2 Four Lines of Javascript that Can't Possibly Work So why do they?

by Dan Kaminsky

```
// 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();
```

2.1 Introduction

When Apple's iPhone 5S was announced, a litany of criticism against its fingerprint reader was unleashed. Clearly, it would be vulnerable to decade old gelatin cloning attacks. Or clearly, it would utilize subdermal analysis or electrical measurement or liveness checking and not be vulnerable at all. Both fates were possible.

It took Nick DePetrillo and Rob Graham to say, "PoC || GTFO."

What Starbug eventually demonstrated was that the old attacks do indeed still work. It didn't have to be that way, but at the heart of science is experimentation and testing. The very definition of unscientific work is not merely that it will not be subjected to test but that by design it cannot.

Of course, I am not submitting an article about the iPhone 5S. I'm here to write about a challenge that's been quietly going on for the last two years, one that remains unbroken.

Can we use the clock differentials, baked into pretty much every piece of computing equipment, as a source for a True Random Number Generator? We should find out.

2.2 Context

"The generation of random numbers is too important to be left to chance," as Robert R. Coveyou from Oak Ridge liked to say. Computers, at least as people like to mentally model them, are deterministic devices. The same input will always lead to the same output.

Electrically, this is unnecessary. It takes a lot of work to make an integrated circuit completely reliable. Semiconductors are more than happy to behave unpredictably. Semiconductor manufacturers, by contrast, have behaved very predictably, refusing to implement what would admittedly be a rather difficult part to test.

Only recently have we gotten an instruction out of Intel to retrieve random numbers, RDRAND. I can't comment as to the validity of the function except to say that any audit process that refuses its auditors physical access to the part in question and disables all possible debugging or post-verification after release is not a process that inspires confidence.

But do we need the instruction? The core assumption is that in lieu of RDRAND the computer is deterministic, that the same input will lead to the same output. Seems reasonable, until you ask:

If all I do is turn a computer on, will it take the same number of nanoseconds to reach the boot screen?

If you think the answer is yes, PoC || GTFO.

If you think the answer is no, that there will be some amount of nanosecond drift, then where does this drift come from? The answer is that the biggest lie about your computer is that it's just one computer. CPU cores talk to memory busses talk to expansion busses talk to storage and networking and the interrupt of the month club. There are generally some number of clocks, they have different speeds and different tolerances, and you do not get them synchronized for free. (System-on-Chip devices are a glaring exception, but it's still rather common for them to be speaking to peripherals.)

Merely turning the machine on does not synchronize everything, so there is drift. Where there is drift, there is entropy. Where there is entropy, there is security.

2.3 This is Actually a Problem

To stop a brute force attack against your random number generator, you need a few bits. At least 80, ideally 128. Not 128 million. 128. Ever. For the life of that particular device. (Not model! The attacker can just go out and buy one of those devices, and find those 128 bits.) Now you may say, "We need more than 128 bits for production." And that's fine. For that, we have what are known as Cryptographically Secure Pseudo Random Number Generators (CSPRNG's). Seed 128 bits in, get an infinite keystream out. As long as the same seed is never repeated, all is well.

Cryptographers love arguing about good CSPRNGs, but the reality is that it's not that hard to construct one. Run a good cipher or hash function (not RC4) in pretty much any sort of loop and the best attack reduces to breaking that cipher or hash function. (If you disagree, PoC ‖ GTFO.) That's not to say there aren't "nice to have" properties that an ideal CSPRNG can acquire, but empirically two things have actually happened in the real world some of us are trying to defend.

First, most PRNG's aren't cryptographically secure. Most random numbers are not securely generated. They could be. CSPRNGs can certainly be fast enough. If we really wanted, they could be simple enough too. To be fair, the advice of "Just use /dev/urandom." is what most languages should follow. But there's a second issue, and it's severe.

The second issue, the hard part, is not expanding 128 bits to an infinite stream. The hard part is actually getting those 128 bits! So called "True Random Number Generation" is actually the thing we are bad at, in the real world. The CSPRNG of the gods falls to a broken TRNG. What is a kernel supposed to do when /dev/urandom wants data and there is no seed? The whole idea behind /dev/urandom is that it will provide answers immediately. And so, in general, it does.

And then Nadia Heninger scans the Internet, and finds that 1/200 RSA keys are badly formed. That's a floor, by the way. Keys that are similar but not quite identical are not counted in that 1/200. But of course, buying a handful of devices gives you the similarity map.

However bad clock differentials might be, they would not have created this apocalyptic failure rate.

2.4 This Didn't Have to Happen

In 1999, Daniel J. Bernstein pointed out that the 16 bit transaction ID in DNS was insufficient and that the UDP source port could be overloaded to provide almost 32 bits of entropy per DNS request. His advice was not accepted.

In 1996, Matt Blaze created Truerand, a scheme that pitted the CPU against signal handlers. His approach actually has a long and storied history, back to the VMS days, but it was never accepted either.

In 2011, I released Dakarand. Dakarand is a collection of approaches for pitting various clocks inside against a computer against each other. Many random number generation schemes come down to measuring something that varies by millisecond with something that varies by nanosecond. (Your CPU, running in a

tight loop, is a fast clock operating in the gigahertz. Your RTC—Real Time Clock—is much slower and is not reporting milliseconds accurate to the nanosecond. In confusion, profit.)

Dakarand may in fact fail, somehow, somewhere, in some mode. But thus far, it seems to work pretty much everywhere, even virtual machines. (As a TRNG, each read event can generate new seed material without depending on data that might have been inherited before VM cloning.)

In 2013, in honor of Barnaby Jack, I tossed together the code at the top of this article. It's the weakest possible formulation of this concept, written in JavaScript and hardened only with the barest level of Von Neumann. It is called oi.js, and you should break it.

After all, it's just JavaScript. It can't be secure.

The idea is, in fact, to find the weakest formulation of this concept that still works. PoC ‖ GTFO shows us where known security stops and safety margin begins.

2.5 On Measuring the Strength of Cryptosystems

Sometimes people forget that we regularly build remarkably safe code out of seemingly trivial to break components. Hash functions are generally composed of simple operations that, with only a few rounds of those functions, start becoming seriously tricky to reverse. RSA, through this lens, is just multiply as an encryption function, albeit with a mind bending number of rounds.

Humans do not require complex radioactivity measurements or dwellings on the nature of the universe to get a random bit. They can merely flip a coin, a system that is well described as the Newtonian interaction between a slow clock (coin goes up, coin goes down) and a fast clock (coin spins round and round.) Pretending that there is nothing with the properties of a simple coin anywhere in the mess that is a device that can at least run Linux is enabling vulnerability.

PoC's in defense are rare—now let's see what you've got ;)