

FLARE-On 13 Challenge 5: CatThief

By Nick Taylor (@NickTaylorRE)

Overview

CatThief is a compiled Windows binary (`catthief.exe`) written in Rust paired with a network packet capture (`capture.pcapng`). The challenge functions as a lightweight exfiltration utility: upon execution, it performs an initial check-in with a remote Command and Control (C2) server and subsequently receives commands instructing it to exfiltrate local files from the victim's host. The goal is to reverse the binary's custom compression and encryption layers to extract the exfiltrated files from the PCAP and recover the hidden flag: `th3r3_b3_tr34sure_1n_th4t_pc4p@flare-on.com`.

To evade trivial network inspection and reduce exfiltration size, `catthief.exe` employs a custom multi-stage encoding and cryptographic pipeline consisting of:

- **Canonical Huffman Compression:** A custom implementation that computes character frequencies, constructs a Huffman tree, serializes symbol code lengths, and encodes the data into a variable-length bitstream.
- **Bit-Packed Framing:** A custom binary header packing an 11-bit tree length and a 32-bit compressed data length followed by raw byte payloads.
- **RC4 Stream Cipher:** A symmetric cipher layer keyed with an obfuscated 25-byte string to encrypt all HTTP `POST` exfiltration traffic and C2 command responses.

By reverse-engineering **CatThief**'s binary logic, recovering the RC4 encryption key, and reconstructing the custom Canonical Huffman decompression algorithm, the player can parse the network capture, extract the exfiltrated JPEG images, and discover the hidden flag inscribed on a pirate parchment in one of the images.

Challenge Files:

- **Binary:** `catthief.exe` (SHA256:
`904e67200ac373a8929e04e799a805d57f0032325e6cf0bf808c69721cb94c22`)
- **Capture:** `capture.pcapng` (SHA256:
`bdc90661caa86c1d9ccff8d04b1cef5ac656237813e17091687ebb0fa34decec`)

Initial Inspection & Triage

Network Traffic Analysis (`capture.pcapng`)

Opening `capture.pcapng` in Wireshark reveals 2,810 total packets spanning six distinct TCP connections between client `192.168.56.103` (ephemeral source ports `49680–49686`) and server `192.168.56.102:8080`.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.103	192.168.56.102	TCP	66	49680 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
4	0.001189200	192.168.56.102	192.168.56.103	TCP	66	8080 → 49680 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
5	0.001264900	192.168.56.103	192.168.56.102	TCP	54	49680 → 8080 [ACK] Seq=1 Ack=1 Win=65280 Len=0
6	0.001410300	192.168.56.103	192.168.56.102	TCP	187	49680 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=65280 Len=133 [TCP PDU reassembled in 7]
7	0.001427500	192.168.56.103	192.168.56.102	HTTP	318	POST /exfil HTTP/1.1
8	0.001630300	192.168.56.102	192.168.56.103	TCP	60	8080 → 49680 [ACK] Seq=1 Ack=134 Win=64128 Len=0
9	0.001630300	192.168.56.102	192.168.56.103	TCP	60	8080 → 49680 [ACK] Seq=1 Ack=398 Win=64128 Len=0
10	0.003634700	192.168.56.102	192.168.56.103	TCP	146	8080 → 49680 [PSH, ACK] Seq=1 Ack=398 Win=64128 Len=92 [TCP PDU reassembled in 11]
11	0.003998300	192.168.56.102	192.168.56.103	HTTP	90	HTTP/1.0 200 OK

Figure 1: Wireshark packet capture showing TCP connections between 192.168.56.103 and 192.168.56.102:8080

Filtering on `http` displays a repeating communication pattern:

- The client issues an `HTTP POST /exfil` request with high-entropy binary payload data in the request body.
- The server responds with `HTTP/1.0 200 OK` containing a short binary payload (ranging from 0 to 40 bytes).
- The size of the `POST` body starts small (~300 bytes) in the first session, and then scales to several megabytes in subsequent connections (e.g., 3.1 MB, 2.7 MB, 3.0 MB, 3.0 MB, 2.8 MB), indicative of file exfiltration.

```

Wireshark · Follow HTTP Stream (tcp.stream eq 0) · capture.pcapng

POST /exfil HTTP/1.1
Host: 192.168.56.102:8080
User-Agent: ureq/2.12.1
Accept: */*
accept-encoding: gzip
Content-Length: 264

[Binary Payload]

HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.8.10
Date: Thu, 09 Jul 2026 01:36:30 GMT

[Binary Payload]
  
```

Figure 2: HTTP POST /exfil request and 200 OK response stream in Wireshark

Binary Inspection (catthief.exe)

Inspecting `catthief.exe` confirms it is a 64-bit Windows PE executable (x86-64) compiled in Rust:

- **Subsystem:** Windows GUI / CUI
- **Runtime:** Rust Standard Library (`rustc`)
- **Embedded Strings:**
 - `http://192.168.56.102:8080/exfil` at virtual address `0x1400C9135` (in `.rdata`).
 - `READY` at `0x1400C9130`.
 - `ureq` HTTP client crate metadata and standard Rust panic handling structures.

Reverse Engineering the Binary

Main Execution Flow

By cross-referencing the C2 URL string `http://192.168.56.102:8080/exfil` at `0x1400C9135`, we locate the binary's primary worker loop in the function at `0x1400029AA`. This is the challenge's `main` function.

The control flow follows a loop bounded by a counter (`requests_accepted < 5`):

1. **Initialize `next_payload`:** On the first loop iteration, the payload buffer is initialized with a "READY" string.
2. **Compress Payload:** Passes `next_payload` to the compression engine.
3. **Pack Bitstream:** Prepends bit-length headers to the serialized tree and compressed body.
4. **Encrypt:** Applies an in-place stream cipher to the packed buffer.
5. **Exfiltrate:** Issues `HTTP POST http://192.168.56.102:8080/exfil` with the ciphertext payload.
6. **Decrypt C2 Command:** Reads the HTTP response body, decrypts it with the cipher, and parses the UTF-8 command string. C2 commands are not compressed, only encrypted.
7. **Load Next File:** Uses the decrypted string as a filesystem path, setting its raw file bytes as `next_payload` for the subsequent iteration.
8. **Sleep & Repeat:** Sleeps for 30 seconds between iterations, returning to Step 2 after sleep.

Initialize `next_payload` (`0x140002A0E`)

The first payload is a "READY" string to tell the C2 that the tool is ready. Requested payloads are stored within the same variable during the C2 communication loop.

```

310  v305 = -2;
311  alloc_vec_from_slice(&v289, (__int64)aReady);
312  v280 = v253;
313  v287 = &v256[266];
314  requests_accepted = 0;
315  while ( 1 )
316  {

```

Figure 3: Initializing `next_payload` with the "READY" check-in string at `0x140002A0E`

Canonical Huffman Coding Compression (`0x140002A65–0x140003800`)

Recognizing the Frequency Array (`0x140002A65–0x140002BB7`):

The compression routine begins by building a frequency table of all bytes in the payload starting at `0x140002AA7`. The code iterates over the payload buffer, reads a single byte, and inserts it into a `HashMap`. The Rust compiler uses a `SwissTable` implementation for this `HashMap`. This can be identified by the `SipHash` function and by `_mm_loadu_si128` (loading 16 control bytes from the hash bucket) followed closely by `_mm_cmpeq_epi8` and `_mm_movemask_epi8`. This specific trio of instructions is the signature of

SwissTable quickly scanning a 16-byte bucket metadata block to find a collision or an empty slot for the payload byte.

```

327 while ( v1 != v4 )
328 {
329     LOBYTE(v0) = *v1++;
330     v5 = SipHash(*(_QWORD *)&a3[32], *(_QWORD *)&a3[40], (unsigned __int8)v0);
331     v6 = v5 >> 0x39;
332     v7 = *(_QWORD *)&a3;
333     v8 = *(_QWORD *)&a3[8];
334     v9 = _mm_cvtsi32_si128(v5 >> 0x39);
335     v10 = _mm_shuffle_epi32(_mm_shufflelo_epi16(_mm_unpacklo_epi8(v9, v9), 0), 68);
336     v11 = 0;
337     v12 = v5;
338     while ( 1 )
339     {
340         v13 = *(_QWORD *)&a3[8] & v12;
341         v14 = _mm_loadu_si128((const __m128i *)(&a3 + v13));
342         for ( i = _mm_movemask_epi8(_mm_cmpeq_epi8(v14, v10)); (_WORD)i; i &= i - 1 )

```

Figure 4: **SwissTable** frequency counting loop using SipHash and SSE2 instructions

The frequencies **HashMap** can easily be traced at the end of this **SwissTable** implementation once recognized.

```

372     LODWORD(v21) = sub_1400012BA(v7, v8, v19);
373     v22 = v7->m128i_i8[v21];
374     v7->m128i_i8[v21] = v6;
375     v7[1].m128i_i8[v8 & (v21 - 16)] = v6;
376     m128i_i64 = (__int64)v7[-v21].m128i_i64;
377     *(_QWORD *)&freqs_hashmap[16] = v20 - (v22 & 1);
378     *(_BYTE *)(&m128i_i64 - 16) = v4;
379     *(_QWORD *)(&m128i_i64 - 8) = 0;
380     ++*(_QWORD *)&freqs_hashmap[24];
381 LABEL_15:
382     ++*(_QWORD *)(&m128i_i64 - 8);
383 }
384 namelen[0].m128i_i64[0] = 0;
385 namelen[0].m128i_i64[1] = 8;
386 namelen[1].m128i_i64[0] = 0;
387 v23 = *(const __m128i **)freqs_hashmap;
388 v24 = *(_QWORD *)&freqs_hashmap[24];
389 si128 = _mm_load_si128(*(const __m128i **)freqs_hashmap);

```

Figure 5: Frequency **HashMap** output after scanning payload bytes

Recognizing the Tree Builder (0x140002C6F–0x140002D50):

Once the frequencies are calculated, the algorithm builds the Huffman tree. A standard giveaway for a Huffman algorithm is the presence of a Minimum Priority Queue (implemented via a **BinaryHeap**). This heap is initially populated in the loop at 0x140002C6F.

```

404     for ( j = ~_mm_movemask_epi8(si128); ; j = v31 )
405     {
406         v28 = items-- == 0;
407         if ( v28 )
408             break;
409         while ( !j )
410         {
411             v29 = _mm_load_si128(v26);
412             ctrl_bytes_ptr -= 256;
413             ++v26;
414             j = ~_mm_movemask_epi8(v29);
415         }
416         _BitScanForward(&v30, j);
417         v31 = j & (j - 1);
418         v32 = 16 * v30;
419         symbol_byte = ctrl_bytes_ptr[-v32 - 16];
420         frequency_count = *&ctrl_bytes_ptr[-v32 - 8];
421         leaf_node = Box::new();
422         *leaf_node = 0;
423         leaf_node[8] = symbol_byte;
424         BinaryHeap::push(heap, leaf_node, frequency_count);
425     }

```

Figure 6: Populating the BinaryHeap minimum priority queue at 0x140002C6F

In the loop at 0x140002CCF, we see the binary repeatedly popping the two lowest-frequency elements from the heap (call 0x14000275A), allocating a new parent node (call 0x1400012FE) where the frequency is the sum of the two children, and pushing the new parent node back into the heap (call 0x14000284B). This collapses the heap of disconnected leaf nodes into the Huffman tree.

```

428 while ( 1 )
429 {
430     heap_len = heap[1].m128i_u64[0];
431     left_node = BinaryHeap::pop(heap);
432     v39 = left_node;
433     if ( heap_len <= 1 )
434         break;
435     if ( !left_node )
436     {
437         v243 = &off_1400C9170;
438         goto LABEL_348;
439     }
440     v40 = v38;
441     right_node = BinaryHeap::pop(heap);
442     if ( !right_node )
443     {
444         v243 = &off_1400C9188;
445 LABEL_348:
446         core::option::unwrap_failed(v243);
447     }
448     v43 = right_node;
449     combined_freqs = v40 + v42;
450     new_node = Box::new();
451     *new_node = v39;
452     new_node[1] = v43;
453     BinaryHeap::push(heap, new_node, combined_freqs);
454 }

```

Figure 7: Collapsing leaf nodes into the Huffman tree at 0x140002CCF

Recognizing Canonical Code Assignment (0x140002ECB–0x140002EF6):

A conventional Huffman implementation would serialize this entire tree structure and transmit it. However, the challenge utilizes Canonical Huffman Coding. A canonical code is uniquely determined solely by the bit-length of each symbol, meaning the binary only has to transmit a 256-byte array of code lengths instead of a complex tree. You can recognize that the binary is generating Canonical codes by observing the bitwise logic used to assign the prefix codes at 0x140002EE4.

```

532     if ( last_length )
533     {
534         code = (code + 1) << (length - last_length);
535         v263[0].m128i_i32[0] = code;
536     }

```

Figure 8: Canonical Huffman prefix code assignment at 0x140002EE4

This assembly block maps directly to the textbook Canonical Huffman generation algorithm: `code = (code + 1) << (length - last_len)`. The generated code is then formatted into a binary string and stored in a dictionary mapped to the original byte symbol.

Bitstream Packing (0x140003310–0x140003480)

Rather than using a standard bit-shifting accumulator to compress the data, the challenge utilizes an unorthodox approach: it builds the bitstream entirely out of ASCII text before parsing it back down into raw bytes. The challenge iterates over the payload, replacing each byte with its newly assigned Canonical Huffman code formatted as literal '0' and '1' ASCII characters (e.g., the string "0100110").

The ASCII Bit Writer (0x1400053FE):

To construct the header frame, the binary calls 0x1400053FE. When decompiled, this function reveals an internal `_bittest64` loop that extracts bits from an integer, XORs them with 0x31 ('1'), and appends them as literal text characters to the intermediate string buffer. At 0x1400033B8, it writes the 11-bit Tree Length, then the 32-bit Compressed Data Length:

```

706     serialized_tree_len_dup = serialized_tree_len;
707     write_bits(&BitWriter_self, serialized_tree_len, 11u);
708     write_bits(&BitWriter_self, compressed_data_len, 32u);
709     write_bytes(&BitWriter_self, a2, serialized_tree_len_dup);
710     serialized_tree_len = (__int64)compressed_data;
711     write_bytes(&BitWriter_self, compressed_data, compressed_data_len);

```

Figure 9: Writing the 11-bit Tree Length and 32-bit Compressed Data Length headers at 0x1400033B8

Following the headers, the binary appends the serialized Huffman tree and the compressed payload string (via 0x140005457).

The Bit Packer (0x140001329):

Once the entire payload has been constructed as an ASCII string of 1s and 0s, it is packed back down into raw bytes. Starting at 0x140003310, the challenge enters a loop that consumes the string 8 characters at a time, checking if each is '1' (`cmp eax, 0x31`), packing the booleans into a single 8-bit byte, and pushing the compressed byte into the final `Vec<u8>` buffer for encryption.

Decompression Odd Bit Alignment:

Because the bitstream is written continuously, the 11-bit Tree Length throws off the byte alignment of the entire structure. The combined header length is 43 bits (11 + 32). This means the serialized Huffman tree array does not begin on a clean byte boundary; it begins offset by 3 bits into the 6th byte of the payload. Furthermore, the compressed data itself is misaligned based on the variable bit-lengths of the tree. The structure must be parsed continuously as a bitstream, reading MSB-first.

RC4 Encryption (0x140003AF0–0x140003B40)

An RC4 encryption implementation is called at 0x140003AF0–0x140003B40 to encrypt the compressed `next_payload`.

```

739 rc4_key_base = encrypted_rc4_key;
740 rc4_key_2 = 0x407576C41004A6CLL;
741 rc4_key_3 = 7;
742 for ( ii = 0; ii != 25; ++ii )
743     *((_BYTE *)&rc4_key_base + ii) ^= 0x33u;
744 RC4_KSA(&state, (__int64)&rc4_key_base);
745 RC4_PRGA((__int64)&state, (__int64)data, (__int64)v117);

```

Figure 10: RC4 stream cipher initialization and encryption routine at 0x140003AF0

Key Deobfuscation:

The full 25-byte key is split between a 16-byte buffer in `.rdata` and two inlined stack strings. The bytes are XOR-encoded with the single-byte constant 0x33 (`Key[i] = ObfuscatedKey[i] ^ 0x33`). Applying this single-byte XOR deobfuscation reconstructs the plaintext key `i11_b3_p1und3r1n_y3r_d474`:

Hex Byte	Key XOR (0x33)	Plaintext Character
0x5A	0x69	'i'
0x02	0x31	'1'
0x02	0x31	'1'
0x6C	0x5F	'_'
0x51	0x62	'b'
0x00	0x33	'3'
0x6C	0x5F	'_'
0x43	0x70	'p'
0x02	0x31	'1'
0x46	0x75	'u'
0x5D	0x6E	'n'
0x57	0x64	'd'
0x00	0x33	'3'
0x41	0x72	'r'
0x02	0x31	'1'
0x5D	0x6E	'n'

0x6C	0x5F	'_'
0x4A	0x79	'y'
0x00	0x33	'3'
0x41	0x72	'r'
0x6C	0x5F	'_'
0x57	0x64	'd'
0x07	0x34	'4'
0x04	0x37	'7'
0x07	0x34	'4'

Decrypting C2 Responses:

The responses from the C2 are not compressed, only encrypted. Using the RC4 key "i11_b3_p1und3r1n_y3r_d474" on the HTTP responses in `capture.pcapng` yields the five file paths requested by the C2 operator:

- C:\Users\flare\Desktop\important.jpg
- C:\Users\flare\Desktop\anothercat.jpg
- C:\Users\flare\Desktop\yetanothercat.jpg
- C:\Users\flare\Desktop\catcatcat.jpg
- C:\Users\flare\Desktop\myfavoritecat.jpg

Exfiltrate, Decrypt C2 Command, Load next_payload & Sleep

At 0x140003B6A, calls are made into the `ureq` library to load the HTTP method and the C2 URL (`http://192.168.56.102:8080/exfil`). At 0x140003E1E, the encrypted payload slice is passed to `ureq::request::Request::send_bytes` (via 0x140085022), exfiltrating the requested file.

The C2 server responds with an `HTTP 200 OK` whose body contains the next file path encrypted with the same RC4 key ("i11_b3_p1und3r1n_y3r_d474"). The binary invokes `std::fs::read` (recognizable via unobfuscated `CreateFile` and `NtReadFile` calls) on the decrypted path, loads the raw file bytes into `next_payload`, and calls `std::thread::sleep` for 30 seconds before repeating the loop until 5 files are exfiltrated.

Deconstructing Custom Serialization & Huffman Compression

Once the RC4 decryption layer is removed from each HTTP `POST` payload, the resulting plaintext represents a continuous MSB-first bitstream structured as follows:

Tree Length (11)	Data Length (32)	Serialized Tree (N)	Compressed Bitstream
------------------	------------------	---------------------	----------------------

11 bits	32 bits	<code>tree_len</code> Bytes	<code>data_len</code> Bytes
---------	---------	-----------------------------	-----------------------------

1. Header Layout

The bitstream is consumed in Most-Significant-Bit (MSB) first order:

- `tree_len` (11 bits): Read as an 11-bit unsigned integer. In all captures, `tree_len = 256 (0x100)`, representing the code lengths for all 256 possible byte values (0–255).
- `data_len` (32 bits): Read as a 32-bit unsigned integer specifying the exact byte length of the compressed body.
- `tree_data` (`tree_len` bytes): The raw code lengths $L_0, L_1, \dots, L_{255}$ for each symbol 0–255. A length of 0 denotes that the symbol is not present in the payload.
- `compressed_data` (`data_len` bytes): The packed Huffman bitstream.

2. Canonical Huffman Code Generation

The canonical prefix table is reconstructed as follows:

1. Collect all pairs (`symbol`, `length`) where `length > 0`.
2. Sort the pairs primarily by `length` ascending, and secondarily by `symbol` ascending: `SortKey(s, l) = (l, s)`.
3. Assign prefix codes iteratively: initialize `code = 0`, `last_len = 0`; for each (`s`, `l`), if `last_len > 0` set `code = (code + 1) << (l - last_len)`, format `code` as an `l`-bit binary string `code_str`, set `DecodingMap[code_str] = s`, and update `last_len = l`.

3. Bitstream Decompression

With `DecodingMap` constructed, iterate over the bits of `compressed_data` from MSB to LSB, accumulating bits into `current_prefix`. Whenever `current_prefix` matches an entry in `DecodingMap`, append the corresponding symbol byte to the output buffer and reset `current_prefix = ""`.

Solution

We can write a standalone Python script (`solve.py`) to automate PCAPNG extraction, TCP stream reassembly, RC4 decryption, and Canonical Huffman decompression.

Solution Script (`solve.py`)

```
#!/usr/bin/env python3
"""
FLARE-On 13 CatThief Solver & PCAP Payload Extractor
Reassembles TCP flows, decrypts RC4 traffic, and decompresses Canonical Huffman payloads.
"""
```

```

import struct
import os
import sys

# Precomputed 8-bit lookup table for bitstream conversion
LUT = [f"{b:08b}" for b in range(256)]

def generate_canonical_codes(lengths):
    """Rebuilds the canonical Huffman decoding map from an array of symbol bit-lengths."""
    sorted_symbols = [(sym, length) for sym, length in enumerate(lengths) if length > 0]
    sorted_symbols.sort(key=lambda x: (x[1], x[0]))

    decoding_map = {}
    code = 0
    last_len = 0
    for sym, length in sorted_symbols:
        if last_len > 0:
            code = (code + 1) << (length - last_len)
            code_str = bin(code)[2:].zfill(length)
            decoding_map[code_str] = sym
            last_len = length

    return decoding_map

class BitReader:
    """Consumes arbitrary bitfields and byte streams from packed binary data."""
    def __init__(self, data):
        self.bits = "".join(LUT[b] for b in data)
        self.pos = 0

    def read_bits(self, num_bits):
        if self.pos + num_bits > len(self.bits):
            return None
        val_str = self.bits[self.pos:self.pos+num_bits]
        self.pos += num_bits
        return int(val_str, 2)

    def read_bytes(self, num_bytes):
        out = []
        for _ in range(num_bytes):
            val = self.read_bits(8)
            if val is None:
                break
            out.append(val)
        return bytes(out)

def huffman_decompress(compressed_data, decoding_map):
    """Decompresses a bitstream using the canonical code map."""
    bit_string = "".join(LUT[b] for b in compressed_data)
    current = ""
    out = bytearray()
    for bit in bit_string:
        current += bit
        if current in decoding_map:
            out.append(decoding_map[current])
            current = ""
    return bytes(out)

def rc4_crypt(key, data):
    """Standard RC4 stream cipher encryption/decryption."""
    s = list(range(256))

```

```

j = 0
for i in range(256):
    j = (j + s[i] + key[i % len(key)]) % 256
    s[i], s[j] = s[j], s[i]

i = 0
j = 0
out = []
for byte in data:
    i = (i + 1) % 256
    j = (j + s[i]) % 256
    s[i], s[j] = s[j], s[i]
    t = (s[i] + s[j]) % 256
    out.append(byte ^ s[t])

return bytes(out)

def parse_pcapng(pcap_data):
    """Parses PCAPNG blocks and extracts raw Ethernet/IP packets with timestamps."""
    offset = 0
    packets = []
    endian = '<'
    if_ts_resolutions = {}
    if_index = 0

    while offset < len(pcap_data):
        if offset + 8 > len(pcap_data):
            break
        block_type, block_len = struct.unpack(endian + "II", pcap_data[offset:offset+8])
        if block_type == 0x0A0D0D0A: # Section Header Block
            endian = '>' if pcap_data[offset+8:offset+12] == b'\x1a\x2b\x3c\x4d' else '<'
            block_type, block_len = struct.unpack(endian + "II", pcap_data[offset:offset+8])

        if block_len < 12:
            break

        block_body = pcap_data[offset+8:offset+block_len-4]
        if block_type == 0x00000001: # Interface Description Block
            ts_resol = 6
            opt_off = 16
            while opt_off < len(block_body):
                if opt_off + 4 > len(block_body):
                    break
                opt_code, opt_len = struct.unpack(endian + "HH", block_body[opt_off:opt_off+4])
                if opt_code == 0:
                    break
                if opt_code == 9:
                    ts_resol = block_body[opt_off+4]
                    opt_off += 4 + ((opt_len + 3) & ~3)
            if_ts_resolutions[if_index] = ts_resol
            if_index += 1
        elif block_type == 0x00000006: # Enhanced Packet Block
            if len(block_body) >= 20:
                if_id, ts_high, ts_low, cap_len, orig_len = struct.unpack(endian + "IIIII",
block_body[:20])
                ts_resol = if_ts_resolutions.get(if_id, 6)
                ts = ((ts_high << 32) | ts_low) * (10 ** -ts_resol)
                packets.append((ts, block_body[20:20+cap_len]))
            offset += block_len
    return packets

def extract_tcp_streams(packets):

```

```

"""Reassembles TCP payload streams targeting C2 port 8080."""
streams = {}
for ts, pkt in packets:
    if len(pkt) < 34 or struct.unpack(">H", pkt[12:14])[0] != 0x0800:
        continue
    ip_hdr = pkt[14:]
    ihl = (ip_hdr[0] & 0x0f) * 4
    if ip_hdr[9] != 6:
        continue # Must be TCP

    src_ip = ".".join(map(str, ip_hdr[12:16]))
    dst_ip = ".".join(map(str, ip_hdr[16:20]))
    tcp_hdr = ip_hdr[ihl:]
    src_port, dst_port, seq = struct.unpack(">HHI", tcp_hdr[:8])
    data_offset = (struct.unpack(">H", tcp_hdr[12:14])[0] >> 12) * 4
    payload = tcp_hdr[data_offset:]
    if not payload:
        continue

    flow_key = (src_ip, src_port, dst_ip, dst_port)
    streams.setdefault(flow_key, []).append((ts, seq, payload))
return streams

def reassemble(pkts):
    pkts.sort(key=lambda x: x[1])
    assembled = bytearray()
    last_seq = pkts[0][1]
    for ts, seq, payload in pkts:
        if seq >= last_seq:
            assembled.extend(payload)
            last_seq = seq + len(payload)
    return bytes(assembled)

def main():
    pcap_path = "capture.pcapng"
    out_dir = "extracted_files"
    key = b"i11_b3_p1und3r1n_y3r_d474"

    os.makedirs(out_dir, exist_ok=True)
    with open(pcap_path, 'rb') as f:
        packets = parse_pcapng(f.read())

    streams = extract_tcp_streams(packets)
    c2_flows = {k: v for k, v in streams.items() if k[3] == 8080}

    print(f"[*] Reassembling {len(c2_flows)} exfiltration sessions...")

    last_cmd = None
    for idx, (flow_key, pkts) in enumerate(sorted(c2_flows.items(), key=lambda x: x[1][0][0])):
        data = reassemble(pkts)
        body_idx = data.find(b"\r\n\r\n")
        if body_idx == -1:
            continue

        # 1. Decrypt RC4
        ciphertext = data[body_idx+4:]
        decrypted = rc4_crypt(key, ciphertext)

        # 2. Parse Bitstream Header
        reader = BitReader(decrypted)
        tree_len = reader.read_bits(11)
        data_len = reader.read_bits(32)

```

```

tree_data = reader.read_bytes(tree_len)
compressed_data = reader.read_bytes(data_len)

# 3. Decompress Huffman
decoding_map = generate_canonical_codes(tree_data)
decompressed = huffman_decompress(compressed_data, decoding_map)

# 4. Save Extracted Artifact
if idx == 0:
    filename = "000_checkin.txt"
else:
    filename = f"{idx:03d}_" + (os.path.basename(last_cmd) if last_cmd else
f"file_{idx}.bin")

    with open(os.path.join(out_dir, filename), 'wb') as out_f:
        out_f.write(decompressed)

print(f"[+] Session {idx}: Extracted {len(decompressed)} bytes -> {filename}")

# Fetch corresponding C2 response to identify next requested filename
resp_key = (flow_key[2], flow_key[3], flow_key[0], flow_key[1])
if resp_key in streams:
    resp_data = reassemble(streams[resp_key])
    r_body = resp_data.find(b"\r\n\r\n")
    if r_body != -1:
        dec_cmd = rc4_crypt(key, resp_data[r_body+4:])
        last_cmd = dec_cmd.decode('utf-8', errors='ignore').replace('\\', '/')

if __name__ == '__main__':
    main()

```

Artifact Analysis & Flag Recovery

Running the automated solver recovers six files from the PCAP capture into `extracted_files/`:

Filename	Type	Size	Description
000_checkin.txt	ASCII Text	7 bytes	Initial check-in handshake payload: <code>READYDD</code>
001_important.jpg	JPEG Image	3,123,324 bytes	Calico ship cat sitting by a gunpowder barrel
002_anothercat.jpg	JPEG Image	2,738,892 bytes	Siamese pirate cat wearing a red bandana in the crow's nest
003_yetanothercat.jpg	JPEG Image	3,065,529 bytes	Black pirate cat wearing an eyepatch by ship cannons
004_catcatcat.jpg	JPEG Image	3,084,399 bytes	Fluffy pirate cat on deck beside a treasure chest with a flag parchment
005_myfavoritecat.jpg	JPEG Image	2,844,616 bytes	Orange tabby cat resting inside a treasure chest filled with gold doubloons

Flag Inspection



Figure 11: Exfiltrated image 004_catcatcat.jpg revealing the flag on the pirate parchment

Inspecting `004_catcatcat.jpg`, we observe a parchment scroll placed directly on top of the wooden chest on the ship's quarterdeck. The flag is clearly legible on the scroll:

th3r3_b3_tr34sure_1n_th4t_pc4p@flare-on.com.