CTF Write-up: Augury Crypto Challenge

Challenge: Augury

Category: Cryptography

Difficulty: Beginner/Intermediate

solver: valague

Flag: bctf{pr3d1c7_7h47_k3y57r34m}

Challenge Description

This file storage uses only the most secure encryption techniques.

Service: augury.challs.pwnoh.io:1337 (SSL)

Files provided: main.py

Initial Analysis

The challenge provides a Python file main.py that implements a simple file storage service with "encryption". The service allows users to:

1.Upload files with encryption

2.View stored files (in encrypted form)

3.Exit

The encryption scheme uses:

- SHAKE-128 for initial key derivation (but only 4 bytes)
- A Linear Congruential Generator (LCG) for keystream generation
- XOR encryption with the keystream

The Vulnerability

Weak Key Derivation

python

```
m = hashlib.shake_128()
m.update(password.encode())
keystream = int.from_bytes(m.digest(4), byteorder="big")
```

Only 4 bytes from SHAKE-128 are used, significantly reducing the keyspace.

Predictable Keystream

python

```
def generate_keystream(i):
    return (i * 3404970675 + 3553295105) % (2 ** 32)
```

This LCG is completely predictable. Once an attacker knows one keystream value, they can predict all future values using the formula.

The Attack

Step 1: Service Interaction

First, connect to the service and explore available files:

python

```
from pwn import *

r = remote("augury.challs.pwnoh.io", 1337, ssl=True)
```

The service reveals one file: secret_pic.png - this is our target.

Step 2: Known Plaintext Attack

PNG files have a known header: 89 50 4E 47 0D 0A 1A 0A

We can recover the initial keystream by XORing the encrypted file's beginning with the known PNG header:

python

```
png_header = bytes.fromhex("89504E470D0A1A0A")

encrypted_start = bytes.fromhex(encrypted_hex[:16])

keystream_start = bytearray()

for i in range(8):

    keystream_start.append(encrypted_start[i] ^ png_header[i])
```

Step 3: Keystream Prediction

Using the recovered initial keystream value and the LCG formula, we can predict the entire keystream:

python

```python
def generate_keystream(i):
    return (i * 3404970675 + 3553295105) % (2 ** 32)

class AuguryCracker:
    def __init__(self, known_keystream_int):
        self.current_keystream = known_keystream_int

    def get_next_key_bytes(self):
        key_bytes = self.current_keystream.to_bytes(4, byteorder='big')
        self.current_keystream = generate_keystream(self.current_keystream)
        return key_bytes
```

Step 4: Full Decryption

With the predictable keystream, decrypt the entire file:

python

```python
def decrypt(self, encrypted_hex):
    encrypted_bytes = bytes.fromhex(encrypted_hex)
    decrypted = bytearray()

```

```
        for i in range(0, len(encrypted_bytes), 4):
            key_bytes = self.get_next_key_bytes()
            chunk = encrypted_bytes[i:i+4]

            for j in range(len(chunk)):
                decrypted.append(chunk[j] ^ key_bytes[j])

        return decrypted
```

Solution Script

python

```
from pwn import *


def generate_keystream(i):
    return (i * 3404970675 + 3553295105) % (2 ** 32)


class AuguryCracker:
    def __init__(self, known_keystream_int):
        self.current_keystream = known_keystream_int

    def get_next_key_bytes(self):
        key_bytes = self.current_keystream.to_bytes(4, byteorder='big')
        self.current_keystream = generate_keystream(self.current_keystream)
        return key_bytes

```

```
    def decrypt(self, encrypted_hex):
        encrypted_bytes = bytes.fromhex(encrypted_hex)
        decrypted = bytearray()
        
        for i in range(0, len(encrypted_bytes), 4):
            key_bytes = self.get_next_key_bytes()
            chunk = encrypted_bytes[i:i+4]
            
            for j in range(len(chunk)):
                decrypted.append(chunk[j] ^ key_bytes[j])
        
        return decrypted

# Get encrypted data from service
r = remote("augury.challs.pwnoh.io", 1337, ssl=True)
r.recvuntil(b"Exit")
r.sendline(b"2")
r.recvuntil(b"Choose a file to get")
r.sendline(b"secret_pic.png")

encrypted_data = b""
while True:
    try:
        chunk = r.recv(4096, timeout=1)
        if not chunk: break
```

```
            encrypted_data += chunk
            if b"Please select an option:" in chunk: break
    except: break

# Extract hex data
import re
hex_clean = re.sub(r'[^0-9a-fA-F]', '', encrypted_data.decode('latin-1'))

# Recover keystream using PNG header
png_header = bytes.fromhex("89504E470D0A1A0A")
encrypted_start = bytes.fromhex(hex_clean[:16])
keystream_start = bytearray()
for i in range(8):
    keystream_start.append(encrypted_start[i] ^ png_header[i])

first_keystream_int = int.from_bytes(keystream_start[:4], byteorder='big')

# Decrypt
cracker = AuguryCracker(first_keystream_int)
decrypted_data = cracker.decrypt(hex_clean.lower())

with open('flag.png', 'wb') as f:
    f.write(decrypted_data)
```

Flag

The decrypted PNG file contains the flag:

bctf{pr3d1c7_7h47_k3y57r34m}

Lessons Learned

Never use predictable PRNGs for cryptography - LCGs are completely unsuitable for cryptographic purposes

Use cryptographically secure random number generators for keystream generation

Don't truncate cryptographic hashes unnecessarily - using only 4 bytes from SHAKE-128 severely weakens the system

Stream ciphers require unpredictable keystreams - predictability breaks the entire security model

Mitigation

To fix this vulnerability:

Use a cryptographically secure PRNG (like os.urandom() or secrets module)

Don't truncate hash outputs unnecessarily

Consider using established encryption algorithms (AES in stream mode) instead of custom implementations

This challenge perfectly demonstrates why "rolling your own crypto" is dangerous and why predictable randomness breaks stream ciphers completely.