# CPA-Secure Diffie–Hellman?

Christopher Wrogg, cwrogg, christheyankee, OPz qt

March 27 2021

## Background

This challenge did not see as much traffic as some of the other challenges and that saddened me because I think this was one of the deepest theoretical challenges we had to offer. I wanted to show that Diffie-Hellman key exchanges can be done be very insecure if you are not careful with how you choose your groups and as of the writing of this paper there are actual systems that implement and use convoluted versions of this form of Diffie-Hellman. This is by no means an endorsement to go and attack those systems nor should you from an ethical standpoint. That said it saddened me to find these cryptographic systems in the wild and to spread awareness I wrote this challenge. Without further ado lets get into the math.

## The Theory

First lets discuss formally what it means for a system to be CPA-Secure. CPA here stands for chosen plain-text attack. In the previous example where Eve is only allowed to query the encryption oracle once, in CPA schemes we allow Eve to query the encryption oracle a finite number of times and then ask her to decide whether the oracle is outputting uniform random bits or our bits from our encryption scheme. This Diffie-Hellman scheme is in fact EAV-Secure (unlike our last challenge) but it is in fact, nor CPA-Secure as the challenge name might suggest.

## The Exploit

So lets get into what's actually going on here. You should immediately try to factor $p-1$ first and notice that it has many small factors. What might tip us off to try this? In general undoing the discrete log is known not to be in NP when $p-1$ has many small factors. So we want to check this. Here we have many small factors. Since there are more than just two small factors (and some of them are quite large) we should start to try to create sub-groups with orders that are factors of our $p-1$ group and then brute force those. We will in general obtain a partial solution $x'$ that is equivalent to $x \mod q$ if $q$ is the order of our sub-group.

Let's formalize this a bit. We can guarantee the existence of $q$ who is the order of $g$ in the group $\mathbb{Z}_p$. Further by Lagrange's theorem we know that $q$ must divide $p-1$. Well that's great for us because we only have a few values of $q$ to search through. Once we find that $q$ (spoiler its the largest one) we can go over the remaining factors of $p-1$ in the Pohlig-Hellman algorithm to recover little $a$. Now I chose $p$ so that it would have enough small factors to recover $b$ here. It's critical to notice though that if there are less small factors you can recover part of little $a$ trivially with the same algorithm. So lets go over the algorithm in more detail. Pick one of the other small factors and call it $r$. We then compute $h := rand(1,p)^(p-1)/r)modp$ until $h \neq 1$. Notice that $h$ is not a valid key (it generates the group $\mathbb{Z}_r$ and not $\mathbb{Z}_p$) but we send it to Bob anyway. Bob sends us his signature using $B = h^b \mod p$, $MAC(K, m)$ where $K = g^{x \cdot b} \mod p$. Guess what? Bob has just generated the shared secret in our sub-group $\mathbb{Z}_r$ and consequently we can brute force all of the possible values of $t = MAC(K, m)$ since there are only $r$ values of $K$ he could have chosen (recall $\mathbb{Z}_r$ has order $r$). Repeat the process until $b$ is the flag. The script is included on the next page for convenience.

```
import random
import hashlib
import hmac
import requests
from Crypto.Util.number import bytes_to_long, long_to_bytes

key = {
    '0' : 0 , '1' : 1 , '2' : 2 , '3' : 3 , '4' : 4 , '5' : 5 , '6' : 6 ,
    'a' : 70, 'b' : 71, 'c' : 72, 'd' : 73, 'e' : 74, 'f' : 75,
    'g' : 76, 'h' : 77, 'i' : 78, 'j' : 79,
    'k' : 80, 'l' : 81, 'm' : 82, 'n' : 83, 'o' : 84, 'p' : 85,
    'q' : 86, 'r' : 87, 's' : 88, 't' : 89,
    'u' : 90, 'v' : 91, 'w' : 92, 'x' : 93, 'y' : 94, 'z' : 95,
    '_' : 96, '#' : 97, '$' : 98, '!' : 99,
}

def long_to_bytes_flag(long_in):
    new_map = {v: k for k, v in key.items()}
    list_long_in = [int(x) for x in str(long_in)]
    str_out = ''
    i = 0
    while i < len(list_long_in):
        if list_long_in[i] < 7:
            str_out += new_map[list_long_in[i]]
        else:
            str_out += new_map[int(str(list_long_in[i])
            str_out +=str(list_long_in[i + 1]))]
            i += 1
        i += 1
    return str_out.encode("utf_8")

p = redacted
g = redacted
q = 0
p_factors = factor(p - 1)
for (candidate, _) in p_factors:
    if pow(g, candidate, p) == 1:
        q = candidate
        break
j = (p-1) / q
r_list = factor(j)
r_i = []
b_i = []
total = 1
for (r, r_p) in r_list:
    if r_p != 1 or r < 10 or r == 521:
        continue
    print("r: ", r)
    A = 1
    while A == 1:
        ra = random.randint(1, p)
        o = int((p-1)/r)
        A = pow(ra, o, p)
    query = {'A':A}
    resp = requests.get(url = 'url', params = query)
    target = int(resp.text)
    print(target)
    message = b'My totally secure message to Alice'
    i = int(1)
    first_guess = long_to_bytes(int(pow(A, i, p)))
    guess = 0
    my_hmac = hmac.new(message, first_guess, hashlib.sha256)
    while (int(target) != bytes_to_long(my_hmac.digest()) and
    first_guess != guess):
        i += int(1)
        guess = pow(A, i, p)
        my_hmac = hmac.new(msg=message,
        key=long_to_bytes(int(guess)), digestmod=hashlib.sha256)
    r_i.append(Integer(r))
    b_i.append(Integer(i))
    total *= r
    candidate_a = CRT_list(b_i, r_i)
    if total > q:
        break
    print("r_i: ", r_i)
    print("b_i: ", b_i)
    print("total", total)
    print("q:", q)
    print("cadidate a:", candidate_a)
a = CRT_list(b_i, r_i)
print(a)
print(long_to_bytes_flag(a))
```