10 Backdoors up my Sleeve

by JP Aumasson

SHA-1 was designed by the NSA and uses the constants `5a827999`, `6ed9eba1`, `8f1bbcdc`, and `ca62c1d6`. In case you haven't already noticed, these are hex representations of 2^{30} times the square roots of 2, 3, 5, and 10.

NIST's P-256 elliptic curve was also designed by the NSA and uses coefficients derived from a hash of the seed `c49d3608 86e70493 6a6678e1 139d26b7 819f7e90`. Don't look for decimals of square roots here; we have no idea where this value comes from.

Which algorithm would you trust the most? Right, SHA-1. We don't know why 2, 3, 5, 10 rather than 2, 3, 5, 7, or why the square root rather than the logarithm, but this looks more convincing than some unexplained random-looking number.

Plausible constants such as $\sqrt{2}$ are often called "nothing-up-my-sleeve" (NUMS) constants, meaning that there is a kinda-convincing explanation of their origin. But it isn't impossible to backdoor an algorithm with only NUMS constants, it's just more difficult.

There are basically two ways to create a NUMS-looking backdoored algorithm. One must either (1) bruteforce NUMS constants until one matches the backdoor conditions or (2) bruteforce backdoor constants until one looks NUMS.

The first approach sounds easier, because bruteforcing backdoor constants is unlikely to yield a NUMS constant, and besides, how do you check that some constant is a NUMS? Precompute a huge table and look it up? In that case, you're better off bruteforcing NUMS constants directly (and you may not need to store them). But in either case, you'll need *a lot of NUMS constants*.

I've been thinking about this a lot after my research on malicious hash functions. So I set out to write a simple program that would generate a huge corpus of NUMS-ish constants, to demonstrate to non-cryptographers that "nothing-up-my-sleeve" doesn't give much of a guarantee of security, as pointed out by Thomas Pornin on Stack Exchange.

The `numsgen.py` program generates nearly two million constants, while I'm writing this.[52] Nothing new nor clever here; it's just about exploiting degrees of freedom in the process of going from a plausible seed to actual constants. In that PoC program, I went for the following method:

1. Pick a plausible seed
2. Encode it to a byte string
3. Hash it using some hash function
4. Decode the hash result to the actual constants

Each step gives you some degrees of freedom, and the game is to find somewhat plausible choices.

As I discovered after releasing this, DJB and others did a similar exercise in the context of manipulated elliptic curves in their "BADA55 curves" paper,[53] though I don't think they released their code. Anyway, they make the same point: "The BADA55-VPR curves illustrate the fact that 'verifiably pseudorandom' curves with 'systematic' seeds generated from 'nothing-up-my-sleeve numbers' also do not stop the attacker from generating a curve with a one-in-a-million weakness." The two works obviously overlap, but we use slightly different tricks.

10.1 Seeds

We want to start from some special number, or, more precisely, one that will *look* special. We cited SHA-1's use of $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{10}$, but we could have cited

- π used in ARIA, BLAKE, Blowfish,
- MD5 using "the integer part of $4294967296 \times abs(sin(i))$",
- SHA-1 using `0123456789abcdeffedcba98-76543210f0e1d2c3`,
- SHA-2 using square roots and cube roots of the first primes,
- NewDES using the US Declaration of Independence,
- Brainpool curves using SHA-1 hashes of π and e.

[52] `https://github.com/veorq/numsgen`
`unzip pocorgtfo08.zip numsgen.py`

[53] `http://safecurves.cr.yp.to/bada55.html`

Special numbers may thus be universal math constants such as π or e, or some random-looking sequence derived from a special number: small integers such as 2, 3, 5, or some number related to the design (like the closest prime number to the security level), or the designer's birthday, or his daughter's birthday, etc.

For most numbers, functions like the square root or trigonometric functions yield an *irrational* number, namely one that can't be expressed as a fraction, and with an infinite random-looking decimal expansion. This means that we have an infinite number of digits to choose from!

Let's now enumerate some NUMS numbers. Obviously, what looks plausible to the average user may not be so for the experienced cryptographer, so the notion of "plausibility" is subjective. Below we'll restrict ourselves to constants similar to those used in previous designs, but many more could be imagined (like physical universal constants, text rather than numbers, etc.). In fact, we'll even restrict ourselves to *irrational* numbers: π, e, $\varphi = (1 + \sqrt{5})/2$ (the golden ratio), Euler–Mascheroni's γ, Apéry's $\zeta(3)$ constant, and irrationals produced from integers by the following functions

- Natural logarithm, $\ln(x)$, irrational for any rational $x > 1$;
- Decimal logarithm, $\log(x)$, irrational unless $x = 10^n$ for some integer n;
- Square root, $\sqrt{x}$, irrational unless x is a perfect square;
- Cubic root, $\sqrt[3]{x}$, irrational unless x is a perfect cube;
- Trigonometric functions: sine, cosine, and tangent, irrational for all non-zero integers.

We'll feed these functions with the first six primes: 2, 3, 5, 7, 11, 13. This guarantees that all these functions will return irrationals.

Now that we have a bunch of irrationals, which of their digits do we record? Since there's an infinite number of them, we have to choose. Again, this *precision* must be some plausible number. That's why this PoC takes the first N *significant digits*—rather than just the fractional part—for the following values of N: 42, 50, 100, 200, 500, 1000, 32, 64, 128, 256, 512, and 1024.

We thus have six primes combined with seven functions mapping them to irrationals, plus six irrationals, for a total of 48 numbers. Multiplying by twelve different precisions, that's 576 irrationals. For each of those, we also take the multiplicative inverse. For the one of the two that's greater than one, we also take the fractional part (thus stripping the leading digit from the significant digits). We thus have in total $3 \times 576 = 1728$ seeds.

Note that seeds needn't be numerical values. They can be anything that can be hashed, which means pretty much anything: text, images, etc. However, it may be more difficult to explain why your seed is a Word document or a PCAP than if it's just raw numbers or text.

10.2 Encodings

Cryptographers aren't known for being good programmers, so we can plausibly deny an awkward encoding of the seeds. The PoC tries the obvious raw bytes encoding, but also ASCII of the decimal, hex (lower and upper case), or even binary digits (with and without the `0b` prefix). It also tries Base64 of raw bytes, or of the decimal integer.

To get more degrees of freedom you could use more exotic encodings, add termination characters, timestamps, and so on, but the simpler the better.

10.3 Hashes

The purpose of hashing to generate constants is at least threefold.

1. Ensure that the constant looks *uniformly* random, that it has no symmetries or structure. This is, for example, important for the hash functions' initial values. Hash functions can thus "sanitize" similar NUMS by produce completely different constants:

```
>>> hex(int(math.tanh(5)*10**16))
'0x23861f0946f3a0'
>>> sha1(_).hexdigest()
'b96cf4dcd99ae8aec4e6d0443c46fe0651a44440'
>>> hex(int(math.tanh(7)*10**16))
'0x2386ee907ec8d6'
>>> sha1(_).hexdigest()
'7c25092e3fed592eb55cf26b5efc7d7994786d69'
```

2. Reduce the length of the number to the size of the constant. If your seed is the first 1000 digits of π, how do you generate a 128-bit value that depends on all the digits?

3. Give the impression of "cryptographic strength". Some people associate the use of cryptography with security and confidence, and may believe that constants generated with SHA-3 are safer than constants generated with SHA-1.

Obviously, we want a cryptographic hash rather than some fast-and-weak hash like CRC. A natural choice is to start with MD5, SHA-1, and the four SHA-2 versions. You may also want to use SHA-3 or BLAKE2, which will give you even more degrees of freedom in choosing their version and parameters.

Rather than just a hash, you can use a *keyed hash*. In my PoC program, I used HMAC–MD5 and HMAC–SHA1, both with 3×3 combinations of the key length and value.

Another option, with even more degrees of freedom, is a *key derivation*—or password hashing—function. My PoC applies PBKDF2–HMAC–SHA1, the most common instance of PBKDF2, with: either 32, 64, 128, 512, 1024, 10, 100, or 1000 iterations; a salt of 8, 16, or 32 bytes, either all-zero or all-ones. That's 48 versions.

The PoC thus tries $6 + 18 + 48 = 72$ different hash functions.

10.4 Decoding

Decoding of the hashes to actual constants depends on what constants you want. In this PoC I just want four 32-bit constants, so I only take the first 128 bits from the hash and parse them either as big- or little-endian.

10.5 Conclusion

That's all pretty simple, and you could argue that some choices aren't that plausible (e.g., binary encoding). But that kind of thing would be enough to fool many, and most would probably give you the benefit of the doubt. After all, only some pesky cryptographers object to NIST's unexplained curves.

So with 1728 seeds, 8 encodings, 72 hash function instances, and 2 decodings, we have a total of $1728 \times 8 \times 72 \times 2 = 1,990,656$ candidate constants. If your constants are more sophisticated objects than just 32-bit words, you'll likely have many more degrees of freedom to generate many more constants.

This demonstrates that *any invariant* in a crypto design—constant numbers and coefficients, but also operations and their combinations—can be manipulated. This is typically exploited if there exists a one in a billion (or any reasonably low-probability) weakness that's only known to the designer. Various degrees of exclusive exploitability ("NOBUS") may be achieved, depending on what's the secret: just the attack technique, or some secret value like in the malicious SHA-1.

The latest version of the PoC is copied below. You may even use it to generate non-malicious constants.

```
#!/usr/bin/env python
#https://github.com/veorq/numsgen
"""
Generator of "nothing-up-my-sleeve" (NUMS) constants.

This aims to demonstrate that NUMS-looking constants shouldn't be
blindly trusted.

This program may be used to bruteforce the design of a malicious cipher,
to create somewhat rigid curves, etc. It generates close to 2 million
constants, and is easily tweaked to generate many more.

The code below is pretty much self-explanatory. Please report bugs.

See also <http://safecurves.cr.yp.to/bada55.html>

Copyright (c) 2015 Jean-Philippe Aumasson <jeanphilippe.aumasson@gmail.com>
Under CC0 license <http://creativecommons.org/publicdomain/zero/1.0/>
"""

from base64 import b64encode
from binascii import unhexlify
from itertools import product
from struct import unpack
```

```
from Crypto.Hash import HMAC, MD5, SHA, SHA224, SHA256, SHA384, SHA512
from Crypto.Protocol.KDF import PBKDF2
import mpmath as mp
import sys


# add your own special primes
PRIMES = (2, 3, 5, 7, 11, 13)

PRECISIONS = (
    42, 50, 100, 200, 500, 1000,
    32, 64, 128, 256, 512, 1024,
)

# set mpmath precision
mp.mp.dps = max(PRECISIONS)+2

# some popular to-irrational transforms (beware exceptions)
TRANSFORMS = (
    mp.ln, mp.log10,
    mp.sqrt, mp.cbrt,
    mp.cos, mp.sin, mp.tan,
)


IRRATIONALS = [
    mp.phi,
    mp.pi,
    mp.e,
    mp.euler,
    mp.apery,
    mp.log(mp.pi),
] +\
[ abs(transform(prime))\
        for (prime, transform) in product(PRIMES, TRANSFORMS) ]

SEEDS = []
for num in IRRATIONALS:
    inv = 1/num
    seed1 = mp.nstr(num, mp.mp.dps).replace('.', '')
    seed2 = mp.nstr(inv, mp.mp.dps).replace('.', '')
    for precision in PRECISIONS:
        SEEDS.append(seed1[:precision])
        SEEDS.append(seed2[:precision])
    if num >= 1:
        seed3 = mp.nstr(num, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed3[:precision])
        continue
    if inv >= 1:
        seed4 = mp.nstr(inv, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed4[:precision])


# some common encodings
def int10(x):
    return x

def int2(x):
    return bin(int(x))

def int2_noprefix(x):
    return bin(int(x))[2:]
```

```
def hex_lo(x):
    xhex = '%x' % int(x)
    if len(xhex) % 2:
        xhex = '0' + xhex
    return xhex

def hex_hi(x):
    xhex = '%X' % int(x)
    if len(xhex) % 2:
        xhex = '0' + xhex
    return xhex

def raw(x):
    return hex_lo(x).decode('hex')

def base64_from_int(x):
    return b64encode(x)

def base64_from_raw(x):
    return b64encode(raw(x))

ENCODINGS = (
    int10,
    int2,
    int2_noprefix,
    hex_lo,
    hex_hi,
    raw,
    base64_from_int,
    base64_from_raw,
)


def do_hash(x, ahash):
    h = ahash.new()
    h.update(x)
    return h.digest()

def do_hmac(x, key, ahash):
    h = HMAC.new(key, digestmod=ahash)
    h.update(x)
    return h.digest()

HASHINGS = [
    lambda x: do_hash(x, MD5),
    lambda x: do_hash(x, SHA),
    lambda x: do_hash(x, SHA224),
    lambda x: do_hash(x, SHA256),
    lambda x: do_hash(x, SHA384),
    lambda x: do_hash(x, SHA512),
]

# HMACs
for hf in (MD5, SHA):
    for keybyte in ('\x55', '\xaa', '\xff'):
        for keylen in (16, 32, 64):
            HASHINGS.append(lambda x,\
                hf=hf, keybyte=keybyte, keylen=keylen:\
                do_hmac(x, keybyte*keylen, hf))

# PBKDF2s
for n in (32, 64, 128, 512, 1024, 10, 100, 1000):
    for saltbyte in ('\x00', '\xff'):
        for saltlen in (8, 16, 32):
            HASHINGS.append(lambda x,\
```

```
                n=n, saltbyte=saltbyte, saltlen=saltlen:\
156             PBKDF2(x, saltbyte*saltlen, count=n))

158
DECODINGS = (
160     lambda h: (
            unpack('>L', h[:4])[0],
162         unpack('>L', h[4:8])[0],
            unpack('>L', h[8:12])[0],
164         unpack('>L', h[12:16])[0],),
        lambda h: (
166         unpack('<L', h[:4])[0],
            unpack('<L', h[4:8])[0],
168         unpack('<L', h[8:12])[0],
            unpack('<L', h[12:16])[0],),
170 )

172
MAXNUMS =\
174     len(SEEDS) *\
        len(ENCODINGS) *\
176     len(HASHINGS) *\
        len(DECODINGS)
178

180 def main():
        try:
182         nbnums = int(sys.argv[1])
            if nbnums > MAXNUMS:
184             raise ValueError
        except:
186         print 'expected argument < %d (~2^%.2f)'\
              % (MAXNUMS, mp.log(MAXNUMS, 2))
188         return -1
        count = 0
190
        for seed, encoding, hashing, decoding in\
192         product(SEEDS, ENCODINGS, HASHINGS, DECODINGS):

194         constants = decoding(hashing(encoding(seed)))

196         for constant in constants:
                sys.stdout.write('%08x ' % constant)
198         print
            count += 1
200         if count == nbnums:
                return count
202

204 if __name__ == '__main__':
        sys.exit(main())
```