2025

CTF INTERNACIONAL METARED ETAPA MEXICO

WRITEUP

CHALLENGE

Info Stealer

ANUIES ANUIES - TIC metared MÉXICO

Description

A new software tool, dubbed "Info Stealer," has been detected. A criminal group is using it to commit identity fraud and sell stolen data on the dark web. The tool can read and encrypt victims' personal information, but the encryption key is a closely guarded secret.

. **File:** info_stealer, exfiltrated-data.dat

Solution:

0) Quick summary of the challenge

You are given a binary (no source), and file with exfiltrated data encoded. The program reads `user_data.txt`, processes 24 bytes starting at offset 7, encodes and XOR-encrypts them and writes the result to `exfiltred_data.dat`. The goal: from the delivered binary and the produced `exfiltred_data.dat`, recover the original 24-byte substring (and thus the part of the secret that was exfiltrated).

---

1) Static inspection (how to quickly find what the binary does)

1. Use `strings`, `objdump`, `readelf` / `nm` or a decompiler (Ghidra/IDA/radare2) to inspect constants and functions inside the binary. Look for:
   - File names like `user_data.txt` and `exfiltred_data.dat`.
   - Any static arrays that resemble keys (an array of bytes).
   - Functions named or behaving like `encode` or `encrypt`.
2. From these artifacts you can reconstruct the two steps applied to the data:
   - An **encode** transform that modifies each plaintext byte depending on its index parity.
   - An **encrypt** step that XORs with a repeating key; but before use the binary *mutates* the static key with a fixed XOR (so the runtime key differs from the static bytes in the binary).

(When reversing the provided challenge, these exact behaviours were revealed in the binary)

---

2) The algorithm (precise, so you can invert it)

1. The program reads 32 bytes from `user_data.txt` (the "flag" buffer), then takes a **substring starting at offset 7** of length **24** (`flag[7]` .. `flag[30]`) — that 24-byte substring is processed.
2. **Encode step** (applied to each byte of the 24-byte substring, index `i` from `0` to `23`):
   - if `i % 2 == 0` (even index): `encoded[i] = original[i] + 3`
   - else (odd index): `encoded[i] = original[i] - 2`
3. **Encrypt step**:
   - A static byte array is present in the binary (the initial bytes are `0x49, 0x5F, 0x59, 0x48, 0x5F, 0x4E, 0x51, 0x5F, 0x43`).
   - At runtime the program XORs each element of that array with `0x1A` (in place) before using it. After that transformation the key bytes become ASCII for `SECRETKEY` (i.e. the runtime key is the string `"SECRETKEY"`).
   - The program XORs each `encoded[i]` with `key[i % key_len]` (key_len = 9).

So the encryption is: `cipher[i] = encoded[i] ^ key[i % 9]` with key == `"SECRETKEY"` at runtime.

To **decrypt**, reverse the operations in the opposite order:

- XOR ciphertext with the same repeating key to get `encoded`.
- Reverse the encode: for even positions subtract 3; for odd positions add 2 — yielding the original substring.

---

3) Recovering the key (how you would discover the key from the binary)

When reversing the binary you will find the static byte array. You will also see the code that XORs each byte of that array with `0x1A`. Compute `static_byte ^ 0x1A` for each element — those results are printable ASCII and spell `SECRETKEY`. That is the actual repeating XOR key used at runtime.

(Example: `0x49 ^ 0x1A = 0x53` = `'S'`, etc.)

---

4) Practical steps to solve (command sequence)

Assume you have:

- the binary (`info_stealer`) — what you distributed to students
- the produced `exfiltred_data.dat` (24 bytes) — captured by the "network monitoring" or given as challenge output

Suggested commands to run:

```
# 1) Inspect binary for strings and byte arrays
strings info_stealer | egrep "user_data|exfiltred|SECRET|KEY"
# or hexdump to look for static byte array
hexdump -C info_stealer | less

# 2) If available, load in a decompiler/IDA/Ghidra to inspect the
function that performs the xor with 0x1a.
# 3) After confirming the key is formed by XORing the static array with
0x1a,
#    compute the runtime key -> "SECRETKEY" (9 bytes).
```

---

5) Decryption script (ready-to-run)

Save the following as `decrypt_exfil.py`. It reads `exfiltred_data.dat` (24 bytes), decrypts and decodes the substring and prints it.

```
#!/usr/bin/env python3

# decrypt_exfil.py
# Usage: python3 decrypt_exfil.py exfiltred_data.dat

import sys

def build_runtime_key():
    # original static bytes found in the binary (as little-endian bytes)
    static = [0x49, 0x5F, 0x59, 0x48, 0x5F, 0x4E, 0x51, 0x5F, 0x43]
    # runtime key bytes are static XOR 0x1A
    runtime = bytes([b ^ 0x1A for b in static])
    return runtime  # this will be b'SECRETKEY'

def decrypt(cipher_bytes):
    key = build_runtime_key()
    key_len = len(key)
    # first reverse the XOR step
    encoded = bytearray(len(cipher_bytes))
    for i, c in enumerate(cipher_bytes):
        encoded[i] = c ^ key[i % key_len]

    # then reverse the encode transform
    original = bytearray(len(encoded))
    for i, b in enumerate(encoded):
        if i % 2 == 0:
            # encoded was original + 3
            original[i] = (b - 3) & 0xFF
        else:
            # encoded was original - 2
            original[i] = (b + 2) & 0xFF

    return bytes(original)

def main():
    if len(sys.argv) != 2:
        print("Usage: python3 decrypt_exfil.py exfiltred_data.dat")
        sys.exit(1)

    fname = sys.argv[1]
    with open(fname, "rb") as f:
        data = f.read()

    if len(data) != 24:
        print(f"Warning: expected 24 bytes but read {len(data)} bytes. Proceeding
anyway.")

    recovered = decrypt(data)
    print("Recovered substring (24 bytes):")
    try:
        print(recovered.decode('utf-8'))
    except UnicodeDecodeError:
        # show hex if non-printable
        print(recovered.hex())


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

Make it executable:

```
chmod +x decrypt_exfil.py
./decrypt_exfil.py exfiltred_data.dat
```

6) Example interpretation of output

- The script prints the recovered 24-byte substring `flag[7:31]`.
- The original binary only exfiltrated bytes 7..30 (24 bytes). The first 7 and last 1 bytes of the 32-byte "flag" are not in `exfiltred_data.dat`.
- If your challenge intended the whole 32-byte value to be the flag, you must also provide context for the remaining bytes (e.g., they could be known constants, or the flag format could be `flagmx{...}` so students can deduce the missing leading/trailing bytes). Otherwise, the captured substring is the intended capture.

7) Notes

- **Why was the key obfuscated?** The binary stores a static array which is then XORed with `0x1A`, a tiny layer of obfuscation. The correct runtime key is `SECRETKEY`, which is printable and obvious once you reverse the tweak.
- **Alternative reversing route:** Instead of fully reversing the binary, if you have multiple `user_data.txt` inputs and their corresponding `exfiltred_data.dat` outputs you could use chosen-plaintext attacks to learn how bytes change and deduce the transformations. For example, giving a `user_data.txt` of known ASCII and observing output reveals the encode pattern and XOR key. This is an important practical technique in reversing/crypto challenges.
- **Edge cases:** The encode step wraps by bytes (it uses `char` arithmetic), so when reversing use `(b - 3) & 0xFF` or Python byte arithmetic (the script above uses that). If the data contains non-ASCII, print hex.

8) Final remarks (flags and scoring)

- If your score requirement is to recover the entire 32-byte flag, ensure you provide either:
    - the first 7 + last byte via other hints in the challenge, or
    - a way for students to brute-force the missing bytes (e.g., limited charset / short length).
- If recovering the 24-byte substring is the objective, then the script above is a complete solution.

Get Flag:

```
python3 decrypt_exfil.py exfiltred_data.dat

Recovered substring (24 bytes):

d4t4_pr1v4cy_m4tt3r5!!!!
```

Final Flag:

```
flagmx{d4t4_pr1v4cy_m4tt3r5!!!!}
```