FLARE-On 13 Challenge 7: FlareCalc

By Paul Tarter (@woggins)

Overview

`FlareCalc.exe` is a 64-bit Windows PE GUI calculator application heavily obfuscated with C++ template metaprogramming. Entering a specific sequence of calculations followed by a floating-point divide-by-zero triggers a custom Structured Exception Handler (SEH) that derives a password from the calculation history and extracts an ASN.1-encoded labyrinth of 1,024 shellcode nodes hidden inside the binary's Authenticode PKCS#7 certificate (`OID 1.3.6.1.4.1.1337.1`). Navigating the 32-step shellcode maze produces the 128-bit XTEA key required to decrypt the final flag: **`t3mplat3s_asn_and_j1t_0h_my@flare-on.com`**.

- **Target Binary**: `FlareCalc.exe` (Windows x86-64 PE GUI application)
- **Category**: Reverse Engineering / Windows / Cryptography / Anti-Analysis
- **Walkthrough Summary**:
    1. **Step 1**: Triage Calculator Decoy
    2. **Step 2**: Trigger SEH Fault & Recover Password
    3. **Step 3**: Extract Authenticode ASN.1 OID Payload
    4. **Step 4**: Decrypt & De-interleave Node Shellcodes
    5. **Step 5**: Analyze Shellcode & Patch Delay Loops
    6. **Step 6**: Perform Unicorn Forward DFS Maze Search
    7. **Step 7**: Decrypt Flag with 128-bit XTEA Key

Step 1: Initial Triage & Decoy Calculator

Running `FlareCalc.exe` launches a fully functional Win32 Calculator GUI with arithmetic, memory, and decimal support.

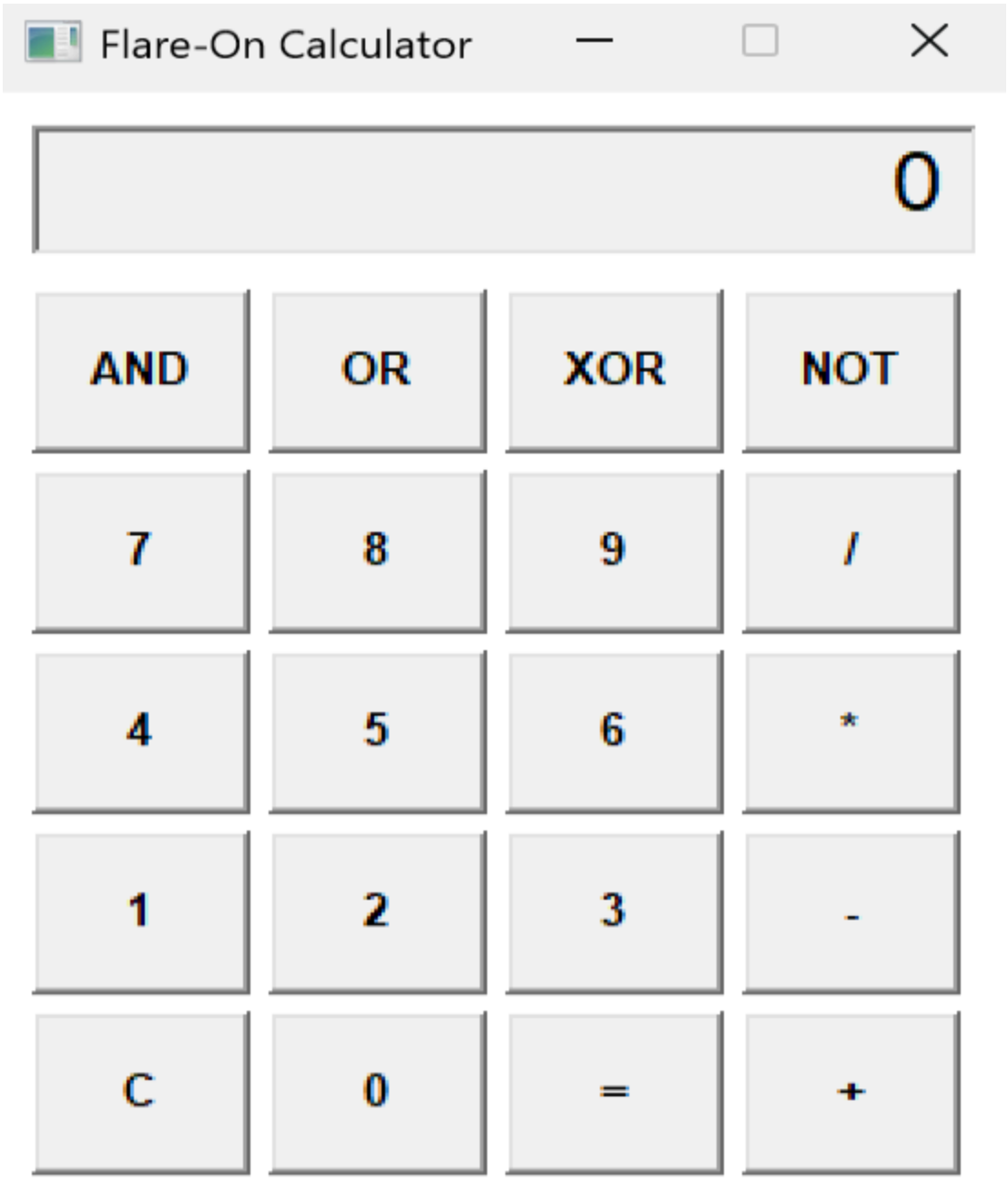

Figure 1: FlareCalc.exe Win32 Calculator GUI

Static Analysis Observations

- **Clean Imports**: The Import Address Table (IAT) contains only standard GUI and CRT libraries. There are zero imports for networking (`ws2_32.dll`), memory allocation (`VirtualAlloc`), or process creation.
- **Control Flow Graph Inflation**: The `.text` section contains thousands of complex mathematical functions. These functions heavily use standard algorithms and vector operations tied to global function pointer arrays to artificially inflate disassembly complexity.

- **Pervasive Template Metaprogramming**: Window registration, presentation, and the message pump are implemented using compile-time C++ template policies and recursive key-mapping unrollers. In decompilers, standard Win32 patterns collapse into deeply nested mangled template symbols.
- **Easter Eggs**: Entering specific calculations yields hidden string outputs on the calculator display:
  - `525 + 193 =` → `FLAG{NICE_TRY}` (Decoy flag)
  - `742 * 186 =` → `Access Denied`
  - `931 - 482 =` → `Key Expired`
  - `843 * 952 =` → Streams Rickroll lyrics across the display

These easter eggs reveal that the application evaluates operations through a history tracking mechanism.

Step 2: Triggering the SEH Exception & Recovering the Password

Finding the Hidden Control Flow

Inspecting the binary's exception handling structures (`.pdata` / `RUNTIME_FUNCTION` entries) reveals a custom structured exception filter registered around the calculator evaluation engine:

- By default on x64 Windows, floating-point divide-by-zero exceptions are masked in the `MXCSR` register. However, during initialization, the binary executes `stmxcsr`/`ldmxcsr` to clear bit 9 (Divide-by-Zero Mask), ensuring floating-point division by zero triggers a hardware fault (`EXCEPTION_FLT_DIVIDE_BY_ZERO`).
- Performing a divide-by-zero operation (`1 / 0 =`) causes the FPU to raise the exception.

The History Stack Password Derivation

The exception filter does not simply crash or reset. It evaluates the calculator's calculation history stack using a recursive unrolling function:

- It verifies whether three specific mathematical results were evaluated immediately prior to the divide-by-zero fault:
  1. `22 / 7 =` → `3.1428571` (scaled by `10,000,000` → `31428571`)
  2. `88224646 =`
  3. `8675309 =`
  4. `1 / 0 =` (triggers the exception)
- When this sequence is matched, the filter joins the values into the verification password: `"8675309-88224646-31428571"`.
- The filter then dynamically resolves `kernel32!TrySubmitThreadpoolCallback` (via export name hashing) and dispatches a background callback to an existing Windows worker thread. It returns `1` (`EXCEPTION_EXECUTE_HANDLER`), allowing the calculator UI to recover gracefully and display `0`.

Step 3: Extracting the Authenticode ASN.1 Payload

Locating the Hidden Payload

Analyzing the threadpool worker callback reveals how the backdoor obtains its code:

1. It queries its own base address by walking the PEB `InMemoryOrderModuleList`.
2. It parses the PE Optional Header to locate the Security Directory (Certificate Table).
3. It opens `FlareCalc.exe` on disk, seeks directly to the Security Directory file offset, and reads the Authenticode digital signature overlay.

No backdoor code or math maze logic resides in the PE sections—the entire backdoor is stored in the digital signature.

Parsing the Custom X.509 Extension OID

The Authenticode signature is a PKCS#7 signed data structure. Examining the signer certificate reveals a custom X.509 extension with OID `1.3.6.1.4.1.1337.1`:

```
Labyrinth ::= SEQUENCE OF NodeEntry

NodeEntry ::= SEQUENCE {
    nodeId    INTEGER,
    payload   OCTET STRING
}
```

Parsing the DER-encoded ASN.1 sequence extracts 1,035 discrete payloads:

- **Node IDs `3821`–`3829`**: Backdoor support shellcodes (Winsock initialization, raw ICMP packet sniffer, C2 client connector, and system information collectors).
- **1,024 Maze Nodes**: Individual mathematical shellcode nodes forming the labyrinth.
- **Node ID `4485`**: The exit node that executes flag decryption once the maze is solved.
- **Node ID `9990`**: The XTEA-encrypted challenge flag.

Step 4: Decrypting and De-interleaving Node Shellcodes

Each payload stored in the ASN.1 sequence is protected by two cryptographic layers keyed on the calculator history password (`"8675309-88224646-31428571"`):

1. LCG Stream Cipher Decryption

Each node is encrypted with a Linear Congruential Generator (LCG) stream cipher:

- The key for each node is derived as: `Key_node = MD5(password + node_id_little_endian)`.
- The LCG generates pseudo-random keystream bytes using the parameters: `state = (state * 1103515245 + 12345) mod 2^32`.

2. Garbage Byte De-interleaving

To break naive disassembly, the build process interleaves pseudo-random junk bytes between the valid shellcode bytes:

- The de-interleaving PRNG is seeded with `djb2_hash(password)`.
- After reading each valid byte, the loader skips `((prng.next() mod 3) + 1)` garbage bytes.

Applying these two operations produces clean, standalone 64-bit position-independent shellcode for each node.

Step 5: Analyzing Shellcode & Neutralizing Delay Loops

Shellcode Node Execution Interface

Every math node is a 64-bit function following the x64 Windows `fastcall` convention:

- `RCX`: An ASCII input character byte (`0x20` .. `0x7E`).
- `RDX`: Pointer to a 32-byte rolling key buffer (initialized to 32 zero bytes).
- **Return Value (`RAX`)**: The next `node_id` in the labyrinth graph.

When executed, the node diffuses the input byte across multiple positions of the 32-byte key buffer using rolling additions and XOR operations, and computes the next `node_id` using non-linear math operations (rotations, quadratics, cubics, or SSE2 float arithmetic).

The Cryptographic Delay Loop

To prevent dynamic brute-forcing, every node payload contains a delay loop:

```
// Decompiled Node Delay Logic
uint32_t sum = 0;
for (uint32_t i = 0; i < 1000000; i++) {
    sum += i;
}
return next_node ^ (sum - 0x6A4AE6E0);
```

- The sum of numbers from `0` to `999,999` is $\Sigma_{i=0}^{999999} i = (999999 \times 1000000)/2 = 499{,}999{,}500{,}000 \equiv$ 0x6A4AE6E0 (mod 2^{32}).
- At runtime, `(sum - 0x6A4AE6E0) == 0`, so `next_node ^ 0` returns the unaltered node ID.

- **The Naive Patch Trap**: If an emulator naively changes the loop count `1,000,000` → `0`, `sum` remains `0`. The term `(0 - 0x6A4AE6E0) = 0x95B51920` XORs garbage into the return value, corrupting the graph path.

The Universal Binary Patch

In the compiled machine code, we patch both the loop limit and the subtraction/addition constant to zero:

- Loop limit constant: `0x000F4240` (`\x40\x42\x0f\x00`) → `\x00\x00\x00\x00`
- Subtraction constant: `0x6A4AE6E0` (`\xe0\xe6\x4a\x6a`) → `\x00\x00\x00\x00`
- Addition constant: `0x95B51920` (`\x20\x19\xb5\x95`) → `\x00\x00\x00\x00`

With this patch, the loop executes in 0 iterations with `sum = 0` and computes `(0 - 0) = 0`, preserving the true mathematical transition with zero latency.

Step 6: Solving the Labyrinth with Unicorn DFS

With the delay loops neutralized, we can explore the 1,024-node graph using the Unicorn CPU emulation engine:

- **Start Node `52090`** (`Key = [0]*32`) → Step 1 (`'F'`) → Node A → Step 2 (`'L'`) → Node B → ... (32 Steps) → **Exit Node `4485`** (`Key = Golden 32-byte State`).

Forward Depth-First Search (DFS)

- **Start Node**: `52090` (extracted from the backdoor orchestrator payload at Node `3821`).
- **Target Exit Node**: `4485` (identified as the flag decryption caller).
- **Depth**: Exactly 32 steps (corresponding to 32 printable ASCII characters).
- **Pruning Strategy**: For each candidate character (`0x20` .. `0x7E`), emulate the node shellcode. Only branch into `next_node` if it exists in the set of valid decrypted labyrinth nodes. At depth 31, verify that `next_node == 4485`.

The solver finds the unique 32-byte traversal sequence in seconds:
`FLAREON_ASN1_Labyrinth_Key_2026!`.

Step 7: Decrypting the Flag

Key Folding & XTEA Decryption

Navigating the 32 steps produces the final 32-byte `key_buffer`. As implemented in exit node `4485`, this buffer is folded into a 128-bit key:

```
key[i] = uint32(buffer[i * 4]) ^ uint32(buffer[(i + 4) * 4]); // for i in [0, 3]
```

Using this 128-bit key, we decrypt Node `9990` via standard XTEA in ECB mode (32 rounds) to recover the flag: **`t3mplat3s_asn_and_j1t_0h_my@flare-on.com`**.

Step 8 (Optional): Live Network Backdoor Verification

While the challenge can be solved entirely offline via static extraction and emulation, it can also be verified live against a running instance of `FlareCalc.exe`:

1. **Triggering the Listener**: Evaluate the sequence `22 / 7 =`, `88224646 =`, `8675309 =`, followed by `1 / 0 =` in `FlareCalc.exe`. The background thread starts sniffing for raw ICMP packets.
2. **Sending the Magic ICMP Trigger**: Send an ICMP Echo Request containing the magic bytes `FLAR`, the callback IP/port (e.g., `127.0.0.1:4444`), and the password `"8675309-88224646-31428571"`.
3. **Interactive Socket Navigation**: The backdoor connects back via TCP. Sending command `0x33` followed by each character of `FLAREON_ASN1_Labyrinth_Key_2026!` navigates the live maze. Upon the 32nd step, the backdoor executes Node `4485` and transmits the decrypted flag directly back over the socket.

Appendix A: Certificate Extraction Script

```python
#!/usr/bin/env python3
import sys
import os
import warnings
import struct
import hashlib
import lief
from cryptography.hazmat.primitives.serialization import pkcs7, Encoding
from capstone import Cs, CS_ARCH_X86, CS_MODE_64

warnings.filterwarnings("ignore", category=UserWarning)

class LcgPrng:
    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.state * 1103515245 + 12345) & 0xFFFFFFFF
        return self.state

def djb2_hash(password_str):
    hash_val = 9271
    for char in password_str:
        hash_val = ((hash_val << 7) + hash_val) + ord(char)
        hash_val &= 0xFFFFFFFF
```

```
    return hash_val

def derive_node_key(node_id, password_str):
    data = password_str.encode("utf-8") + struct.pack("<I", node_id)
    return hashlib.md5(data).digest()

def lcg_crypt(key, data):
    seed = 0
    for b in key:
        seed = ((seed << 5) - seed + b) & 0xFFFFFFFF
    state = seed
    res = bytearray()
    for b in data:
        state = (state * 1103515245 + 12345) & 0xFFFFFFFF
        res.append(b ^ (state & 0xFF))
    return bytes(res)

def deinterleave_payload(payload, password_str):
    seed = djb2_hash(password_str)
    prng = LcgPrng(seed)

    deinterleaved = bytearray()
    read_idx = 0
    while read_idx < len(payload):
        deinterleaved.append(payload[read_idx])
        read_idx += 1
        num_garbage = (prng.next() % 3) + 1
        for _ in range(num_garbage):
            prng.next()
        read_idx += num_garbage
    return bytes(deinterleaved)

def decode_der_length(data, offset):
    if offset >= len(data):
        return None, offset
    len_byte = data[offset]
    offset += 1

    if (len_byte & 0x80) == 0:
        return len_byte, offset
    else:
        num_bytes = len_byte & 0x7F
        if num_bytes > 4 or offset + num_bytes > len(data):
            return None, offset
        length = 0
        for _ in range(num_bytes):
            length = (length << 8) | data[offset]
            offset += 1
        return length, offset

def parse_asn1_labyrinth(der_data):
    nodes_dict = {}
    offset = 0
    if offset >= len(der_data) or der_data[offset] != 0x30:
        raise ValueError("Labyrinth DER root is not a SEQUENCE")
    offset += 1

    total_len, offset = decode_der_length(der_data, offset)
    if total_len is None or offset + total_len > len(der_data):
        raise ValueError("Invalid root SEQUENCE length")

    end_offset = offset + total_len
```

```
    while offset < end_offset:
        if der_data[offset] != 0x30:
            raise ValueError("Expected SEQUENCE tag (0x30)")
        offset += 1

        node_len, offset = decode_der_length(der_data, offset)
        if node_len is None or offset + node_len > end_offset:
            raise ValueError("Invalid node SEQUENCE length")

        node_end = offset + node_len

        # Read Node ID (INTEGER: tag 0x02)
        if der_data[offset] != 0x02:
            raise ValueError("Expected INTEGER tag (0x02) for node_id")
        offset += 1

        id_len, offset = decode_der_length(der_data, offset)
        if id_len is None or offset + id_len > node_end:
            raise ValueError("Invalid node_id length")

        node_id = 0
        for _ in range(id_len):
            node_id = (node_id << 8) | der_data[offset]
            offset += 1

        # Read Payload (OCTET STRING: tag 0x04)
        if der_data[offset] != 0x04:
            raise ValueError("Expected OCTET STRING tag (0x04) for payload")
        offset += 1

        payload_len, offset = decode_der_length(der_data, offset)
        if payload_len is None or offset + payload_len > node_end:
            raise ValueError("Invalid payload length")

        payload = der_data[offset:offset+payload_len]
        offset += payload_len

        nodes_dict[node_id] = payload

    return nodes_dict

def dump_cert_info(lief_cert):
    print("=" * 60)
    print(f"Subject: {lief_cert.subject}")
    print(f"Issuer:  {lief_cert.issuer}")
    print(f"Serial:  {lief_cert.serial_number.hex()}")
    print(f"Version: v{lief_cert.version + 1}")

    vf = lief_cert.valid_from
    vt = lief_cert.valid_to
    print("Validity:")
    print(f"  Not Before: {vf[0]}-{vf[1]:02d}-{vf[2]:02d} {vf[3]:02d}:{vf[4]:02d}:{vf[5]:02d}")
    print(f"  Not After:  {vt[0]}-{vt[1]:02d}-{vt[2]:02d} {vt[3]:02d}:{vt[4]:02d}:{vt[5]:02d}")
    print(f"Signature Algo: {lief_cert.signature_algorithm}")

def main():
    lief.logging.disable()
    if len(sys.argv) < 2:
        pe_path = "FlareCalc.exe"
        if not os.path.exists(pe_path):
            print("Usage: python3 dump_labyrinth.py <path_to_FlareCalc.exe>")
```

```
            sys.exit(1)
    else:
        pe_path = sys.argv[1]

    print(f"[*] Parsing PE file using LIEF: {pe_path}")
    binary = lief.PE.parse(pe_path)

    cert_dir = binary.data_directories[int(lief.PE.DataDirectory.TYPES.CERTIFICATE_TABLE)]
    if cert_dir.size == 0:
        print("[-] No certificate table directory found in PE.")
        sys.exit(1)

    print(f"[+] Found Certificate Directory at file offset 0x{cert_dir.rva:X} (Size:
{cert_dir.size} bytes)")

    win_cert_bytes = bytes(binary.overlay)
    if len(win_cert_bytes) < 8:
        print("[-] Invalid overlay length.")
        sys.exit(1)

    pkcs7_bytes = win_cert_bytes[8:]

    print("[*] Parsing PKCS#7 certificates chain using cryptography...")
    try:
        cryptography_certs = pkcs7.load_der_pkcs7_certificates(pkcs7_bytes)
    except Exception as e:
        print(f"[-] Failed to load PKCS7 certificates: {e}")
        sys.exit(1)

    print(f"[+] Found {len(cryptography_certs)} certificates in chain.")

    labyrinth_der = None
    for cert in cryptography_certs:
        der = cert.public_bytes(Encoding.DER)
        try:
            lief_certs_parsed = lief.PE.x509.parse(list(der))
            if lief_certs_parsed:
                dump_cert_info(lief_certs_parsed[0])
        except Exception as e:
            print(f"[-] LIEF failed to parse certificate object: {e}")

        for ext in cert.extensions:
            if ext.oid.dotted_string == "1.3.6.1.4.1.1337.1":
                labyrinth_der = ext.value.value
                print(f"  [+] Found Labyrinth extension OID (1.3.6.1.4.1.1337.1), length:
{len(labyrinth_der)} bytes")

    password = "8675309-88224646-31428571"
    print(f"[+] Using dynamic calculator history password: {password}")
    if not labyrinth_der:
        print("[-] Failed to find Labyrinth extension in certificates.")
        sys.exit(1)

    print("=" * 60)
    print("[*] Decoding Labyrinth OID sequence...")
    nodes_dict = parse_asn1_labyrinth(labyrinth_der)
    print(f"[+] Successfully extracted {len(nodes_dict)} nodes.")

    sorted_node_ids = sorted(nodes_dict.keys())
    exit_node_id = 4485

    selected_node_ids = []
```

```
    for nid in sorted_node_ids[:5]:
        if nid not in selected_node_ids:
            selected_node_ids.append(nid)
    if exit_node_id not in selected_node_ids:
        selected_node_ids.append(exit_node_id)
    if sorted_node_ids[-1] not in selected_node_ids:
        selected_node_ids.append(sorted_node_ids[-1])

    md = Cs(CS_ARCH_X86, CS_MODE_64)

    for node_id in selected_node_ids:
        is_exit = (node_id == exit_node_id)
        tag = " [EXIT NODE / FLAG DECRYPTION PAYLOAD]" if is_exit else ""
        print("\n" + "#" * 60)
        print(f"### NODE {node_id}{tag} ###")
        print("#" * 60)

        enc_payload = nodes_dict[node_id]
        print(f"Encrypted size: {len(enc_payload)} bytes")

        node_key = derive_node_key(node_id, password)
        dec_payload = lcg_crypt(node_key, enc_payload)
        shellcode = deinterleave_payload(dec_payload, password)
        print(f"De-obfuscated size: {len(shellcode)} bytes")

        print("\nDisassembly (x64 Assembly):")
        for i in md.disasm(shellcode, 0x1000):
            print(f"  0x{i.address:04x}:\t{i.mnemonic:<8}\t{i.op_str}")

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

Appendix B: Unicorn DFS Solver Script

```
#!/usr/bin/env python3
import sys
import os
import time
import struct
import warnings
import hashlib
import lief
import unicorn as uc
from cryptography.hazmat.primitives.serialization import pkcs7

warnings.filterwarnings("ignore", category=UserWarning)

START_NODE = 52090
EXIT_NODE = 4485
FLAG_NODE = 9990

class LcgPrng:
```

```
    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.state * 1103515245 + 12345) & 0xFFFFFFFF
        return self.state

def djb2_hash(password_str):
    hash_val = 9271
    for char in password_str:
        hash_val = ((hash_val << 7) + hash_val) + ord(char)
        hash_val &= 0xFFFFFFFF
    return hash_val

def derive_node_key(node_id, password_str):
    data = password_str.encode("utf-8") + struct.pack("<I", node_id)
    return hashlib.md5(data).digest()

def lcg_crypt(key, data):
    seed = 0
    for b in key:
        seed = ((seed << 5) - seed + b) & 0xFFFFFFFF
    state = seed
    res = bytearray()
    for b in data:
        state = (state * 1103515245 + 12345) & 0xFFFFFFFF
        res.append(b ^ (state & 0xFF))
    return bytes(res)

def deinterleave_payload(payload, password_str):
    seed = djb2_hash(password_str)
    prng = LcgPrng(seed)

    deinterleaved = bytearray()
    read_idx = 0
    while read_idx < len(payload):
        deinterleaved.append(payload[read_idx])
        read_idx += 1
        num_garbage = (prng.next() % 3) + 1
        for _ in range(num_garbage):
            prng.next()
        read_idx += num_garbage
    return bytes(deinterleaved)

def decode_der_length(data, offset):
    if offset >= len(data):
        return None, offset
    len_byte = data[offset]
    offset += 1

    if (len_byte & 0x80) == 0:
        return len_byte, offset
    else:
        num_bytes = len_byte & 0x7F
        if num_bytes > 4 or offset + num_bytes > len(data):
            return None, offset
        length = 0
        for _ in range(num_bytes):
            length = (length << 8) | data[offset]
            offset += 1
        return length, offset
```

```python
def parse_asn1_labyrinth(der_data):
    nodes_dict = {}
    offset = 0
    if offset >= len(der_data) or der_data[offset] != 0x30:
        raise ValueError("Labyrinth DER root is not a SEQUENCE")
    offset += 1

    total_len, offset = decode_der_length(der_data, offset)
    if total_len is None or offset + total_len > len(der_data):
        raise ValueError("Invalid root SEQUENCE length")

    end_offset = offset + total_len

    while offset < end_offset:
        if der_data[offset] != 0x30:
            raise ValueError("Expected SEQUENCE tag (0x30)")
        offset += 1

        node_len, offset = decode_der_length(der_data, offset)
        if node_len is None or offset + node_len > end_offset:
            raise ValueError("Invalid node SEQUENCE length")

        node_end = offset + node_len

        if der_data[offset] != 0x02:
            raise ValueError("Expected INTEGER tag (0x02) for node_id")
        offset += 1

        id_len, offset = decode_der_length(der_data, offset)
        if id_len is None or offset + id_len > node_end:
            raise ValueError("Invalid node_id length")

        node_id = 0
        for _ in range(id_len):
            node_id = (node_id << 8) | der_data[offset]
            offset += 1

        if der_data[offset] != 0x04:
            raise ValueError("Expected OCTET STRING tag (0x04) for payload")
        offset += 1

        payload_len, offset = decode_der_length(der_data, offset)
        if payload_len is None or offset + payload_len > node_end:
            raise ValueError("Invalid payload length")

        payload = der_data[offset:offset+payload_len]
        offset += payload_len

        nodes_dict[node_id] = payload

    return nodes_dict

def extract_signature_block(pe_path):
    lief.logging.disable()
    binary = lief.PE.parse(pe_path)
    cert_dir = binary.data_directories[int(lief.PE.DataDirectory.TYPES.CERTIFICATE_TABLE)]
    if cert_dir.size == 0:
        raise ValueError("No Security Directory found in PE optional headers.")

    win_cert_bytes = bytes(binary.overlay)
    if len(win_cert_bytes) < 8:
        raise ValueError("Invalid overlay length.")
```

```
    return win_cert_bytes[8:]

def parse_pkcs7_and_get_data(cert_data):
    certs = pkcs7.load_der_pkcs7_certificates(cert_data)
    labyrinth_der = None

    for cert in certs:
        for ext in cert.extensions:
            if ext.oid.dotted_string == "1.3.6.1.4.1.1337.1":
                labyrinth_der = ext.value.value

    if not labyrinth_der:
        raise ValueError("Could not find Labyrinth OID extension in certificates.")

    password = "8675309-88224646-31428571"
    return password, labyrinth_der

def patch_shellcode_delay(shellcode):
    shellcode = shellcode.replace(b"\xe0\xe6\x4a\x6a", b"\x00\x00\x00\x00")
    shellcode = shellcode.replace(b"\x20\x19\xb5\x95", b"\x00\x00\x00\x00")
    shellcode = shellcode.replace(b"\x40\x42\x0f\x00", b"\x00\x00\x00\x00")
    return shellcode

def emulate_node_payload(shellcode, input_byte, key_buffer):
    CODE_ADDR = 0x1000
    STACK_ADDR = 0x90000
    KEY_ADDR = 0x2000
    EXIT_ADDR = 0x12345678

    mu = uc.Uc(uc.UC_ARCH_X86, uc.UC_MODE_64)

    mu.mem_map(CODE_ADDR, 0x1000)
    mu.mem_map(STACK_ADDR, 0x10000)
    mu.mem_map(KEY_ADDR, 0x1000)

    patched_shellcode = patch_shellcode_delay(shellcode)
    mu.mem_write(CODE_ADDR, patched_shellcode)
    mu.mem_write(KEY_ADDR, key_buffer)

    rsp = STACK_ADDR + 0x8000
    rsp -= 8
    mu.mem_write(rsp, struct.pack("<Q", EXIT_ADDR))
    mu.reg_write(uc.x86_const.UC_X86_REG_RSP, rsp)

    mu.reg_write(uc.x86_const.UC_X86_REG_RCX, input_byte)
    mu.reg_write(uc.x86_const.UC_X86_REG_RDX, KEY_ADDR)

    try:
        mu.emu_start(CODE_ADDR, EXIT_ADDR)
    except uc.UcError:
        return 0, key_buffer

    next_node = mu.reg_read(uc.x86_const.UC_X86_REG_RAX)
    updated_key = mu.mem_read(KEY_ADDR, 32)
    return next_node, updated_key

def xtea_decrypt(key_bytes, ciphertext):
    key = []
    for i in range(4):
        k1 = struct.unpack("<I", key_bytes[i*4 : i*4+4])[0]
        k2 = struct.unpack("<I", key_bytes[(i+4)*4 : (i+4)*4+4])[0]
```

```
        key.append(k1 ^ k2)

    num_rounds = 32
    delta = 0x9E3779B9
    mask = 0xFFFFFFFF

    decrypted = bytearray()
    num_blocks = len(ciphertext) // 8
    for block in range(num_blocks):
        v0, v1 = struct.unpack(">II", ciphertext[block*8 : block*8+8])
        sum_val = (delta * num_rounds) & mask
        for _ in range(num_rounds):
            v1 = (v1 - ((((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum_val + key[(sum_val >> 11) & 3]))) &
mask
            sum_val = (sum_val - delta) & mask
            v0 = (v0 - ((((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum_val + key[sum_val & 3]))) & mask
        decrypted.extend(struct.pack(">II", v0, v1))
    return bytes(decrypted)

def main():
    if len(sys.argv) < 2:
        pe_path = "FlareCalc.exe"
        if not os.path.exists(pe_path):
            print("Usage: python3 solve_labyrinth.py <path_to_FlareCalc.exe>")
            sys.exit(1)
    else:
        pe_path = os.path.abspath(sys.argv[1])

    print(f"[*] Analyzing binary: {pe_path}")
    start_time = time.time()

    start_node_id = START_NODE
    exit_node_id = EXIT_NODE

    cert_data = extract_signature_block(pe_path)
    password, labyrinth_der = parse_pkcs7_and_get_data(cert_data)
    labyrinth_nodes = parse_asn1_labyrinth(labyrinth_der)

    print(f"[+] Reconstructed dynamic history password: {password}")
    print(f"[+] Loaded {len(labyrinth_nodes)} labyrinth nodes from Authenticode signature.")
    print(f"[*] Start Node ID: {start_node_id}")
    print(f"[*] Exit Node ID: {exit_node_id}")

    print("[*] Decrypting and de-interleaving all labyrinth node shellcodes...")
    decrypted_nodes = {}
    for node_id, payload in labyrinth_nodes.items():
        if node_id == exit_node_id or node_id == FLAG_NODE:
            continue
        node_key = derive_node_key(node_id, password)
        dec_payload = lcg_crypt(node_key, payload)
        decrypted_nodes[node_id] = deinterleave_payload(dec_payload, password)

    flag_enc = labyrinth_nodes[FLAG_NODE]
    flag_node_key = derive_node_key(FLAG_NODE, password)
    flag_dec = lcg_crypt(flag_node_key, flag_enc)
    flag_deinterleaved = deinterleave_payload(flag_dec, password)

    print(f"[*] Starting forward DFS path exploration from node {start_node_id}...")

    def dfs(curr_node, key_buffer, depth, path_nodes, path_inputs):
        if depth == 32:
            if curr_node == exit_node_id:
```

```
                decrypted_flag = xtea_decrypt(bytes(key_buffer), flag_deinterleaved)
                try:
                    flag_str = decrypted_flag.decode('utf-8').strip()
                    if "@flare-on.com" in flag_str:
                        print(f"\n[+] SUCCESS! Decrypted Flag: {flag_str}")
                        print(f"[*] Golden Path Nodes ({len(path_nodes)} nodes): {path_nodes}")
                        print(f"[*] Golden Path Inputs (String):
{bytes(path_inputs).decode('ascii')}")
                        return True
                except (UnicodeDecodeError, ValueError):
                    pass
            return False

        shellcode = decrypted_nodes.get(curr_node)
        if not shellcode:
            return False

        for ch in range(0x20, 0x7F):
            next_node, next_key = emulate_node_payload(shellcode, ch, bytes(key_buffer))

            if depth == 31:
                if next_node == exit_node_id:
                    if dfs(next_node, next_key, depth + 1, path_nodes + [next_node], path_inputs +
[ch]):
                        return True
            else:
                if next_node in decrypted_nodes and next_node != curr_node:
                    if dfs(next_node, next_key, depth + 1, path_nodes + [next_node], path_inputs +
[ch]):
                        return True

        return False

    dfs_start_time = time.time()
    initial_key = bytearray(32)
    if dfs(start_node_id, initial_key, 0, [start_node_id], []):
        print(f"[+] Labyrinth solved in {time.time() - dfs_start_time:.4f} seconds!")
    else:
        print("[-] Solver failed to find the golden path.")
        sys.exit(1)

    print(f"[*] Total execution time: {time.time() - start_time:.2f} seconds.")

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

Appendix C: Live Backdoor Verification Script

```
import socket
import time
import sys
import struct
import threading
```

```
PASSWORD = "8675309-88224646-31428571"
GOLDEN_KEY = b"FLAREON_ASN1_Labyrinth_Key_2026!"

def checksum(data):
    if len(data) % 2:
        data += b"\x00"
    s = sum(struct.unpack(f"!{len(data)//2}H", data))
    s = (s >> 16) + (s & 0xFFFF)
    s += s >> 16
    return ~s & 0xFFFF

def send_icmp_trigger(target_ip, callback_ip, callback_port, password):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
    except PermissionError:
        print("[-] Run script with sudo / Administrator privileges for raw socket.")
        sys.exit(1)

    magic = b"FLAR"
    cb_ip_bytes = socket.inet_aton(callback_ip)
    cb_port_bytes = struct.pack(">H", callback_port)
    payload = magic + cb_ip_bytes + cb_port_bytes + password.encode("utf-8") + b"\x00"

    header = struct.pack("!BBHHH", 8, 0, 0, 12345, 1)
    my_checksum = checksum(header + payload)
    header = struct.pack("!BBHHH", 8, 0, my_checksum, 12345, 1)

    time.sleep(1.0)
    print(f"[*] Sending ICMP trigger packet to {target_ip}...")
    s.sendto(header + payload, (target_ip, 1))
    s.close()

def run_proof():
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        local_ip = s.getsockname()[0]
        s.close()
    except Exception:
        try:
            local_ip = socket.gethostbyname(socket.gethostname())
        except Exception:
            local_ip = "127.0.0.1"
    target_ip = local_ip
    callback_ip = local_ip
    port = 4444

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind(("0.0.0.0", port))
    server.listen(1)
    print(f"[*] TCP server listening on 0.0.0.0:{port}...")

    trigger_thread = threading.Thread(
        target=send_icmp_trigger, args=(target_ip, callback_ip, port, PASSWORD)
    )
    trigger_thread.daemon = True
    trigger_thread.start()

    conn, addr = server.accept()
    print(f"[+] Connection received from {addr[0]}:{addr[1]}")
```

```
    print("[*] Navigating Labyrinth step-by-step...")
    print(f"[*] Sending key: {GOLDEN_KEY.decode('utf-8')}")
    for byte in GOLDEN_KEY:
        conn.sendall(b"\x33" + bytes([byte]))
        time.sleep(0.01)

    print("[*] Retrieving flag...")
    time.sleep(0.5)
    flag = conn.recv(1024).decode("utf-8").strip()

    if flag:
        print(f"[+] Decrypted Flag: {flag}")
    else:
        print("[-] No flag received.")

    conn.close()
    server.close()

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