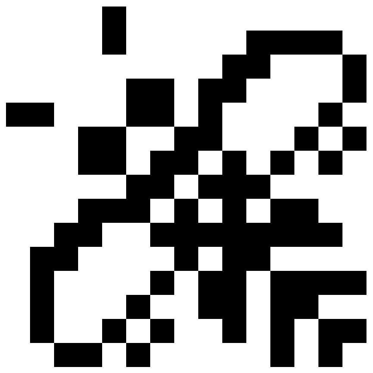
The curious case of BLATSTING's RSA implementation



Laanwj's blog

Randomness

Blog About

Among BLATSTING's modules is one named `crypto_rsa`. According to the name one'd expect it to implement the well-known asymmetric cryptosystem going under that name.

RSA

One'd expect an encryption operation

```
o = m^e (mod n)
```

and a matching decryption operation

```
o = m^d (mod n)
```

where *m* is the message, *n* is the RSA modulus (*p* times *q*), and *e* and *d* are the encryption and decryption exponent respectively, parameters computed during key generation and stored in a public or private key structure.

The interface

`crypto_rsa` offers four functions:

```
i01020000 crypto_rsa

  + 0x14: 2 args create_ctx(keyblock,size)
      Allocates a context, endian-swaps and copies the key data.

  + 0x18: 4 args encrypt(ctx, inptr, outptr, inlen)
      Performs (supposedly) RSA encryption using a fixed exponent e=65537

  + 0x1c: 0 args dummy()
      Unimplemented, just "ret"

  + 0x20: 1 args free_ctx(ctx)
      Frees the context ctx
```

Apparently the interface only offers encryption, likely used for signature verification. The keyblock is a structure of 544 bytes containing a (up to 1024 bit) RSA key, with various bignum parameters represented by arrays of 32-bit integers;

```
struct key_block {
    uint32_t n[32];    // modulus
    uint32_t unk0[32]; // Unused
    uint32_t x[32];    // ?
    uint32_t unk1[32]; // Unused
    uint32_t unk2;     // Unused
    uint32_t fudge;    // ?
    uint32_t padding[6];
};
```

The fixed exponent *e* is not encoded in this structure.

But what are those extra fields for, *x* and *fudge*?

Curioser and curioser

Here’s a Python version of what `encrypt` does:

```
def weirdmod(a, b, fudge): # function at 0x08000cd4
    # tally: 32 integer multiplications, 32 bn_muls, 32 bn_adds, 1 bn_compare, 1
bn_sub
    v = a
    for i in range(32):
        v = ((((fudge * (v&0xffffffff))&0xffffffff) * b) + v) >> 32
    if v > b:
        v -= b
    return v

# RSA according to BLATSTING
def bs_rsa_encrypt(ctx, temp): # function factored out for clarity
    # pre-multiplication
    temp = weirdmod(temp * ctx.x, ctx.n, ctx.fudge)

    # m ** 65537 mod n
    to = temp
    for i in range(16):
        to = weirdmod(to * to, ctx.n, ctx.fudge)
    temp = weirdmod(temp * to, ctx.n, ctx.fudge)

    return weirdmod(temp, ctx.n, ctx.fudge)

def bs_rsa_encrypt_outer(ctx, inptr, outptr, len): # function at 0x08000170
    temp = memcpy_bswap4_in(inptr, len)
    temp = bs_rsa_encrypt(ctx, temp)
    memcpy_bswap4_out(outptr, temp, len)
```

Broadly it looks like a RSA encrypt operation with a hard-coded exponent of `65537` (which is standard), except that an unconventional pre-multiplication with *x* is done. After each operation a mod is applied to bring the result back within the range [0..n-1].

But wait: note that `weird_mod` does not actually, as would be first expected, implement a modulus operator. I'm honestly not sure what it is. Unlike mod, applying it repeatedly to a value does not yield the same result, applying it to 1 does not yield 1. What use would they have for such a a mutilated version of RSA?

Call site

The only place where this module is used from is TADAQUEOUS, from the hooked function `__add_ipsec_sa`. It supplies the following parameters:

```
class Context:
    n =
0xd257c42f17e16815bef4c2f3fede55b5b7ed35fa4ae040aac0515a7bc662f564ac4e98272b61c24b6665

    unk0 =
0x2da83bd0e81e97ea410b3d0c0121aa4a4812ca05b51fbf553faea584399d0a9b53b167d8d49e3db4999a

    x =
0x76bc66dabca44047215cedfe4b6182cee4a9af38201d5b83ea8b3ab5ad7a05e835327be2337d8c302adb

    unk1 = 0
    unk2 = 1
    fudge = 0xbb4d023f
```

Surprise

So imagine my surprise when I tried it out and compared, using the above parameters:

```
# Conventional RSA
def rsa_encrypt(ctx, m):
    return pow(m, 65537, ctx.n)

# Try with random 1024-bit value
ctx = Context()
m = random.randint(0, (1<<1024)-1)

# Compare results
assert(rsa_encrypt(ctx, m) == bs_rsa_encrypt(ctx, m))
```

The result matches convential RSA without pre-multiplication and with a normal expmod operator! So it is some kind of optimization, but I had not seen it before, which doesn't say that much, ~~but it's not part of e.g. OpenSSL~~. *Edit: it is, according to k240df and martins_m on reddit this is Montgomery reduction which is in OpenSSL under* `crypto/bn/bn_mont.c`*. The thought came up when writing this that it was Montgomery reduction but I did not recognize it as such.*

I'm not up to date with the state of the art is with regard to efficient bignum arithmetic. Assuming 1024-bit numbers: A naive modulus implementation based on long division would take up to *1024* bignum comparisons and *1024* bignum subtractions, whereas the `weird_mod` operation takes *32* integer multiplications, *32* bignum muls, *32* bignum adds, *1* bignum compare, and *1* bignum sub. Whether it is a win depends on how bignum multiplication is implemented. A naive bignum multiplication would take up to *32*32* integer multiplications and *32* bignum adds in which case it would not really help. I have not studied the particular `bn_mul` implementation in BLATSTING (address `0x080004a0` *).

~~Independent of the performance characteristics, I think this alternative implementation is worth highlighting, as it is in things like this that the Equation Group keeps true to their name. It looks like a form of Barrett reduction, turning divisions into multiplies, and precomputing a~~

~~multiplicant over the exact number of modular reductions required.~~ Edit: apparently this is a well-known optimization called Montgomery Reduction. Disappointing, I had hoped to catch at least some crypto magic in the act.

* All mentioned memory addresses are as shown by radare2, which loads the ELF part of Firewall/BLATSTING/BLATSTING_201381/LP/lpconfig/m01020000/m01020000.impmod at 0x08000000.

Written on September 13, 2016

Tags: eqgrp malware cryptography
Filed under Reverse-engineering