RodrigoTeixeira - Roullete Simulator — Reverse Engineering Write-up

Challenge link: https://crackmes.one/crackme/693c48822d267f28f69b8518
Author: *RodrigoTeixeira*
Write-up by: *SenorGPT*
Tools used: *JADX*

Platform	Difficulty	Quality	Arch	Language
Windows	3.0	4.0	java	Java

Status: Complete
Goal: Document a clean path from initial recon → locating key-check logic → validation/reversal strategy

RodrigoTeixeira - Roullete Simulator — Reverse Engineering Write-up

1. Executive Summary

This crackme presents itself as a simple European roulette simulator, but its challenge lies entirely beneath the surface. The program initializes a custom *16-bit* pseudorandom number generator (*PRNG*) and allows the player to place bets while attempting to drive their balance into the negative which is the defined “victory” condition.

Under normal roulette odds and the program’s betting constraints, achieving a negative balance is **mathematically impossible** without exploiting implementation flaws. Through static and dynamic analysis, I determined that the balance variable is a *32-bit signed integer* meaning sufficiently large wins can trigger an **integer overflow** into the negative range.

By reverse-engineering the *PRNG* and brute-forcing its *16-bit* seed I was able to predict all future win/loss outcomes. Betting minimally on losses and going all-in on predicted wins causes exponential balance growth, eventually producing a winning overflow without violating the *crackme’s* rules. The challenge ultimately demonstrates the interplay between *RNG* predictability, integer limits, and controlled exploitation.

Running the program provides us with the following information:

- You start with 100 notes
- You are considered to have beaten the crackme if you get the note count to be negative.
- If you lose all your money - that is, if you have exactly 0 notes - the program automatically closes and you are not considered to having beaten the crackme.
- You may bet as much as you want, as long as it is a positive amount less than or equal to your current notes.
- Follows the European Roulette rules:
  - *18/37* chance of doubling the bet
  - *19/37* chance of losing the bet
- No patching, altering code, seed manipulation (IE, via System Time Adjustment), debuggers to extract the seed from memory, and using automated tools to automatically input to the *Portable Executable* (*PE*).
- Yes utilizing betting strategies such as Martingale or similar, decompiling and analysing the code, and whatever information the program provides during runtime.

---

2. Target Overview

2.1 UI / Behaviour

- **Inputs:**
  - A single integer bet per round
  - Must satisfy: `1 ≤ bet ≤ current_money`
  - Any non-integer input triggers the “Please Input a valid *32-bit* integer.” message
  - Entering `0`, negative values, decimals, floats, or extremely large integers causes the input loop to repeat until valid input is provided

- **Outputs:**
  - Displays current money after every round: `Money: <value>`
  - Displays all rules and restrictions if the user selects “See rules? [Y]es/[N]o”
  - On hitting exactly **0**, the program immediately exits (failure)
  - On hitting **negative money**, the program prints the success message: *“Congratulations, you solved the crackme!”*
  - Win/loss feedback is only inferred via changes in the money value (no explicit win/loss text)

2.2 Screens

Start-up

See Rules

```
See rules? [Y]es/[N]o: Y

You are not allowed to do the following:

1 -> Decompile, alter the code, and recompiling a modified version of the code.
2 -> Manipulating the system clock in any way ir order to obtain control over th
e initial RNG seed choice.
3 -> Using debug RAM watch or any other way to extract the seed value from memor
y.
4 -> Using automated tools to automaticly input to this program (exception: manu
ally copy-pasting text).

You are allowed to do the following:

1 -> Using betting strategies such as martingale or similar.
2 -> Decompiling and analysing the code in any way. The code is not obfuscated.
Using the information that the program gives you at runtime in any way of you li
kings.
```

Failure case - Exits Once Money Reaches *0*

```
Money: 100
Enter bet: 100
Money: 0

david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/RodrigoTeixeira - Roullete Sim
ulator/binary (main)
$
```

---

3. Tooling & Environment

- OS: *Windows 11*
- Other: *JDK*
- Static tools: *JADX*

---

4. Static Recon

Upon extracting the files from the `.rar` file from the download, I noticed that it was just a plain `.class` *Java* file. A `.class` file does not strictly require packaging, but using a `.jar` greatly simplifies execution and ensures the entry point is explicit.

This requirement will be met by utilizing the `JDK`. Opening a terminal in the same directory as the `main.class` file I run the following command to convert the `.class` file into a `.jar` file.

```
jar cfe RouletteSimulator.jar Main Main.class
```

- `c` = create
- `f` = write to file
- `e` = set entry point (`Main-Class`)

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/RodrigoTeixeira - Roullete Sim
ulator/binary (main)
$ jar cfe RouletteSimulator.jar Main Main.class
```

Success, this built the `.jar` file with no error. Just to be sure I run the newly created `.jar` file with the following command:

```
java -jar ./RouletteSimulator.jar
```

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/RodrigoTeixeira - Roullete Sim
Welcome to this crackme by Rodrigo Teixeira from Portugal.

This is a roullete simulator.

You start with 100 money units and you are considered to
have beaten the crackme if you get negative money.

If you lose all your money (that is, if you have exactly 0 money units),
the program automatically closes and you are not considered to having beaten the
 crackme.

You may bet as much as you want, as long as it is a positive ammount
less than or equal to your current money.

This follows the European Roullete rules, that is, 18/37 chance of doubling the
bet,
and 19/37 chance of losing the bet.

Good luck!


See rules? [Y]es/[N]o:
```

Amazing!

4.1 Source Code

My first approach is to start analysing the source code to see if I can spot any clear or obvious logical bugs I could exploit. *BUT*, before I get to that, I decide to poke the binary a bit with some Dynamic Analysis.

```
// Source code is decompiled from a .class file using FernFlower decompiler (from Intellij IDEA).
import java.util.InputMismatchException;
import java.util.Scanner;

public class Main {
   private int seed;
   byte DecompilingJavaCodeIsReallyEasyIsntIt;

   public Main() {
      System.out.println("Welcome to this crackme by Rodrigo Teixeira from Portugal.\n\nThis is a roullete simulator.\n\nYou start with 100 money units and you are considered to\nhave beaten the crackme if you get negative money.\n\nIf you lose all your money (that is, if you have exactly 0 money units),\nthe program automatically closes and you are not considered to having beaten the crackme.\n\nYou may bet as much as you want, as long as it is a positive ammount\nless than or equal to your current money.\n\nThis follows the European Roullete rules, that is, 18/37 chance of doubling the bet,\nand 19/37 chance of losing the bet.\n\nGood luck!\n\n");
      this.seed = (int)System.nanoTime() & '\uffff';
      Scanner var1 = new Scanner(System.in);

      try {
         System.out.print("See rules? [Y]es/[N]o: ");
         String var2 = var1.nextLine();
         if (!var2.isEmpty() && var2.toLowerCase().charAt(0) == 'y') {
```

```
            System.out.println("\nYou are not allowed to do the following:\n\n1 -
> Decompile, alter the code, and recompiling a modified version of the code.\n2 -
> Manipulating the system clock in any way ir order to obtain control over the
initial RNG seed choice.\n3 -> Using debug RAM watch or any other way to extract
the seed value from memory.\n4 -> Using automated tools to  input to this program
(exception: manually copy-pasting text).\n\nYou are allowed to do the
following:\n\n1 -> Using betting strategies such as martingale or similar.\n2 ->
Decompiling and analysing the code in any way. The code is not obfuscated.\nUsing
the information that the program gives you at runtime in any way of you
likings.");
         }

         int var3 = 100;
         System.out.println("\n\nMoney: " + var3);

         while(true) {
            if (var3 <= 0) {
               if (var3 < 0) {
                  System.out.println("Congratulations, you solved the
crackme!\n\nPress Enter to Exit.");
                  var1.nextLine();
               }
               break;
            }

            int var4 = -1;

            do {
               try {
                  System.out.print("Enter bet: ");
                  var4 = var1.nextInt();
               } catch (InputMismatchException var7) {
                  System.out.println("Please Input a valid 32-bit integer.");
               }
            } while(var4 < 1 || var4 > var3);

            if (this.rand() % 37 < 18) {
               var3 += var4;
            } else {
               var3 -= var4;
            }

            System.out.println("Money: " + var3);
         }
```

```
      } catch (Throwable var8) {
         try {
            var1.close();
         } catch (Throwable var6) {
            var8.addSuppressed(var6);
         }

         throw var8;
      }

      var1.close();
   }

   private int rand() {
      this.seed ^= this.seed << 7 & '\uffff';
      this.seed ^= this.seed >>> 9;
      this.seed ^= this.seed << 8 & '\uffff';
      return this.seed;
   }

   public static void main(String[] var0) {
      new Main();
   }
}
```

Upon program initialization, the *PE* defines a global variable `seed` and assigns it the result of the following operation upon entering `Main`.

```
this.seed = ((int) System.nanoTime()) & 65535;
```

So `seed` is effectively *16-bits* : 0-65535.

It appears that the main logic of the roulette game resides within the `while` loop.

```
while (i > 0) {
    int iNextInt = -1;
    while (true) {
        try {
            System.out.print("Enter bet: ");
            iNextInt = scanner.nextInt();
        } catch (InputMismatchException e) {
```

```
            System.out.println("Please Input a valid 32-bit integer.");
        }
        if (iNextInt >= 1 && iNextInt <= i) {
            break;
        }
    }
    if (rand() % 37 < 18) {
        i += iNextInt;
    } else {
        i -= iNextInt;
    }
    System.out.println("Money: " + i);
}
```

Which utilizes its own `rand` method to manipulate the `seed` global.

```
private int rand() {
    this.seed ^= (this.seed << 7) & 65535;
    this.seed ^= this.seed >>> 9;
    this.seed ^= (this.seed << 8) & 65535;
    return this.seed;
}
```

`rand() % 37 < 18` decides win/lose:

- **Win** if `(rand() % 37) ∈ [0..17]`
- **Lose** otherwise.

After some thought I deduce the following. The balance `i` can never go negative. You either wander around `> 0`, or eventually hit `i = 0` and exit. So there is no **pure betting strategy** (*Martingale*, etc.) that guarantees `i < 0`. The only way to go negative is **integer overflow**. Recall that the max signed 32-bit int = `2,147,483,647` (= `0x7FFFFFFF`). If `i` and `bet` get large enough they will overflow and enter the *win condition*.

4.2 Analysing the Rules

Given the rules:

- Start: **100 notes**
- On each bet of size `b`:
  - Win: `notes += b` with prob **18/37**
  - Lose: `notes -= b` with prob **19/37**
- You must always bet `0 < b ≤ notes`
- If `notes == 0`, the program **exits**
- You "win the crackme" only if `notes < 0`

Key observation, **You can't cross 0 if the code really enforces `b ≤ notes`**.

- If `notes > 0` and you lose, new notes = `notes - b ≥ 0`.
- If you win, notes stays > 0.

So with those rules, a perfect implementation *never* reaches a negative balance. Therefore, **it's mathematically impossible to guarantee a negative balance**. That's the big hint: the "solution" is almost certainly about **finding a bug in the *Java* code**, not about being clever with Roulette strategies.

---

5. Dynamic Analysis

Before looking at the source code, I attempt poking it with weird inputs:

- Trying a *positive whole number*.

```
Enter bet: 5
Money: 105
Enter bet:
```

- Trying *zero*.

```
Money: 100
Enter bet: 0
Enter bet:
```

- Trying a *negative number*.

```
Enter bet: -100
Enter bet:
```

- Trying an *extremely large number* - value greater than a *signed 32-bit* max value limit.

```
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
Enter bet: Please Input a valid 32-bit integer.
```

  Causes this prompt to be output to console on repeat for what seems indefinitely.

- Trying a *positive & negative decimal number*.
  Same effect as above with an *extremely large number*.

- Trying junk like *1e9* & *helloworld*.
  Same effect as above with an *extremely large number* and *positive & negative decimal numbers*.

- Trying *spamming Enter*.

```
Money: 100
Enter bet:
```

  Doesn't parse the input.

No immediate weird behaviour was observed. Following this observation, I continue on back to the Source Code in order to attempt to find any bugs there.

---

6. Getting Gud

So the plan:

1. **Reverse the Pseudorandom Number Generator (*PRNG*) and win/lose mapping.**
2. **Interact with the *crackme* with small bets**, recording the win/lose pattern.
3. **Offline brute-force all *65536* possible seeds** to see which seed produces that exact pattern.
4. Once current seed/state is determined, **predict all future win/lose outcomes**.
5. Use that prediction to:
    - bet **1** when a loss is coming
    - bet **everything (`i`)** when a win is coming so money grows ~exponentially.
6. Keep going until `i` gets huge and a predicted win will overflow it past `2,147,483,647` into negative.

I first start by creating two *Java* helper functions in *Python*. These will emulate the *Java* specific behaviour in our *Python* code.

```
def to_int32(x: int) -> int:
    """Emulate Java's 32-bit signed int."""
    x &= 0xFFFFFFFF
    if x & 0x80000000:
        return x - 0x100000000
    return x


def java_urshift(x: int, n: int) -> int:
    """
    Emulate Java's >>> (unsigned right shift) on a 32-bit int.
    """
    return (x & 0xFFFFFFFF) >> n
```

Then start by emulating the `rand` method and branching logic.

```
# PRNG implementation from rand() function
```

```
def java_rand_step(seed: int) -> int:
    """
    One step of the Java rand() function given in the crackme:

        seed ^= (seed << 7) & 65535;
        seed ^= seed >>> 9;
        seed ^= (seed << 8) & 65535;
        return seed;

    We emulate it on a 32-bit int, like Java.
    """
    seed = to_int32(seed)
    seed ^= (seed << 7) & 0xFFFF
    seed = to_int32(seed)
    seed ^= java_urshift(seed, 9)
    seed = to_int32(seed)
    seed ^= (seed << 8) & 0xFFFF
    seed = to_int32(seed)
    return seed


# Win implementation from inside while loop
def is_win(seed: int) -> Tuple[int, bool]:
    """
    Given current seed, perform one rand() step and
    return (new_seed, is_win).

    Win condition in the crackme: (rand() % 37) < 18
    """
    new_seed = java_rand_step(seed)
    win = (new_seed % 37) < 18
    return new_seed, win
```

Also need to make some *Python* helper functions.

```
def win_losses_to_string(win_losses: List[bool]) -> str:
    """
    Convert a list of bool `win_losses` to a string representative format with
`L` characters for
    losses and `W` characters for wins.
    """
    win_lose_string = ""
    for _ in win_losses:
```

```
        win_lose_string += "W" if _ else "L"
    return win_lose_string


def parse_win_lose_string(win_lose_string: str) -> List[bool]:
    """
    Convert a string like 'WLLW' or 'w l L W' to [True, False, False, True].

    Any character that is not `w` or `W` will be considered a loss
    """
    win_losses = []
    for char in win_lose_string:
        win_losses.append(True) if char.lower() == 'w' else
win_losses.append(False)

    return win_losses
```

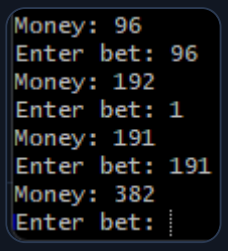
Finally, the code to brute force the patterns and match them against the input pattern provided by the user.

```
def find_matching_patterns(win_losses: List[bool]) -> List:
    found_matches = []
    rounds_played = len(win_losses)

    for seed in range(0, 65536+1): # inclusive, exclusive
        current_seed = seed
        found_match = True

        for i in range(rounds_played):
            current_seed, has_won = is_win(current_seed)
            if has_won != win_losses[i]:
                found_match = False
                break

        if found_match:
            found_matches.append([seed, current_seed])

    return found_matches
```

The `main` function in my *Python* code is just *Quality of Life* (*QOL*) code that gets the input and neatly outputs it to the user so I have not included it here.

---

7. Init to Winit



Ohh yeah! After confirming that our *PRNG* prediction *sidekick* is outputting good results, I continue *betting it all* when the sidekick predicts a **win** and only bet *1* when it predicts a **lose**.
Slowly but surely, with enough wins, the `money` variable should overflow from `2,147,.....` and bring us to a negative.

```
$ java -jar ./RouletteSimulator.jar
Welcome to this crackme by Rodrigo Teixeira from Portugal.

This is a roullete simulator.

You start with 100 money units and you are considered to
have beaten the crackme if you get negative money.

If you lose all your money (that is, if you have exactly 0 money units),
the program automatically closes and you are not considered to having beaten the
crackme.

You may bet as much as you want, as long as it is a positive amount
less than or equal to your current money.

This follows the European Roullete rules, that is, 18/37 chance of doubling the
bet,
and 19/37 chance of losing the bet.

Good luck!
```

```
See rules? [Y]es/[N]o: n


Money: 100
Enter bet: 1
Money: 99
Enter bet: 1
Money: 98
Enter bet: 1
Money: 99
Enter bet: 1
Money: 98
Enter bet: 1
Money: 99
Enter bet: 1
Money: 100
Enter bet: 1
Money: 99
Enter bet: 1
Money: 100
Enter bet: 1
Money: 101
Enter bet: 1
Money: 102
Enter bet: 1
Money: 101
Enter bet: 1
Money: 102
Enter bet: 1
Money: 101
Enter bet: 1
Money: 102
Enter bet: 1
Money: 103
Enter bet: 1
Money: 102
Enter bet: 1
Money: 101
Enter bet: 101
Money: 202
Enter bet: 1
Money: 201
Enter bet: 1
Money: 200
```

```
Enter bet: 1
Money: 199
Enter bet: 1
Money: 198
Enter bet: 1
Money: 197
Enter bet: 197
Money: 394
Enter bet: 1
Money: 393
Enter bet: 1
Money: 392
Enter bet: 1
Money: 391
Enter bet: 1
Money: 390
Enter bet: 1
Money: 389
Enter bet: 389
Money: 778
Enter bet: 1
Money: 777
Enter bet: 1
Money: 776
Enter bet: 776
Money: 1552
Enter bet: 1552
Money: 3104
Enter bet: 3104
Money: 6208
Enter bet: 1
Money: 6207
Enter bet: 1
Money: 6206
Enter bet: 1
Money: 6205
Enter bet: 1
Money: 6204
Enter bet: 1
Money: 6203
Enter bet: 1
Money: 6202
Enter bet: 6202
Money: 12404
```

```
Enter bet: 12404
Money: 24808
Enter bet: 1
Money: 24807
Enter bet: 24807
Money: 49614
Enter bet: 1
Money: 49613
Enter bet: 49613
Money: 99226
Enter bet: 1
Money: 99225
Enter bet: 99225
Money: 198450
Enter bet: 198450
Money: 396900
Enter bet: 1
Money: 396899
Enter bet: 1
Money: 396898
Enter bet: 1
Money: 396897
Enter bet: 1
Money: 396896
Enter bet: 396896
Money: 793792
Enter bet: 1
Money: 793791
Enter bet: 793791
Money: 1587582
Enter bet: 1
Money: 1587581
Enter bet: 1587581
Money: 3175162
Enter bet: 3175162
Money: 6350324
Enter bet: 6350324
Money: 12700648
Enter bet: 12700648
Money: 25401296
Enter bet: 25401296
Money: 50802592
Enter bet: 1
Money: 50802591
```

```
Enter bet: 1
Money: 50802590
Enter bet: 50802590
Money: 101605180
Enter bet: 101605180
Money: 203210360
Enter bet: 1
Money: 203210359
Enter bet: 203210359
Money: 406420718
Enter bet: 1
Money: 406420717
Enter bet: 406420717
Money: 812841434
Enter bet: 812841434
Money: 1625682868
Enter bet: 1625682868
Money: -1043601560
Congratulations, you solved the crackme!

Press Enter to Exit.

```

After *57* rounds, the trophy is won! And with that the crackme is now solved.

---

8. Conclusion

This *crackme* appears at first glance to challenge the player to beat unfavourable roulette odds, but deeper inspection shows that the intended solution is rooted in reverse engineering, not gambling strategy. Because the program enforces `1 ≤ bet ≤ money`, the balance can never reach a negative value through normal arithmetic. Wins increase the balance, losses decrease it, but neither can cross below zero.

The true weakness lies in two implementation details:

1. A **predictable *16-bit PRNG*** seeded from `System.nanoTime()`, which can be brute-forced using a short sequence of observed outcomes.
2. A ***32-bit signed integer*** **balance variable**, which overflows into the negative

range once sufficiently large.

By reconstructing the *PRNG*, matching the seed, and forecasting future win/loss outcomes, it becomes possible to bet the minimum on losing rounds and go all-in on winning rounds. This produces exponential growth in the balance until a win causes it to surpass `2,147,483,647`, overflow, and become negative. Satisfying the *crackme's* victory condition without modifying the code, manipulating the clock, or automating program input.

This challenge reinforces the importance of understanding integer boundaries, implementing secure *RNGs*, and recognizing that even simple logic can become vulnerable when underlying assumptions (like "integers don't overflow") are broken.