Itaming CTF - Comprehensive Step-by-Step Writeup

1) This guide details how to solve the `**inthebenigging**` and `**forensics100.php**` challenges on Kali Linux. It includes every command you need to type in your terminal.

Prerequisites
Open your terminal and navigate to the challenge directory:
```bash
cd "/home/nopalinto/Downloads/Gemini CTF/"
```

Challenge 1: `inthebenigging` (Binary Exploitation)

Objective: Find the flag hidden in the executable binary.

Phase 1: Analysis
First, let's understand what we are dealing with.

1. **Make the binary executable:**
   ```bash
   chmod +x inthebenigging
   ```

2. **Check the file type:**
   ```bash
   file inthebenigging
   ```
   Output: It confirms this is a 64-bit ELF executable.

3. **Look for readable strings:**
   ```bash
   strings inthebenigging | grep -i "flag"
   ```
   Observation: You will see `ictff8{%s}` which confirms the flag format. Other strings like `cerPi` (TracerPid backwards) suggest the binary is checking for debuggers.

The Problem: If you try to run it with a debugger (like `ltrace` or `gdb`), it detects it and closes, or worse, it changes the way it calculates the flag hash so you get the wrong answer.

Phase 2: The Attack (Side-Channel)
Instead of cracking the binary, we will "spy" on it. The binary compares your input against the correct flag using the `strncmp` function. We will

force the binary to use *our* version of `strncmp` that logs the expected flag to the screen.

Step 1: Create the "Spy" Library
We need to write a small piece of C code.

1. Create the file:
    ```bash
    nano hook.c
    ```

2. Paste the following code into the editor:
    ```c
    #define _GNU_SOURCE
    #include <dlfcn.h>
    #include <stdio.h>
    #include <string.h>

    // Pointer to the original strncmp function
    static int (*real_strncmp)(const char*, const char*, size_t) = NULL;

    // Our spy function
    int strncmp(const char *s1, const char *s2, size_t n) {
        // Load the original function if not loaded
        if (!real_strncmp) {
            real_strncmp = dlsym(RTLD_NEXT, "strncmp");
        }

        // If it's comparing 2 bytes (which this binary does for the hash), log it!
        // s1 is usually what the binary expects, s2 is what you typed.
        if (n == 2) {
            fprintf(stderr, "SPY: Expected='%s' vs Input='%s'\n", s1, s2);
        }

        // Run the original function so the program doesn't crash
        return real_strncmp(s1, s2, n);
    }
    ```

3. Save and exit (`Ctrl+O`, `Enter`, `Ctrl+X`).

Step 2: Compile the Library
Compile `hook.c` into a shared library (`hook.so`) using `gcc`.

```bash
gcc -shared -fPIC -o hook.so hook.c -ldl
```

```

#### Step 3: Create the Automating Script
Now we need a script to run the binary over and over, reading the "SPY" logs to piece together the flag.

1. Create the script:
    ```bash
    nano solve_bin.py
    ```

2. Paste this Python code:
    ```python
    import subprocess
    import os
    import re

    # Configuration
    binary_path = "./inthebenigging"
    hook_path = "./hook.so"
    known_flag_hash = "" # We will build this up

    print("[-] Starting Brute-Force via Side-Channel...")

    # The hash is 32 hex characters long
    while len(known_flag_hash) < 32:
        # Create a dummy flag with what we know so far + padding
        # Format: ictff8{<KNOWN_HASH>...0000}
        current_input = known_flag_hash + ("0" * (32 -
len(known_flag_hash)))
        full_input = f"ictff8{{{current_input}}}"

        # Prepare environment to load our spy hook
        env = os.environ.copy()
        env['LD_PRELOAD'] = hook_path

        # Run the binary
        proc = subprocess.Popen(
            [binary_path],
            stdin=subprocess.PIPE,
            stderr=subprocess.PIPE,
            stdout=subprocess.DEVNULL,
            env=env,
            text=True
        )

        # Send our input
```
```

```
        _, stderr = proc.communicate(input=full_input + "\n")

        # Look for our SPY logs
        # We need the comparison occurring at our current position
        # The binary compares 2 bytes at a time.
        # So if we have 4 bytes known, we are looking for the 3rd
comparison chunk.

        chunk_index = len(known_flag_hash) // 2
        matches = re.findall(r"SPY: Expected='(..)'", stderr)

        if len(matches) > chunk_index:
            next_byte = matches[chunk_index]
            known_flag_hash += next_byte
            print(f"[+] Found chunk: {next_byte} -> Flag Hash:
{known_flag_hash}")
        else:
            print("[!] Failed to find next chunk. Quitting.")
            break

    print(f"\n[SUCCESS] FINAL FLAG: ictff8{{{known_flag_hash}}}")
    ```

3. Save and exit (`Ctrl+O`, `Enter`, `Ctrl+X`).

Step 4: Run the Solution
Execute the python script.

```bash
python3 solve_bin.py
```

Final Output:
```
[SUCCESS] FINAL FLAG: ictff8{693411d6e73f77f566a181e056fa217e}
```

Challenge 2: `forensics100.php` (Forensics)

Objective: Decode the massive, obfuscated PHP file to find the flag.

Phase 1: Analysis
1. **Check file size:**
    ```bash
    ls -lh forensics100.php
```

    ```
    *Output:* It's about 19MB. This is huge for a text file.

2. **Inspect contents:**
    ```bash
    head -n 1 forensics100.php
    ```
    Output: You will see `<?php eval(base64_decode('...` repeated. This is a "Russian Doll" obfuscation. The code decodes itself, runs `eval`, which decodes more code, runs `eval`, and so on.

Phase 2: The Solution (Automated Decoding)
We cannot decode this by hand because there are 83 layers! We will write a PHP script to do it for us.

Prerequisite: Ensure you have PHP installed.
```bash
sudo apt update
sudo apt install php-cli
```

Step 1: Create the Solver Script
1. Create the file:
    ```bash
    nano solve_forensics.php
    ```

2. Paste this PHP code:
    ```php
    <?php
    // Increase memory limit for the 19MB file processing
    ini_set('memory_limit', '4G');

    $file = "forensics100.php";
    echo "[*] Reading $file...\n";
    $code = file_get_contents($file);

    $depth = 0;

    // Loop until we find the flag or crash
    while (true) {
        $depth++;

        // 1. Hunt for the Flag
        if (preg_match('/ictff8\{[^}]+\}/', $code, $matches)) {
            echo "\n\n[SUCCESS] FLAG FOUND at Depth $depth!\n";
            echo ">> " . $matches[0] . " <<\n\n";

```
            exit;
        }

        // 2. Hunt for the obfuscation pattern
        // We look for: eval(...);
        // We capture the content inside eval()
        if (preg_match('/eval\((.+)\);/s', $code, $m)) {
            $payload = $m[1];

            // 3. Decode without executing
            // We trick PHP into returning the code instead of running it
            $decoded = eval("return $payload;");

            if ($decoded) {
                $code = $decoded;
                echo "Depth $depth: Decoded layer... (Size: " .
strlen($code) . " bytes)\r";
            } else {
                echo "\n[!] Decoding failed at depth $depth.\n";
                break;
            }
        } else {
            echo "\n[!] No more eval() calls found.\n";
            break;
        }
    }
    ?>
    ```

3. Save and exit (`Ctrl+O`, `Enter`, `Ctrl+X`).

Step 2: Run the Solver
Execute the PHP script.

```bash
php solve_forensics.php
```

It will churn through the layers (Depth 1, Depth 2... Depth 83).

Final Output:
```
[SUCCESS] FLAG FOUND at Depth 83!
>> ictff8{9a7e738ae6687f1d179144bedb34fd5d} <<
```

Detailed Write-up: Supercomputer.exe Reverse Engineering

Challenge Information
- **Challenge Name:** bin400
- **File:** supercomputer.exe
- **Platform:** Windows x64

Executive Summary
The executable prompts for a specific input, performs a massive mathematical calculation (Factorial), and uses the resulting digits to generate a flag. The native implementation is deliberately inefficient ("Supercomputer" theme), requiring us to reverse the logic and implement an optimized solver in Python to get the flag instantly.

Step-by-Step Analysis

1. Input Validation
Upon decompiling the `main` function (Address `0x140148AA0`), we observe a loop processing a user-supplied string. The program reads 10 bytes but validates the first 6 bytes using a chain of XOR operations.

Derivation of the correct input:
The checks are performed sequentially or can be solved backwards:
1. `Input[5] ^ 0x53 == 0x0B`
   => `Input[5] = 0x53 ^ 0x0B = 0x58 ('X')`

2. `Input[4] ^ Input[5] == 0x2A`
   => `Input[4] = Input[5] ^ 0x2A = 0x58 ^ 0x2A = 0x72 ('r')`

3. `Input[3] ^ Input[4] == 0x17`
   => `Input[3] = Input[4] ^ 0x17 = 0x72 ^ 0x17 = 0x65 ('e')`

4. `Input[2] ^ Input[3] == 0x15`
   => `Input[2] = Input[3] ^ 0x15 = 0x65 ^ 0x15 = 0x70 ('p')`

5. `Input[1] ^ Input[2] == 0x09`
   => `Input[1] = Input[2] ^ 0x09 = 0x70 ^ 0x09 = 0x79 ('y')`

6. `Input[0] ^ Input[1] == 0x31`
   => `Input[0] = Input[1] ^ 0x31 = 0x79 ^ 0x31 = 0x48 ('H')`

Resulting Valid Input: **"HyperX"**

2. Logic Determination
Once "HyperX" is entered, the program performs the following setup:
1. Calculates the Sum of the ASCII values of the input:
   `Sum("HyperX") = 72 + 121 + 112 + 101 + 114 + 88 = 608`.
2. Calculates a `Limit` value:

`Limit = Sum * 82 = 608 * 82 = 49856`.

The program then enters a function `GetSeed` which implements a Big Integer Factorial calculation (`49856!`).
- It uses a buffer in the `.data` section to store decimal digits.
- It performs multiplication via repeated addition loops, which is computationally prohibitive for such a large number.

3. Flag Generation Algorithm
The flag is derived from the digits of `49856!`.
1. **Initialize** a 32-byte array with the value `1`.
2. **Iterate** through the digits of the factorial from Most Significant to Least Significant.
3. **Update** the flag array in a round-robin fashion (indices 0 to 31):
   `Flag[i] = (Flag[i] + FactorialDigit[j]) % 10`

4. Solution Script
A Python script (`RE/solver.py`) was written to replicate this logic efficiently:
- Uses `math.factorial(49856)` to compute the number instantly.
- Converts the result to a string/list of digits.
- Simulates the round-robin addition logic.

Final Flag
ictff8{50858403313122412227446072397578}

Challenge 3: The Russian Doll (Forensics)

CTF Write-up: evidence.bin
Flag: ictff8{binwalk_is_your_best_friend}

Description
We have intercepted a confidential case file named evidence.bin. Analyze the file to retrieve the hidden flag.

Solution: Static Analysis (Strings)
1. Initial Inspection The challenge provides a binary file evidence.bin. While binary files often require specialized tools for analysis, a common "sanity check" in CTF competitions is to inspect the raw contents of the file for visible plain text.

2. Execution We opened the file using a standard text editor (Notepad). While much of the file contained repeating characters or unreadable data, scrolling to the end revealed readable ASCII text.

3. Discovery Near the bottom of the file, we located a file header and the flag in plain text:

flag.txtictff8{binwalk_is_your_best_friend}

Conclusion
Sometimes the simplest tools are the most effective. By checking the raw text first, we bypassed the need for extraction tools and found the flag immediately.

Challenge 4: RAID 0 Recovery (`disk_0.bin` & `disk_1.bin`)

Objective: Reconstruct a file from two "disk" fragments to find the hidden flag.

Phase 1: Analysis
We are given two files: `disk_0.bin` (60 bytes) and `disk_1.bin` (59 bytes).

1. **Inspect the Hex Dumps:**
    ```bash
    xxd disk_0.bin
    xxd disk_1.bin
    ```
    Observation:
    * `disk_0.bin` starts with `89 50 4e 47` (the start of a PNG signature).
    * `disk_1.bin` starts with `0d 0a 1a 0a` (the second half of a PNG signature).
    * `disk_0.bin` also contains parts like `00 00 00 0d` (IHDR length).
    * `disk_1.bin` contains `49 48 44 52` (IHDR type).

The Pattern: This is a **RAID 0** storage pattern with a **stripe size of 4 bytes**. Every 4 bytes of the original file alternate between `disk_0` and `disk_1`.

Phase 2: The Solution (Python Reconstruction)

We can write a script to "interweave" these 4-byte blocks back into a single file.

Step 1: Create the Solver Script
1. Create the file:
    ```bash
    nano solve_raid.py
    ```

2. Paste this Python code:
    ```python
    ```

```
    def reconstruct():
        with open('disk_0.bin', 'rb') as f:
            d0 = f.read()
        with open('disk_1.bin', 'rb') as f:
            d1 = f.read()

        reconstructed = bytearray()
        stripe_size = 4

        # Interleave 4-byte chunks
        for i in range(0, max(len(d0), len(d1)), stripe_size):
            reconstructed.extend(d0[i:i+stripe_size])
            reconstructed.extend(d1[i:i+stripe_size])

        with open('reconstructed.png', 'wb') as f:
            f.write(reconstructed)
        print("[+] Reconstructed file saved as reconstructed.png")

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

Step 2: Extract the Flag
1. Run the script:
    ```bash
    python3 solve_raid.py
    ```

2. Check for strings in the output file:
    ```bash
    strings reconstructed.png
    ```

Observation: You will see a `tEXt` chunk metadata entry containing the flag.

Final Output:
```
ictff8{R41D_0_R3c0v3ry_M4st3r}
```