Cryptohack Diffie-Hellman

-

Challenge Room Walkthrough

Paweł Murdzek

Warsaw University of Technology

December 3, 2025

Contents

Introduction

This document is a walkthrough for the "Diffie-Hellman" challenge category on the Cryptohack platform.

1. Starter

1.1. Working with Fields

— **Code:**

```python
from Crypto.Util.number import inverse

p = 991
g = 209
d = inverse(g, p)
print(d)
```

— **Flag:** 569

— **Explanation:**

The task is to calculate the modular inverse of g = 209 modulo p = 991. We use the `inverse()` function from the PyCryptodome library, which finds a number d such that $(g \cdot d) \bmod p = 1$. In this case, the result is 569.

1.2. Generators of Groups

— **Code:**

```python
from sympy import primerange

def is_primitive_root(g, p):
    for q in primerange(2, p):
        if (p - 1) % q == 0 and pow(g, (p - 1) // q, p) == 1:
            return False
    return True

p = 28151

for g in range(2, p):
    if is_primitive_root(g, p):
        print(g)
        break
```

— **Flag:** 7

— **Explanation:**

The task requires finding the smallest primitive root for p = 28151. A primitive root is a generator of the multiplicative group modulo p. The `is_primitive_root()` function checks whether for every prime q dividing (p-1), $g^{(p-1)/q} \bmod p \neq 1$. The smallest primitive root for 28151 is 7.

1.3. Computing Public Values

— **Code:**

```python
g =2
p=24103124269210325885520760221975660748569505485024599426541169419581088316826122288900938582613416146732271414779040121965036489570505826319427307068050092230627347453410734066962460145893616597740
a=9721074438370337962458643162004582468469045984889816058567658904788530882468973454873284910377102192220389309433658486261941098303091793930182167633275721201247601400180386739998376433775904344138
print(pow(g,a,p))
```

— **Flag:** 18068576978407265233225867218209113584894201281292480786739336535339306816761817538494117157141

— **Explanation:**

This task illustrates the first step of the Diffie-Hellman protocol - computing the public value. Given a generator g = 2, a large prime p, and a private key a, we compute the public value as $A = g^a \bmod p$. The `pow(g, a, p)` function efficiently calculates modular exponentiation.

1.4. Computing Shared Secrets

— **Code:**

```python
p = 2410312426921032588552076022197566074856950548502459942654116941958108831682612228890093858261341614673227141477904012196503648957050582631942730706805009223062734745341073406696246014589361659 7
A = 70249943217595468278554541264975482909289174351516133994495821400710625291840101960595720462672604202133493023241393916394629829526272643847352371534839862030410331485087487331809285533195024369
b = 12019233252903990344598522535774963020395770409445296724034378433497976840167805970589960962221948290951873387728102115996831454482299243226839490999713763440412177965861508773420532266484619126
print(pow(A,b,p))
```

— **Flag:** 11741307404138206565338327460348419858773020863163883801659844366723076924437113102850141385452

— **Explanation:**

The second step of Diffie-Hellman - computing the shared secret. Given the other party's public value A and our own private key b, we compute the shared secret as $s = A^b \bmod p$. Both parties (Alice with key a and Bob with key b) will obtain the same secret: $g^{ab} \bmod p$.

1.5. Deriving Symmetric Keys

— **Code:**

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import hashlib

def decrypt_flag(shared_secret, iv, ciphertext):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]

    cipher = AES.new(key, AES.MODE_CBC, bytes.fromhex(iv))
    plaintext = cipher.decrypt(bytes.fromhex(ciphertext))
    return unpad(plaintext, 16).decode('ascii')

p = 24103124269210325885520760221975660748569505485024599426541169419581088316826122288900938582613416146732271414779040121965036489570505826319427307068050092230627347453410734066962460145893616597
A = 11221873913954290888056435953437342401301624977293196269223790757199033448352887751380927262561051206115906173760854728855866287968508668429962448174286501692406500055526797783014474036446797720
b = 19739508381490702899178577271492088590824934192565095155521904941129843621719060519082493478733627922878580978353181450766138511122063932935804819633962606567686911973797917553177076886180858111

shared_secret = pow(A, b, p)

iv = '737561146ff8194f45290f5766ed6aba'
ciphertext = '39c99bf2f0c14678d6a5416faef954b5893c316fc3c48622ba1fd6a9fe85f3dc72a29c394cf4bc8aff6a7b21cae8e12c'

print(decrypt_flag(shared_secret, iv, ciphertext))
```

— **Flag:** `crypto{sh4r1ng_s3cret5_w1th_fr13nd5}`

— **Explanation:**

This demonstrates the complete Diffie-Hellman flow: computing the shared secret, then deriving a symmetric AES key using SHA-1 (first 16 bytes). The key is used to decrypt the flag using AES-CBC. This is the standard way to use DH to establish a secure communication channel.

2. Man In The Middle

2.1. Parameter Injection

— **Code:**

```python
import json
import pwn
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import hashlib

def is_pkcs7_padded(message):
    padding = message[-message[-1]:]
    return all(padding[i] == len(padding) for i in range(0, len(padding)))

def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)
    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('ascii')
    else:
        return plaintext.decode('ascii')

def main():
    remote = pwn.remote("socket.cryptohack.org", 13371)
    remote.recvuntil("Intercepted from Alice: ")
    intercepted_from_alice = json.loads(remote.recvline())
    intercepted_from_alice['p'] = "1"
    remote.recvuntil("Send to Bob: ")
    remote.sendline(json.dumps(intercepted_from_alice))
    remote.recvuntil("Intercepted from Bob: ")
    remote.sendline(remote.recvline())
    remote.recvuntil("Intercepted from Alice: ")
    alice_ciphertext = json.loads(remote.recvline())
    shared_secret = 0
    flag = decrypt_flag(shared_secret, alice_ciphertext["iv"], alice_ciphertext["encrypted_flag"])
    pwn.log.info(flag)

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

— **Flag:** `crypto{n1c3_0n3_m4ll0ry!!!!!!!!}`

— **Explanation:**

A Man-in-the-Middle attack through manipulation of the parameter p. By setting $p = 1$, we force both parties to compute the shared secret as 0 (since any number mod $1 = 0$). Knowing the secret (0), we can decrypt the communication. This demonstrates the critical importance of verifying DH parameters.

2.2. Export-grade

— **Code:**

```python
from sage.all import *
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import json, codecs, hashlib

def is_pkcs7_padded(message):
```

```
    padding = message[-message[-1]:]
    return all(padding[i] == len(padding) for i in range(0, len(padding)))

def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)
    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('ascii')
    else:
        return plaintext.decode('ascii')

alice = {"p": "0xde26ab651b92a129", "g": "0x2", "A": "0x9bf1d8558e7b6768"}
bob = {"B": "0x97e38f7cb7602367"}
flag = {"iv": "427517171ebdc1eb2676462b29c82cab", "encrypted_flag": "a515316381f3af3b965172c99c8a717d5972f2965fb0929245ce42874c15b1b0"}

R = GF(alice["p"])
g = R(alice["g"])
A = R(alice["A"])
B = R(bob["B"])

n = discrete_log(A, g)
print(n)

shared = B**n

print(decrypt_flag(shared, flag["iv"], flag["encrypted_flag"]))
```

- **Flag:** `crypto{d0wn6r4d35_4r3_d4n63r0u5}`
- **Explanation:**
  An attack on weak DH parameters (small p). When the prime p is too small (64-bit), we can use the discrete logarithm algorithm from SageMath to find the private key a from the public value A. Then we compute the shared secret and decrypt the flag. This illustrates the "export-grade" attack - forcing weak parameters.

2.3. Static Client

- **Code:**

```
from pwn import remote
import json, hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

def decrypt_flag(shared_secret, iv, ciphertext):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    cipher = AES.new(key, AES.MODE_CBC, bytes.fromhex(iv))
    plaintext = cipher.decrypt(bytes.fromhex(ciphertext))
    return unpad(plaintext, 16).decode('ascii')

r = remote('socket.cryptohack.org', 13373)
data = json.loads(r.recvline().decode().split("Alice: ")[1])
p_hex = data['p']
g_hex = data['g']
A_hex = data['A']
r.recvline()
data = json.loads(r.recvline().decode().split("Alice: ")[1])
iv = data["iv"]
encrypted = data["encrypted"]
# Send manipulated parameters: swap g and A
r.sendline(json.dumps({"p": p_hex, "g": A_hex, "A": g_hex}).encode())
# Get B which is actually the shared secret
shared_secret = int(json.loads(r.recvline().decode().split("you: ")[1])["B"], 16)
print(decrypt_flag(shared_secret, iv, encrypted))
```

- **Flag:** `crypto{n07_3ph3m3r4l_3n0u6h}`
- **Explanation:**
  An attack based on swapping parameters g and A. When Bob receives g = A and computes $B = A^b$, and then we send him A = g, his shared secret will be equal to $g^b = B$. The value B is public, so we know the shared secret without knowing the private keys. This demonstrates the importance of using ephemeral (one-time) keys.

3. Group Theory

3.1. Additive

- **Code:**

```
from pwn import remote
import json, hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from Crypto.Util.number import inverse

def decrypt_flag(shared_secret, iv, ciphertext):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    cipher = AES.new(key, AES.MODE_CBC, bytes.fromhex(iv))
    plaintext = cipher.decrypt(bytes.fromhex(ciphertext))
    return unpad(plaintext, 16).decode('ascii')
```

```python
r = remote('socket.cryptohack.org', 13380)
data = json.loads(r.recvline().decode().split("Alice: ")[1])
p = int(data['p'], 16)
g = int(data['g'], 16)
A = int(data['A'], 16)
B = int(json.loads(r.recvline().decode().split("Bob: ")[1])['B'], 16)
data = json.loads(r.recvline().decode().split("Alice: ")[1])
iv = data["iv"]
encrypted = data["encrypted"]
# Calculate shared secret
a = A * inverse(g, p) % p
shared_secret = B * a % p
print(decrypt_flag(shared_secret, iv, encrypted))
```

— **Flag:** `crypto{cycl1c_6r0up_und3r_4dd1710n?}`

— **Explanation:**

An attack on a flawed DH implementation using addition instead of multiplication. When Alice uses $A = g \cdot a \bmod p$ (modular addition), we can recover the private key a as $a = A \cdot g^{-1} \bmod p$. Then we compute the shared secret as $s = B \cdot a \bmod p$. This shows that DH requires multiplicative operations, not additive ones.

3.2. Static Client 2

— **Code:**

```python
from pwn import remote
from json import loads, dumps
from ecc_decrypt import decrypt_flag

io = remote("socket.cryptohack.org", 13378)
A = loads(io.recvline().split(b":", 1)[1])
B = loads(io.recvline().split(b":", 1)[1])
cipher = loads(io.recvline().split(b":",1)[1])

p = int(A["p"], 16)
i = 2
p2 = 1
while p2 < p or not is_prime(p2 + 1):
    p2 *= i
    i += 1
p2 += 1
# p2 = 3**1000
assert p2 > p
assert is_prime(p2)
io.sendline(dumps({"p": hex(p2), "g": "0x2", "A": A["A"]}))
reply = loads(io.recvline().split(b":", 2)[2])
print(reply)
b = discrete_log(Zmod(p2)(int(reply["B"], 16)), Zmod(p2)(2))
assert(int(Zmod(p)(2)^b) == int(B["B"], 16))

print(decrypt_flag(pow(int(A["A"], 16), b, p), cipher["iv"], cipher["encrypted"]))
cipher = loads(io.recvline().split(b":",1)[1])
print(decrypt_flag(0, cipher["iv"], cipher["encrypted"]))
```

— **Flag:** `crypto{uns4f3_pr1m3_sm4ll_oRd3r}`

— **Explanation:**

An advanced attack combining two MITM techniques. First, we construct a new prime p2 with a small multiplicative order (product of small primes), which is larger than the original p. We send Bob modified parameters with this p2. Since p2 has a small order, we can use the discrete logarithm algorithm in SageMath (`discrete_log`) to find Bob's private key b from his public value B. Knowing b, we compute the shared secret between Alice and Bob: $s = A^b \bmod p$ (using the original p!) and decrypt the first message.
Then the server sends a second encrypted message. In the second round, we use the attack from Parameter Injection (forcing p=1 or exploiting the fact that the shared secret is 0), which allows us to decrypt the second flag. This attack shows how dangerous it is to reuse a static client key and the lack of DH parameter verification.

4. Misc

4.1. Script Kiddie

— **Code:**

```python
p = 24103124269210325885520760221975660748569505485024599426541169419581088316826122288900938582613416146732271414779040121965036489570505826319427307068050092230627347453410734066962460145893616597
g = 2
A = 5395560198687560190356154870625837645450198037936357129475284638893044868694971620613359975279719770500493374641524784792659921277497801032594204005649068958970775123596287606562270840392152100 3
B = 6528886768094662564069046538863130232886090752627487181350453557860287836111823799191303471652011998767624005234130299086308058885675784141099832285901887581712594205668303747935408919379044023 8
iv = 'c044059ae57b61821a9090fbdefc63c5'
encrypted_flag = 'f60522a95bde87a9ff00dc2c3d99177019f625f3364188c1058183004506bf96541cf241dad1c0e92535564e537322d7'

b = B ^ g  # XOR instead of exponentiation!
secret = A ^ b

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import hashlib

def is_pkcs7_padded(message):
    padding = message[-message[-1]:]
```

```
    return all(padding[i] == len(padding) for i in range(0, len(padding)))

def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)
    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('ascii')
    else:
        return plaintext.decode('ascii')

print(decrypt_flag(secret, iv, encrypted_flag))
```

— **Flag:** `crypto{b3_c4r3ful_w1th_y0ur_n0tati0n}`

— **Explanation:**
An attack exploiting a notation mistake. The ^ operator in Python is XOR, not exponentiation! The flawed implementation uses $b = B \oplus g$ and $s = A \oplus b$ instead of modular operations. XOR is reversible and does not provide DH security. This is a warning against literally copying pseudocode without understanding the operators in a given language.

4.2. Matrix

— **Code:**

```
from sage.all import Matrix, GF

P = 2
N = 50
E = 31337

def bits2bytes(bits):
    byte_array = []
    for i in range(0, len(bits), 8):
        byte_array.append(int(bits[i:i + 8], 2))
    return bytes(byte_array)

def matrix2bytes(matrix):
    bit_sequence = []
    for col_index in range(N):
        bit_sequence.extend([str(row[col_index]) for row in matrix])
    return bits2bytes("".join(bit_sequence)[:272])

# [flag_data matrix - shortened for readability]
rows = [...]
matrix = Matrix(GF(P), rows)

print("Computing multiplicative order...")
order = matrix.multiplicative_order()
print(f"Multiplicative order: {order}")

# C = M^E => M = C^(1/E % k)
dec_exponent = pow(E, -1, order)
print(f"Decryption exponent: {dec_exponent}")

dec_matrix = matrix ** dec_exponent
flag = matrix2bytes(dec_matrix).decode('utf-8')
print(f"\nFlag: {flag}")
```

— **Flag:** `crypto{there_is_no_spoon_66eff188}`

— **Explanation:**
An advanced DH variant using matrices instead of numbers. The flag is encrypted as $C = M^E$ in the matrix group over $GF(2)$. To decrypt, we compute the multiplicative order of the matrix k, then the decryption exponent as $d = E^{-1} \bmod k$, and finally $M = C^d$. This shows that DH can be implemented in any cyclic group, not just numbers modulo p.