Galhacktic Trendsetters

CTF writeups

33C3 CTF – Beeblebrox

Posted on January 4, 2017 by sajninredoc

This crypto challenge is a classic "fake-the-signature" crypto challenge, but with a somewhat unusual signature scheme that depends on the hardness of computing nth roots modulo a semiprime:

1. There is a publicly known semiprime $N = PQ$, whose two prime factors P and Q are known only to the signer. There is also a publicly known element S of $\mathbb{Z}/N\mathbb{Z}$, and a publicly known hash-function H which maps arbitrary-length inputs to 128-bit outputs (in our case, the 128-bit truncation of SHA-256).
2. To sign a message M, you first find a nonce r so that $H(M||r)$ equals some prime, k.
3. The signer then computes a value z that satisfies $z^k = S$ (i.e. z is a kth root of S modulo N). The signer returns z as the signature.

In our challenge, you can pass any message M along with a nonce r to the signer, who will then sign it as long as M is not the target message ("I, Zaphod Beeblebrox, hereby resign from my office as president of the Galaxy.") and $H(M||r)$ is indeed prime.

Of course, as with Ichwixnisse (https://galhacktictrendsetters.wordpress.com/2017/01/04/33c3-ctf-ichnixwisse/), there is an implementation error here, and the signer's code to check that k is prime will actually accept any odd composite number. The question is, how can we abuse this?

Well, unless we can find some hash collisions, we'll need to somehow find a kth root of S, where k is in fact of the form $H(M_0||r)$, with M_0 being the target message. For convenience, write $S^{1/k}$ to represent some kth root of S.

Since we can sign other messages, we want to be able to generate roots of S from other roots of S. Let's start with some simple cases, then. If we have a 15th root of S, $S^{1/15}$, then we can easily construct a third root of S, since $((S^{1/15})^5)^3 = (S^{1/15})^{15} = S$ (and similarly we can easily construct a fifth root of S). What about going the other way? If we're given $S^{1/3}$ and $S^{1/5}$, can we construct a fifteenth root $S^{1/15}$?

Well, we might notice that $\frac{2}{3} - \frac{3}{5} = \frac{1}{15}$, so maybe $(S^{1/3})^2 \cdot (S^{1/5})^{-3} = S^{1/15}$. And indeed, this is not hard to check:

$$((S^{1/3})^2 \cdot (S^{1/5})^{-3})^{15} = (S^{1/3})^{30} \cdot (S^{1/5})^{-45} = S^{10} \cdot S^{-9} = S.$$

More generally, if we have $S^{1/a}$ and $S^{1/b}$, with a and b relatively prime, then we can compute $S^{1/ab}$ via a similar approach to the above (more specifically, running the extended Euclidean algorithm and finding c and d such that $ac + bd = 1$).

This turns out to be all the primitives we need. Our strategy for an attack is now as follows:

1. Find a specific nonce r_0 so that $H(M_0||r_0)$ is smooth (https://en.wikipedia.org/wiki/Smooth_number), with all its prime factors as small as possible. There is a tradeoff here between the number of nonces you try and the size of the largest prime

factor you'll get; we tried around a million different nonces and found one that gave a k whose largest prime factor was 32557549.
2. Write this k as the product $k = p_1 \cdot p_2 \cdot \cdots \cdot p_m$ of a bunch of primes (our number turned out to be square-free, but this approach should work in general). For each prime p_i, generate random messages M_i and nonces r_i until you find a pair that satisfies $p_i | H(M_i || r_i)$. Set $k_i = H(M_i || r_i)$.
3. Ask the server to sign M_i with nonce r_i, and hence obtain S^{1/k_i}.
4. Since $p_i | k_i$, from S^{1/k_i}, compute S^{1/p_i}.
5. Finally, using the extended Euclidean algorithm trick above, from all the roots S^{1/p_i}, compute $S^{1/(p_1 p_2 \cdots p_m)} = S^{1/k}$. This is the signature of the target message with nonce r_0.

The tricky step here is step 1, but since the hash has 128-bit outputs, it is not so bad; Sage can factor 128-bit integers relatively quickly, and there are enough smooth 128-bit integers whose factors are all at most around a million.

Most of this approach is implemented in the below code, but part of it (in particular factoring k and step 5) was performed in a separate sage session.

```
#!/usr/bin/python3
from pwn import *
from hashlib import sha256
import base64
import random
import struct

# CONSTANTS
MODULUS = 165365077378447878926418656618634623976315221502120240910
S = 227987034908959467607813195722342752637294043534287176451034533
SEPARATOR = ";"
TARGET_MSG = "I, Zaphod Beeblebrox, hereby resign from my office as
LISTEN_ON = ('0.0.0.0', 2048)
PROOF_OF_WORK_HARDNESS = 2**23

def encode(i, length):
    i = i.to_bytes(length, 'little')
    return base64.b64encode(i)

def decode(i, min, max):
    i = base64.b64decode(i)
    i = int.from_bytes(i, 'little')
    if i < min:
        raise ValueError("i too small")
    if i >= max:
        raise ValueError("i too large")
    return i

def hash(msg, ctr):
    h = sha256(msg.encode('ASCII') + ctr.to_bytes(4, 'little'))
    h = h.digest()
    h = h[0:16]
    h = int.from_bytes(h, 'little')
    return h

def is_prime(n, c):

    if n <= 1: return False
    if n == 2 or n == 3: return True
    if n % 2 == 0: return False

    for _ in range(c):
```

```
        a = random.randrange(1, n)
        if not pow(a, n-1, n) != 1:
            return False

    return True

def extended_gcd(a, b):

    def _egcd(a, b):
        if a % b == 0:
            return b, 0, 1
        else:
            g, s, t = _egcd(b, a % b)
            assert(s * b + t * (a % b) == g)
            return g, t, s - t * (a // b)

    if a < b:
        g, d, c = _egcd(b, a)
    else:
        g, c, d = _egcd(a, b)

    return g, c, d

def modinv(a, m):

    """ compute the modular inverse of a modulo m.
    Raises an error if a does not have an inverse, (i.e. gcd(a, m)

    g, s, _ = extended_gcd(a, m)
    if g != 1:
        raise ValueError("cannot compute modular inverse of {} modul
    return s % m

def random_string(length = 10):
    characters = [chr(i) for i in range(ord('a'), ord('z') + 1)]
    characters += [chr(i) for i in range(ord('A'), ord('Z') + 1)]
    characters += [chr(i) for i in range(ord('0'), ord('9') + 1)]
    result = ""
    for _ in range(length):
        result += random.choice(characters)
    return result

def proof_of_work_okay(task, solution):
    h = sha256(task.encode('ASCII') + solution.to_bytes(4, 'little')
    return int.from_bytes(h, 'little') < 1/PROOF_OF_WORK_HARDNESS *

def solve_pow(challenge):
    sol = 0
    while not proof_of_work_okay(challenge, sol):
        sol += 1

    return encode(sol, 8)

print(decode(encode(17,4), 1, 2**32))

TMP_MSG = "signmeplease!"

def hash_msg(ctr):
    h = hash(TARGET_MSG, ctr)
    print (h, is_prime(h, 128))

```

```
def gen_hashes():
    f = open('hashes', 'w')
    for ctr in range(1, 2**15):
        if ctr%1000 == 0:
            print(ctr)
        h = hash(TARGET_MSG, ctr)
        if is_prime(h, 128):
            f.write(str(ctr) + ' ' + str(h) + '\n')
    f.close()

CTR = 28039
SMOOTH = 196938090168807626792794007952183901965
PS = [3, 5, 13, 53, 59, 257, 659, 10987, 47207, 252971, 446417, 325

def get_hashes():
    f = open('smooth', 'w')
    ctr = 1
    while len(PS)>0:
        if ctr%1000 == 0:
            print(ctr)
        h = hash(TMP_MSG, ctr)
        for p in PS:
            if h%p == 0:
                if is_prime(h, 128):
                    print("FOUND hash {} for {} (ctr: {})\n".format
                    f.write(str(ctr) + ' ' + str(p) + ' ' + str(h) -
                    PS.remove(p)
        ctr += 1

    f.close()

SHS = [map(int, line.split()) for line in open('smooth', 'r')]

def sign_message(message, ctr):
    conn = remote('78.46.224.72', 2048)
    conn.recvlines(numlines=4)
    line = conn.recvline()

    challenge = line.split()[-1][1:-1].decode()
    print(challenge)

    sol = solve_pow(challenge)

    qry = '{};{};{}'.format(sol.decode(), TMP_MSG, encode(ctr, 4).de
    print(qry)

    conn.sendline(qry)
    conn.recvline()
    ans = conn.recvline().strip()
    print(ans)

    ans = decode(ans, 2, MODULUS)
    print(ans)

    conn.close()
    return ans

def get_roots():
    f = open('smooth_roots', 'w')
    for ctr, p, h in SHS:
        print('signing {}'.format(p))
```

```
        while True:
            try:
                s = sign_message(TMP_MSG, ctr)
                break
            except Exception as e:
                pass
        assert pow(s, h, MODULUS) == S
        f.write('{} {} {} {}\n'.format(ctr, p, h, s))

SIG = 1004671045319865386994571316549662361082032322181462266683999

def resign():
    conn = remote('78.46.224.72', 2048)
    conn.recvlines(numlines=5)

    conn.sendline('Resign!')

    line = conn.recvline()
    print(line)
    challenge = line.split()[-1][1:-1].decode()
    print(challenge)

    sol = solve_pow(challenge)

    qry = '{};{};{};{}'.format(sol.decode(), TARGET_MSG, encode(CTR,
    conn.sendline(qry)

    print(conn.recvline())
    print(conn.recvline())
    print(conn.recvline())
    print(conn.recvline())


resign()
```

Posted in 33C3 CTFTagged crypto

Blog at WordPress.com.