I'm not a math guy, so this is a poor man’s RNG analysis. Try it yourself at home!

9.1 Introduction

In PoC||GTFO 01:02, Dan Kaminsky proposed the following code for use as a Random Number Generator, arguing that the phase difference between a fast clock and a slow clock is sufficient to produce random bits in a high level language. This is a reduced version of his Dakarand program, with the intent of the reduction being that if there is any vulnerability within the code, that vuln ought to be exploitable.

```javascript
// These functions form an RNG.
function millis() { return Date.now(); }
function flip_coin() {
    n=0; then = millis()+1; while (millis()<=then) {n=!n; return n;}
}
function get_fair_bit() {
    while(1) {a=flip_coin(); if(a!=flip_coin()) {return(a);}}
}
function get_random_byte() {
    n=0; bits=8; while(bits--){n<<=1; n|=get_fair_bit();} return n;
}
// Use it like this.
report_console = function() {while(1){console.log(get_random_byte());}}
report_console();
```

Acturally the above code boils down to the function flip_coin, which takes a boolean value n=0 and continuously flips it until the next millisecond. The outcome of this repeated flipping shall be a random bit. We neglect the get_fair_bit function mostly in this analysis, as it just slows down the process and adds almost no additional entropy. For gathering random bits we are just left with the clock ticking for us.

9.2 A Naive Analysis

In order to analyze the output of the RNG we need some of its output, so I simply put up a small HTML piece which would pull out 100.000 random bytes out of the above RNG and log it to the HTML document. Then a severe 90-minute DoS on my Firefox 24 happened, after which I managed to copy and paste one hundred thousand uint8_t results into a text file.

After messing with several tools like ministat, sort and uniq I could show with the following ruby script that this RNG (on my machine) has a strong bias towards bytes with low hamming weights:

```
#!/usr/bin/env ruby

f=File.open(ARGV[0])

h = Hash.new
f.each_line do |m|
    n = m.to_i
    if h[n].nil?
        h[n]=1
    else
        h[n] = h[n]+1
    end
end

r = h.sort_by do [k,v] [v,k] end
```

After messing with several tools like ministat, sort and uniq I could show with the following ruby script that this RNG (on my machine) has a strong bias towards bytes with low hamming weights:
t.each do |a|
  puts "Num: \t#{a[0]} "+
  puts "\tCount: \t#{a[1]} "+
  puts "\tWeight: \t#{a[0].to_s(2).split('').reject{|j| j=='0'}.count}" end

The shortened output of this script on the 100k 8bit numbers is as follows. Note that the heavy
hamming weights, like 11111111 are least common and the light hamming weights, like 00000000 are
most common.

<table>
<thead>
<tr>
<th>Value</th>
<th>Count</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>254</td>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td>251</td>
<td>28</td>
<td>7</td>
</tr>
<tr>
<td>253</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>127</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>239</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>191</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>223</td>
<td>36</td>
<td>7</td>
</tr>
<tr>
<td>247</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>132</td>
<td>1173</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>1821</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1881</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1922</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1934</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2000</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2042</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2133</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>2145</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3918</td>
<td>0</td>
</tr>
</tbody>
</table>

The table lists the Number which is the output of the RNG along with this number’s hamming weight
as well as the count of this number in total within the 100,000 random bytes. For a random distribution
of all possible bytes we could expect roughly a count of 390 for each byte. But as we see, the number 0
with the hamming weight 0 peaks out with a count of 3918, whereas 255 with the hamming weight of 8
is generated 22 times by the RNG. That’s not fair!

9.3 My fair bit is not fair!

Real statistical analysis of an RNG is hard, and I will not attempt it here.
Still, looking at a few simple distributions might give us a hint (alas, only a
hint) of what might behind the unfairness.

First, a short recap on how this RNG works:

We’ve got a 1 millisecond timeslot from t0 to t1, where at t1 the flip_coin
method will stop. The first call to get_random_byte can happen anywhere
between t0 and t1:

```
Somewhere here the JS engine jumps in
```

Let’s say it is here:

```
```

Now the algorithm happily flips the bit until t1 and hands over the result
of this flipping as a random bit (note that we’re omitting get_fair_bit here).
Although we cannot predict the output of a single run of flip_coin, things get a bit more predictable when we make a lot of consecutive calls to flip_coin. Let’s say we need the time $d$ to process and store the result of flip_coin. So the next time we flip_coin we are at $t_1 + d_1$:

Now the RNG flips the coin until $t_2$ in order to give us a random bit. As we are calling the RNG more than twice in a row, the next flip_coin is at $t_2 + d_2$, and so on.

The randomness and fairness of the RNG’s random bit depends on how fairly and randomly we get odd and even values of $d$, since that the same amount of flips yields the same bit as we have a static start value of 0/false.\(^{11}\) So it makes sense to look at the distribution of $d$. To visualize this and to compare it with another browser I came up with this slight modification of the RNG that counts the flips and records them right inside the HTML page:

```javascript
function flip_coin ()
{i=0;n=0; then=millis ()+1; while (millis ()<=-then) {n=!n; i++;} return [n, i];}

function get_fair_bit ()
{while(1) {a=flip_coin (); if(a[0]!=flip_coin ()[0]) {return(a);}}}

function doit (){
 var i = 10000;
 while(i--){
 var d = document.getElementById('target');
 var content = document.createTextNode( get_fair_bit ().toString () + '
');
 d.appendChild(content);
 }
}
```

Loading the page in Chromium and Firefox and throwing them into gnuplot, we get:

![Graphs showing distribution of coin flips](image)

We can see that the graph for Chromium has a lot more variance in the number of coin flip within a millisecond than that for Firefox. Although, strictly speaking, it might still be possible to get good randomness with poor variance if the few frequent values were to alternate just so due to some underlying scheduling magic, it seems reasonable to expect that the same magic would also increase the variance in the flip numbers.

We can also see, with the help of simple UNIX tools, that Chromium counts do not peak out to a certain value, unlike those of Firefox:

\(^{11}\)The second coin flip in get_fair_bit complicates it a bit, but it cannot substantially improve the RNG’s entropy if it lacks in the first place.
9.4 Closing words
In conclusion we see that in Firefox under stress Dan’s RNG appears to fail at exactly the point he wanted to use as the main source of randomness. The tiny clock differentials used to gather the entropy are not given often enough in Firefox. There is still much room to stress this RNG implementation. Bonus rounds would include figuring exactly what the significant difference between the Firefox and Chromium JavaScript runtime is that causes this malfunction on Firefox. Also attacks on other JavaScript runtimes would be interesting to see. It might even be the case that this implementation has different results under different conditions with respect to CPU load.

A broader question occurs: The Dakarand RNG relies on what could be called a “code clock.” It may be that in many kinds of environments stressed code clocks tend to go into phase with one another. Driven by stress to seek comfort in each other’s rhythms, their chance encounters may grow into something more close and intimate, grinding into periodic patterns. Which, of course, is bad for randomness. Can we learn to tell such environments from others, where periodization with stress doesn’t happen? –PML