

FLARE-On 13 Challenge 6: ThreatInvaders

By Vince Weyell (@weyell)

Overview

This challenge is designed around the C# JIT pipeline in .NET 10.0 (`ThreatInvaders.exe`) paired with a network packet capture (`challenge.pcapng`), highlighting native runtime hooks inside `singlefilehost.exe`. While the managed C# assembly (`ThreatInvaders.dll`) appears to be a clean WinForms Space Invaders game with online highscores and telemetry, native detours in CoreCLR (`MethodDesc::DoPrestub` and `MethodDesc::SetNativeCode`) secretly intercept highscore synchronization, telemetry reporting, and the auto-updater to implement an encrypted Command and Control (C2) channel. Reversing the native CoreCLR hooks and decrypting the PCAP traffic reveals the final flag: **wh4t_4_l0ng_str4ng3_tr1p_1ts_b33n@flare-on.com**.

Note that the Space Invaders game itself was created with an LLM and adjusted for the challenge, while all modifications to the .NET JIT were crafted by hand so that the backdoor code would not appear in any training set.

Step 1: Initial Triage & Extracting the .NET Single-File Bundle

Extracting `ThreatInvaders.zip` (with password `flare`) yields two files: a ~110 MB 64-bit Windows executable (`ThreatInvaders.exe`) and a network packet capture (`challenge.pcapng`).

Inspecting the PE headers and strings of `ThreatInvaders.exe` reveals that it is a .NET 10.0 Single-File Host (`singlefilehost.exe`) with the PDB path:

```
E:\runtime\artifacts\obj\coreclr\windows.x64.Release\Corehost.Static\singlefilehost.pdb
```

This PDB path and the PE compilation timestamp (2026-09-15 15:04:00 UTC) indicate a custom-compiled `singlefilehost.exe` built from the `dotnet/runtime` source tree.

First, let's extract the managed assemblies bundled inside the overlay of `ThreatInvaders.exe`. While tools like `SingleFileExtractor` can do this out of the box, we can also extract the bundle directly: a .NET single-file bundle stores the 32-byte SHA-256 signature of "32 bytes represent the bundle header...":

```
8b 12 02 b9 6a 61 20 38 72 7b 93 02 14 d7 a0 32
```

```
13 f5 b9 e6 ef ae 33 18 ee 3b 2d ce 24 b3 6a ae
```

Immediately preceding this signature (at file offset `0x7F8D70`) is an 8-byte little-endian integer pointing to the bundle manifest header at file offset `0x6ED6ECD`. We can extract all bundled files (`ThreatInvaders.dll`, `ThreatInvaders.deps.json`, `ThreatInvaders.runtimeconfig.json`, and framework libraries) using the following Python script:

```
#!/usr/bin/env python3
import struct
import os

BUNDLE_SIG = bytes.fromhex(
    "8b1202b96a612038727b930214d7a032"
    "13f5b9e6efae3318ee3b2dce24b36aae"
)

def read_7bit_int(data: bytes, pos: int):
    res = 0
    shift = 0
    while True:
        b = data[pos]
        pos += 1
        res |= (b & 0x7F) << shift
        if (b & 0x80) == 0:
            break
        shift += 7
    return res, pos

def extract_bundle(exe_path: str, out_dir: str):
    os.makedirs(out_dir, exist_ok=True)
    with open(exe_path, "rb") as f:
        data = f.read()

    sig_off = data.find(BUNDLE_SIG)
    header_off = struct.unpack_from("<Q", data, sig_off - 8)[0]
    print(f"[+] Bundle signature at 0x{sig_off:x}, header offset = 0x{header_off:x}")

    pos = header_off
    major, minor, num_files = struct.unpack_from("<III", data, pos)
    pos += 12
    bundle_id_len, pos = read_7bit_int(data, pos)
    bundle_id = data[pos : pos + bundle_id_len].decode("utf-8")
    pos += bundle_id_len
    print(f"[+] Bundle v{major}.{minor}, files={num_files}, id={bundle_id}")

    # Skip deps.json, runtimeconfig.json, and flags in v6.0+ header
    pos += 16 + 16 + 8

    for _ in range(num_files):
        file_off, file_size, compressed_size, file_type = struct.unpack_from("<QQQB", data, pos)
        pos += 25
        path_len, pos = read_7bit_int(data, pos)
        rel_path = data[pos : pos + path_len].decode("utf-8")
```

```

pos += path_len
if "ThreatInvaders" in rel_path:
    out_path = os.path.join(out_dir, os.path.basename(rel_path))
    with open(out_path, "wb") as out:
        out.write(data[file_off : file_off + file_size])
    print(f"[+] Extracted {rel_path} ({file_size} bytes) -> {out_path}")

if __name__ == "__main__":
    extract_bundle("ThreatInvaders.exe", "bundle_extracted")

```

Step 2: Decompiling ThreatInvaders.dll & Spotting Anomalies

Decompiling `ThreatInvaders.dll` with ILSpy reveals a WinForms Space Invaders clone with online highscores, an auto-updater check, and telemetry reporting. Reviewing the C# classes shows no overt malicious logic, but several design choices stand out:

- **A Decoy Flag in `UpdateLauncher.cs`:**

```

if
(!TryFillArgBuffer("LS1pbnN0YWxsLWRpciBXb3VsZG4ndF9pdF9iZV9uaWNlX2lmX3RoXNfd2FzX3RoZV9yZWFsX2ZsYWdAbm8tZmxhcmUuY29tIC0tY3dkIEJ1dF9zYWRseV9pdF9pc24ndA", array2))

```

Base64-decoding that string gives:

```
--install-dir Wouldn't_it_be_nice_if_this_was_the_real_flag@no-flare.com --cwd But_sadly_it_isn't
```

Furthermore, `LaunchUpdate(byte[] pathBuffer, string expectedBaseDir, byte[]? argument)` takes a 260-byte `byte[] pathBuffer` and a 1024-byte `byte[] argument`, converts both from ASCII strings, runs `Process.Start(FileName = text, Arguments = text2)` with `RedirectStandardOutput = true`, and returns `memoryStream.ToArray()`.

- **Unmanaged Pointers in `HighscoreStruct.cs` / `HighscoreBatch.cs`:** When highscores are downloaded from `/api/v1/highscores`, `HighscoreBatch.Create` packs them into an array of 80-byte value types (`HighscoreStruct[]`), allocating an unmanaged heap buffer (`Marshal.AllocHGlobal`) for each entry's `Comment` (`byte[]`) and storing the raw pointer `nint Comment` at struct offset `+64` and `int CommentLength` at struct offset `+72`. Then `HighscoreService` calls `_remoteCache.ReplaceHighscores(highscoreBatch.Entries)`.

- Suspicious Signature in `TelemetryReporter.cs`:

```
private static string EncodeFeedbackBuffer(byte[] fixedBuffer, ref int length)
{
    if (fixedBuffer.Length != 4096)
        throw new ArgumentException(...);
    if (length > 0)
        return Convert.ToBase64String(fixedBuffer);
    return "";
}
```

Why does `EncodeFeedbackBuffer` take `ref int length` when it never modifies `length`, and why does `_pendingFeedback` start initialized to `new byte[4096]` (`length = 4096`) when no user feedback has been entered? Passing `length` by reference allows a native hook to mutate the reported feedback length sent to the C2 server.

Step 3: Inspecting `challenge.pcapng`

Next, we inspect the HTTP traffic in `challenge.pcapng` between the game client (`192.168.198.129`) and the server (`192.168.198.130:80, Host: threatinvaders.domain`). Every ~5 seconds, `HighscoreService.BackgroundLoopAsync()` issues:

- GET `/api/v1/health`
- GET `/api/v1/highscores?limit=25`
- GET `/api/v1/version`
- POST `/api/v1/telemetry`

Right at the start, an empty feedback buffer is sent because `_hasFeedback` is initialized to `true` to force JIT compilation of the telemetry methods so their native pointers can be hooked. Additionally, the server constantly replies with version `1.0.0.1` via `/api/v1/version`, which triggers `GameForm.MaybePromptForUpdate()` → `UpdateLauncher.LaunchConsentedUpdater()` → `UpdateLauncher.LaunchUpdate()`.

As the packet capture progresses, three new highscore entries appear on `/api/v1/highscores`, each followed by a special telemetry event:

- 1. The "BOOM" Highscore & Handshake:
 - Server adds a highscore entry: `player_name: "BOOM", score: 31337, wave_reached: 1, comment_b64: "ZnVuIQ==" ("fun!")`.
 - On the next telemetry poll, the client sends POST `/api/v1/telemetry` with `"feedback_length": 8208 (0x2010)` and `"feedback_b64": "VK4DLqh1sPjMV4pRH+cWMQAA..."` (16 non-zero bytes `54 ae 03 2e a8 75 b0 f8 cc 57 8a 51 1f e7 16 31`, followed by 4,080 zero bytes).

- **2. The "BANANA" Highscore & First Command:**

- Server adds a highscore entry: `player_name: "BANANA", score: 2066361377 (0x7b2a2c21), wave_reached: 69, comment_b64: "+bmPsSZwC3eIRD/rvqXFfnzf"` (18 bytes).
- Immediately after, the client sends `POST /api/v1/telemetry` with `"feedback_length": 16914 (0x4212, i.e., 0x4000 | 0x212, where 0x212 = 530)` and `"feedback_b64"` containing 530 non-zero ciphertext bytes followed by zeros.

- **3. The "BENIGN" Highscore & Second Command:**

- Server adds a highscore entry: `player_name: "BENIGN", score: 1932142625 (0x732a2821), wave_reached: 67, comment_b64: "GenjMZg28ziV/EU2YTRDGmZISVPsEraqrC003g=="` (28 bytes).
- Immediately after, the client sends `POST /api/v1/telemetry` with `"feedback_length": 16430 (0x402e, i.e., 0x4000 | 0x2e, where 0x2e = 46)` and `"feedback_b64"` containing 46 non-zero ciphertext bytes followed by zeros.
- Server responds with `{"accepted":true,"reward_now":true}`.

Since none of this behavior exists in the C# assemblies, the backdoor must reside in the native .NET CoreCLR runtime inside `ThreatInvaders.exe`.

Step 4: Reversing the Native CoreCLR Backdoor in `ThreatInvaders.exe`

Searching the native `.rdata` section of `ThreatInvaders.exe` for method names from `ThreatInvaders.dll` reveals the ASCII string `"ReplaceHighscores"` at VA `0x140755DD0`. Following the cross-reference leads to `0x14016C050`:

```
if ( qword_1407FE5A0 == 0 )
{
    v8 = (const char *)MethodDesc::GetName(v7);
    if ( strcmp(v8, "ReplaceHighscores") == 0 )
        qword_1407FE5A0 = v7; // Save MethodDesc* of ReplaceHighscores
}
```

Checking cross-references to `qword_1407FE5A0` leads to `0x1400C6E30` (`MethodDesc::SetNativeCode`), which is invoked whenever the JIT finishes compiling a method and sets its native code pointer (`a2`):

```
__int64 __fastcall MethodDesc_SetNativeCode(volatile signed __int32 *a1, __int64 a2)
{
    v2 = (__int64 (__fastcall *)())a2;
    if ( qword_1407FE5A0 != 0 && a1 == (volatile signed __int32 *)qword_1407FE5A0 )
    {
        qword_1407FE5D8 = a2; // Original JITted ReplaceHighscores
    }
}
```

```

v2 = hook_ReplaceHighscores; // 0x1400C66F0
}
if ( qword_1407FE598 != 0 && a1 == (volatile signed __int32 *)qword_1407FE598 )
{
    qword_1407FE5D0 = (__int64)v2; // Original JITted LaunchUpdate
    v2 = hook_LaunchUpdate;      // 0x1400C6A70
}
if ( qword_1407FE5C8 != 0 && a1 == (volatile signed __int32 *)qword_1407FE5C8 )
{
    qword_1407FE5E0 = (__int64)v2; // Original JITted HasFeedback
    v2 = hook_HasFeedback;        // 0x1400C6B70
}
if ( qword_1407FE5C0 != 0 && a1 == (volatile signed __int32 *)qword_1407FE5C0 )
{
    qword_1407FE5B0 = (__int64)v2; // Original JITted EncodeFeedbackBuffer
    v2 = hook_EncodeFeedbackBuffer; // 0x1400C6BA0
}
if ( qword_1407FE5B8 != 0 && a1 == (volatile signed __int32 *)qword_1407FE5B8 )
{
    qword_1407FE5A8 = (__int64)v2; // Original JITted GetSessionID
    v2 = hook_GetSessionID;        // 0x1400C6DF0
}
// ...
}

```

The remaining `MethodDesc*` globals (`qword_1407FE598`, `qword_1407FE5C8`, `qword_1407FE5C0`, `qword_1407FE5B8`, and `qword_1407FFA40`) are populated inside `MethodDesc::DoPrestub` (`0x1400F5520`) by hashing each method's name with a custom 64-bit hash function at `0x1400C8090`:

```

def custom_hash(b: bytes) -> int:
    a2 = (0xC53A1B7F2D9E4467 * (b[0] ^ 0x8F3A2C91D7B4E6A5)) & 0xFFFFFFFFFFFFFFFF
    a3 = 0x5CA6E801706EAAAC
    for ch in b[1:]:
        if ch == 0:
            break
        a2 = (a3 * (a2 ^ ch)) & 0xFFFFFFFFFFFFFFFF
        a3 = (0x2545F4914F6CDD1D * a3 + 1) & 0xFFFFFFFFFFFFFFFF
    return a2

```

Hashing all method names in `ThreatInvaders.dll` resolves every target:

Hash Constant	Method Name	Native Detour Function
0x322131F288B022A7	ReplaceHighscores	0x1400C66F0 (hook_ReplaceHighscores)
0x6F8FF9636EAE6C5A	LaunchUpdate	0x1400C6A70 (hook_LaunchUpdate)
0x6397C52D5293345E	EncodeFeedbackBuffer	0x1400C6BA0 (hook_EncodeFeedbackBuffer)

0x5F6E18DB1BE45DC7	HasFeedback	0x1400C6B70 (hook_HasFeedback)
0x3F926793476B8710	GetMachineGuid	0x1400F54E0 (hook_GetMachineGuid)
0xA3A845AEC791499A	GetSessionID	0x1400C6DF0 (hook_GetSessionID)

Step 5: Understanding the C2 Protocol & Cryptography

Let's examine each of the 6 native detour functions in `ThreatInvaders.exe`:

1. `hook_GetMachineGuid (0x1400F54E0)` and `hook_GetSessionID (0x1400C6DF0)`

Both `GetMachineGuid()` and `GetSessionID()` return a 16-byte `System.Guid` value type via a hidden return-buffer pointer (`rdx` for static `GetMachineGuid(Guid* retBuf)`, `r8` for instance `GetSessionID(this, Guid* retBuf)`):

- `hook_GetMachineGuid` calls the original `GetMachineGuid()`, copies the 16-byte `MachineGuid` into `byte_1407FFA30[0..15]`, and sets `byte_1407FD5FF = 1`.
- `hook_GetSessionID` calls the original `GetSessionID()`, copies the 16-byte `SessionId` into `byte_1407FF800[0..15]`, and sets `byte_1407FD61E = 1`.

2. Activation via `hook_ReplaceHighscores (0x1400C66F0)`

When `LocalHighscoreStore.ReplaceHighscores(HighscoreStruct[] entries)` is called (`rcx = this`, `rdx = entries`), the hook loops over each 80-byte `HighscoreStruct` element in the managed array (`entries + 16 + 80 * i`):

- Before the backdoor is activated (`byte_1407FD61D == 0`), it copies the unmanaged `Comment` string into a local stack buffer and hashes its first 4 bytes with `custom_hash`.
- If the hash equals `0x05045A536B62B2C6`—which is `custom_hash(b"fun!")` from the "BOOM" highscore entry—and both `MachineGuid` and `SessionId` have been captured, it sets `byte_1407FD5FE = 1`.

3. Key Exchange / Handshake via `hook_EncodeFeedbackBuffer (0x1400C6BA0)`

When `byte_1407FD5FE == 1`:

- `hook_HasFeedback (0x1400C6B70)` forces `HasFeedback()` to return `true`.
- `hook_EncodeFeedbackBuffer(byte[] fixedBuffer, ref int length)` computes a 16-byte handshake payload `v6` by XORing `MachineGuid` with the byte-reversed `SessionId`:

```
for (int i = 0; i < 16; i++)
```

```
v6[i] = byte_1407FFA30[i] ^ byte_1407FF800[15 - i];
```

- It writes those 16 bytes into `fixedBuffer + 16` (the data start of the managed `byte[4096]` array), sets `length = 8208 (0x2010)`, and sets `byte_1407FD61D = 1` to mark the handshake complete.

4. Command Reception & Stream Cipher (`backdoor_prng_xor_crypt` at `0x1400C65E0`)

Once `byte_1407FD61D == 1`, `hook_ReplaceHighscores` checks every highscore entry for a command trigger:

- It takes the 4-byte `Score` at `entries + 80*i + 40` and XORs it with the first 4 bytes of `PlayerName` at `entries + 80*i + 60`.
- It hashes the 4-byte XOR result using `custom_hash`.
- If the hash equals `0xCE723A0A685E63E4 (custom_hash(b"cmd:"))`, it triggers command decryption! Both `"BANANA"` and `"BENIGN"` satisfy this condition:
 - `"BANA" (0x414e4142) ^ 2066361377 (0x7b2a2c21) = 0x3a646d63 = "cmd:"`
 - `"BENI" (0x494e4542) ^ 1932142625 (0x732a2821) = 0x3a646d63 = "cmd:"`
- When `"cmd:"` matches, it decrypts the entry's `Comment` bytes in-place using `sub_1400C65E0 (backdoor_prng_xor_crypt)`.

On its very first call (`qword_1407FFA20 == 0`), `backdoor_prng_xor_crypt` seeds a 64-bit state `qword_1407FFA20` from the two 64-bit halves (`v7, v8`) of `MachineGuid (byte_1407FFA30)`:

```
v5 = v7 ^ (v8 + (v7 << 6) + (v7 >> 2) - 0x61C8864680B583EBULL);
x = 0xFF51AFD7ED558CCDULL * (v5 ^ (v5 >> 33));
y = 0xC4CEB9FE1A85EC53ULL * (x ^ (x >> 29));
qword_1407FFA20 = y ^ (y >> 32);
```

Afterwards, for each byte in the buffer, it advances `qword_1407FFA20` using a 64-bit LCG (multiplier `0x5851F42D4C957F2D`, increment `0x14057B7EF767814F`) and XORs the byte with the top 8 bits (`>> 56`):

```
for (uint32_t i = 0; i < len; i++) {
    qword_1407FFA20 = qword_1407FFA20 * 0x5851F42D4C957F2DULL + 0x14057B7EF767814FULL;
    buf[i] ^= (uint8_t)(qword_1407FFA20 >> 56);
}
```

Crucially, `qword_1407FFA20` is a continuous global state that is never reset between calls: every command decryption and output encryption advances the same LCG stream in lockstep. After decrypting the `Comment` ("`cmd:<executable>;<arguments>`"), `hook_ReplaceHighscores` skips the 4-byte "`cmd:`" prefix, copies the executable path up to `' ; '` into `byte_1407FF600`, copies the arguments after `' ; '` into `byte_1407FF820`, and sets `byte_1407FD61F = 1`.

5. Command Execution & Exfiltration via `hook_LaunchUpdate` (0x1400C6A70)

When `/api/v1/version` returns `1.0.0.1`, the client triggers `UpdateLauncher.LaunchConsentedUpdater()`, which calls `LaunchUpdate(byte[] pathBuffer, string expectedBaseDir, byte[]? argument)`. `hook_LaunchUpdate` checks whether `byte_1407FD61F == 1`, overwrites `pathBuffer` (`a1 + 16`) with `byte_1407FF600` and `argument` (`a3 + 16`) with `byte_1407FF820`, and invokes the original `LaunchUpdate` to execute the command and capture `stdout`.

`hook_LaunchUpdate` then copies the `stdout` bytes into `unk_1407FE5F0`, encrypts them in-place using `backdoor_prng_xor_crypt`, stores the length in `qword_1407FE5E8`, and sets `byte_1407FD61C = 1`. On the next telemetry cycle, `hook_EncodeFeedbackBuffer` copies the encrypted output into `fixedBuffer`, sets `length = stdout_len | 0x4000`, and transmits it to the C2 server.

Step 6: Decrypting the PCAP Traffic & Recovering the Flag

Now we have everything needed to write a standalone solver script that:

1. Extracts `session_id` (`478b635d-c335-4c12-b1de-16e0565be335`) and the 16-byte handshake (`feedback_length == 8208`) from `challenge.pcapng`.
2. Recovers the victim's `MachineGuid` by XORing the 16-byte handshake with the reversed `session_id` bytes: `MachineGuid = 78584d61-6348-496e-8045-4964586c756c` (which spells "`xXMachInEIdlu1X`" in ASCII).
3. Initializes the PRNG state and decrypts the C2 session in exact chronological order:
 - "`BANANA`" `comment_b64` (18 bytes)
 - Telemetry response with `feedback_length = 16914` (`0x4212` → 530 bytes)
 - "`BENIGN`" `comment_b64` (28 bytes)
 - Telemetry response with `feedback_length = 16430` (`0x402e` → 46 bytes)

Complete Solve Script

```
#!/usr/bin/env python3
import base64
import json
import os
import struct
import sys
```

```

import uuid

def parse_pcapng_http_messages(pcap_path: str):
    with open(pcap_path, "rb") as f:
        data = f.read()

    offset = 0
    events = []
    while offset + 8 <= len(data):
        block_type, block_len = struct.unpack_from("<II", data, offset)
        if block_len < 12 or offset + block_len > len(data):
            break
        if block_type == 6: # Enhanced Packet Block (EPB)
            cap_len = struct.unpack_from("<I", data, offset + 20)[0]
            pkt = data[offset + 28 : offset + 28 + cap_len]
            if len(pkt) >= 34 and struct.unpack_from(">H", pkt, 12)[0] == 0x0800:
                ip_hdr = pkt[14:]
                tot_len = struct.unpack_from(">H", ip_hdr, 2)[0]
                if tot_len >= 20:
                    ip_hdr = ip_hdr[:tot_len]
                    ihl = (ip_hdr[0] & 0x0F) * 4
                    proto = ip_hdr[9]
                    if proto == 6 and len(ip_hdr) >= ihl + 20:
                        src_ip = ".".join(str(b) for b in ip_hdr[12:16])
                        dst_ip = ".".join(str(b) for b in ip_hdr[16:20])
                        tcp = ip_hdr[ihl:]
                        sport, dport, seq, ack, doff_flags = struct.unpack_from(">HHIIH", tcp, 0)
                        doff = (doff_flags >> 12) * 4
                        payload = tcp[doff:]
                        if payload:
                            flow = f"{src_ip}:{sport}->{dst_ip}:{dport}"
                            events.append((flow, payload))
            offset += block_len

    # Merge contiguous TCP segments per flow direction
    merged = []
    for flow, payload in events:
        if merged and merged[-1][0] == flow:
            merged[-1][1].extend(payload)
        else:
            merged.append((flow, bytearray(payload)))

    return [bytes(buf).decode("utf-8", errors="replace") for _, buf in merged]

def custom_hash(b: bytes) -> int:
    if not b:
        return 0
    a2 = (0xC53A1B7F2D9E4467 * (b[0] ^ 0x8F3A2C91D7B4E6A5)) & 0xFFFFFFFFFFFFFFFF
    a3 = 0x5CA6E801706EAAAC
    for ch in b[1:]:
        if ch == 0:
            break
        a2 = (a3 * (a2 ^ ch)) & 0xFFFFFFFFFFFFFFFF
        a3 = (0x2545F4914F6CDD1D * a3 + 1) & 0xFFFFFFFFFFFFFFFF
    return a2

class BackdoorPRNG:
    def __init__(self, machine_guid_bytes_le: bytes):
        v7, v8 = struct.unpack("<QQ", machine_guid_bytes_le)
        v5 = (v7 ^ (v8 + (v7 << 6) + (v7 >> 2) - 0x61C8864680B583EB)) & 0xFFFFFFFFFFFFFFFF
        x = (0xFF51AFD7ED558CCD * (v5 ^ (v5 >> 33))) & 0xFFFFFFFFFFFFFFFF

```

```

y = (0xC4CEB9FE1A85EC53 * (x ^ (x >> 29))) & 0xFFFFFFFFFFFFFFFF
self.state = (y ^ (y >> 32)) & 0xFFFFFFFFFFFFFFFF

def crypt(self, buf: bytes) -> bytes:
    out = bytearray(buf)
    for i in range(len(out)):
        self.state = (self.state * 0x5851F42D4C957F2D + 0x14057B7EF767814F) &
0xFFFFFFFFFFFFFFFF
        out[i] ^= (self.state >> 56) & 0xFF
    return bytes(out)

def solve(pcap_path: str):
    http_messages = parse_pcapng_http_messages(pcap_path)

    prng = None
    seen_command_ids = set()

    for msg in http_messages:
        if "\r\n\r\n" not in msg:
            continue
        headers, body = msg.split("\r\n\r\n", 1)
        body = body.strip()
        if not body.startswith("{"):
            continue
        try:
            data = json.loads(body)
        except json.JSONDecodeError:
            continue

        # 1. Check for Telemetry POSTs (handshake 0x2010 or command output 0x4000 | len)
        if "feedback_length" in data and "feedback_b64" in data:
            fb_len = data["feedback_length"]
            if fb_len == 0x2010 and prng is None:
                session_guid = uuid.UUID(data["session_id"])
                session_bytes = session_guid.bytes_le
                handshake = base64.b64decode(data["feedback_b64"])[16:]
                machine_bytes = bytes(handshake[i] ^ session_bytes[15 - i] for i in range(16))
                machine_guid = uuid.UUID(bytes_le=machine_bytes)
                prng = BackdoorPRNG(machine_bytes)
                print(f"[+] Session ID      : {session_guid}")
                print(f"[+] MachineGuid    : {machine_guid} (raw: {machine_bytes})")
                print(f"[+] Initial PRNG   : 0x{prng.state:016x}\n")
            elif (fb_len & 0x4000) != 0 and prng is not None:
                out_len = fb_len & 0x3FFF
                enc_out = base64.b64decode(data["feedback_b64"][:out_len])
                dec_out = prng.crypt(enc_out)
                print(f"[<] Telemetry Output ({out_len} bytes):")
                print(dec_out.decode("utf-8", errors="replace"))

        # 2. Check for Highscore GET responses containing C2 commands
        elif "entries" in data and isinstance(data["entries"], list) and prng is not None:
            for entry in data["entries"]:
                entry_id = entry.get("id")
                player_name = entry.get("player_name", "")
                score = entry.get("score", 0)
                if len(player_name) >= 4 and entry_id not in seen_command_ids:
                    score_bytes = struct.pack("<I", score & 0xFFFFFFFF)
                    name_bytes = player_name[:4].encode("ascii", errors="ignore")
                    xored = bytes(a ^ b for a, b in zip(score_bytes, name_bytes))
                    if custom_hash(xored) == 0xCE723A0A685E63E4: # custom_hash(b"cmd:")
                        seen_command_ids.add(entry_id)
                        enc_cmd = base64.b64decode(entry["comment_b64"])

```

```
dec_cmd = prng.crypt(enc_cmd)
print(f"[>] Highscore Command ({player_name}, score={score}):
{dec_cmd.decode('utf-8', errors='replace')}")

if __name__ == "__main__":
    pcap_file = sys.argv[1] if len(sys.argv) > 1 else "challenge.pcapng"
    solve(pcap_file)
```

Solver Output

Running the script produces the complete decrypted C2 transcript and the flag:

```
[+] Session ID      : 478b635d-c335-4c12-b1de-16e0565be335
[+] MachineGuid    : 78584d61-6348-496e-8045-4964586c756c (raw: b'aMXxHcnI\x80EIdXlu1')
[+] Initial PRNG    : 0xc4bfc80d59d86b9b

[>] Highscore Command (BANANA, score=2066361377): cmd:cmd.exe;/c dir
[<] Telemetry Output (530 bytes):
Volume in drive Z is Shared Folders
Volume Serial Number is 0000-0064

Directory of
Z:\CTFs\FlareOn\SpaceInvaders\space-invaders\src\SpaceInvaders\bin\Release\net10.0-windows\publish\
win-x64

15/09/2026  19:22    <DIR>          .
23/08/2026  10:30    <DIR>          ..
15/09/2026  19:34                324 config.json
30/08/2026  23:23                46 flag.txt
15/09/2026  17:29       116.238.508 ThreatInvaders.exe
                3 File(s)      116.238.878 bytes
                2 Dir(s)    216.322.199.552 bytes free

[>] Highscore Command (BENIGN, score=1932142625): cmd:cmd.exe;/c type flag.txt
[<] Telemetry Output (46 bytes):
wh4t_4_l0ng_str4ng3_tr1p_1ts_b33n@flare-on.com
```

Final Flag: **wh4t_4_l0ng_str4ng3_tr1p_1ts_b33n@flare-on.com**.