possible to reduce the time required
- We extract state information from the first character!
- Reduces complexity by a factor of 2^6.5 (91).



Problems for this attack

- So, if we steal the first 6.5 bits of the state from the first letter of the output…

```
=== Benchmark Mode ===
Original token provided: h4zpBwnTclfXtBeiVE2T6TnaxGbT26
Size of token:30
Cracking state...
Number of threads: 4
State = 33318176254522
40.5 bits cracked in 27.162794 seconds.
To crack the state would take a maximum of 4916.990723 seconds.
Or 1.365831 hours.
```

Maximum cracking time: 1 hour, 22 minutes

Problems for this attack

- Multi-thread source code included (Compile with –lpthreads –O3)

How do we steal those 6.5 bits?

IMMUNITY

1. We take the first byte of the sequence and we substract 32, to get the original byte from the prng.

Sequence[0] – 32 = First byte

2. We apply ROL 17, to reverse the ROR from the algorithm. We get our starting point. This represents the **lowest possible guess**.

First byte << 17 = Starting point

How do we steal those 6.5 bits?

IMMUNITY

3. We calculate 91 (The modulus) ROL 17. This is going to be our multiplier.

```
91 << 17 = Multiplier
```

4. We then brute force the last 17 bits. This will be our addendum.

5. Each try will be then:

```
Guess[i,j] = Starting point + Multiplier * i + addendum * j
```

6. Paralelizing is as trivial as starting each thread with incremental values in “i”. One for each thread.

7. Each thread then increments “i” by the total number of threads.

Practical applications

- What is RandomStringUtils used for?
- Anti-CSRF tokens…

```
20          def manage(CeditTemplate ceditTemplateInstance) {
21                  if(session["csrf-token"] == null)
22                          session["csrf-token"] = RandomStringUtils.randomAlphanumeric(255)
23
24                  if(ceditTemplateInstance == null)
25                          render(view: "manage")
26                  else
27                          respond ceditTemplateInstance
28          }
29
```

Practical applications

- Password reset tokens (Whoops!)

```
/**
* Generate a reset key.
*
* @return the generated reset key
*/
public static  String generateResetKey() {
    return RandomStringUtils.randomNumeric(DEF_COUNT);
}
```

- Never use sample apps as a start point. (Jhipster…)

Practical applications

- “Remember me” cookies (Yay!)

```
/**
* Generate a persistent token, used in the authentication remember-me mechanism.
*
* @return the generated token data
*/
public static  String generateTokenData() {
    return RandomStringUtils.randomAlphanumeric(DEF_COUNT);
}
```

Demo + free 0day

Conclusions

IMMUNITY

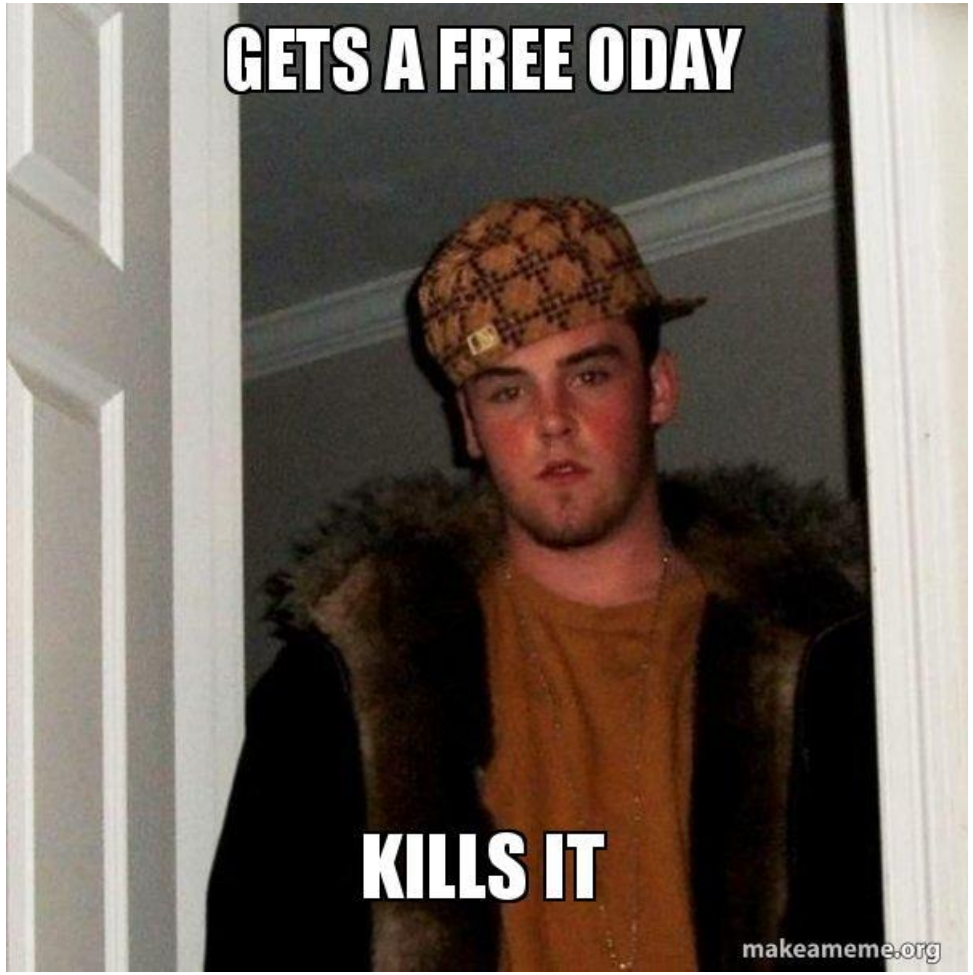
- Always look for funny looking or non-standard tokens on applications.
- Usual places to look are cookies and password reset emails.
- If you have access to the code, check the way the tokens are implemented.
- Never use the default Random() generator for security implementations.
- You can get the **code** in:

https://github.com/alex91ar/randomstringutils/

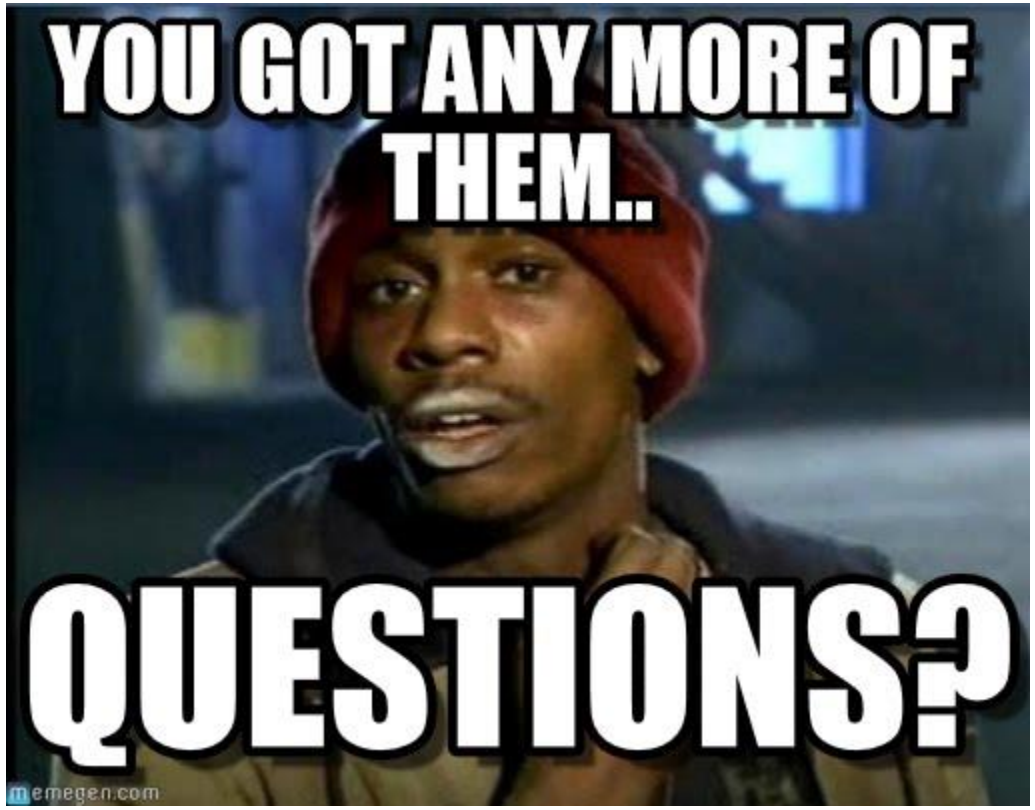
Be safe, always use… A VPN ;)

IMMUNITY

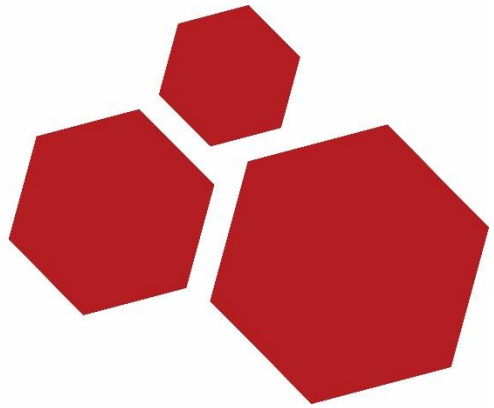
Don’t be THAT guy…



Questions?



We are hiring!

IMMUNITY 

Send your CV to:
CV@immunityinc.com

Special thanks to

- Esteban Guillardoy (Helped me with Java)
- Roderick Asselineau (Helped me with the math)
- Oren Isacson (Helped me to optimize the code)