Sploit 1 description

The memory for variables unsigned I, len and buffer are allocated on the stack after the processor "prolog" with the following assembly code:

```
sub     $0xbd8,%esp
```

Just beside this memory lies the Saved Frame Pointer (SFP) and the return address RET.

We find the address of the buffer in gdb by entering the following commands:

```
print $ebp - 0xbd8
```

```
result: (void *) 0xffbfd060
```

We can hence overwrite the RET by overflowing the buffer as no checks are in place. However, the variable "i" is used as a counter in the copyFile function. Suppose we were to fill our buffer with the value "A" 3016 times.

```
i = 0;
c = fgetc(source);
while (c != EOF) {
  buffer[i]  = (unsigned char) c;
  c = fgetc(source);
  i++;
}
```

On paper we would be overwriting the RET address. However, the variable "i" is stored above buffer. By overflowing the buffer, we would be inadvertently overwriting the LSB of "i" with the value A which would change the counter preventing the buffer from being overwritten.

In order to get past the variable "i", we would need to write the value 0xC4 into the payload text file's 3005^{th}. On the for loop's 3004^{th} iteration, "i" would have a value of 0xBBC. Overwriting the MSB changes it to 0xBC4 which equals to 3012 in decimal causing the counter to skip past the memory locations containing the SFP and the counter variable "i" and directly into overwriting the RET address.

Sploit 1 fix

A simple way to fix this flaw in the programming is to first read the file and check the number of characters. If the number of characters read exceeds the buffer size, then the function would return and the buffer does not get overflowed.

Sploit 2

For Sploit2, we a targeting a format string vulnerability in the usage function. Specifically, the printf function in usage() as shown below.

```
static
void usage(char* parameter)
{
  char output[200];

  snprintf(output, sizeof(output),
        "Usage: %.110s backup|restore|ls pathname\n", parameter);

  printf(output);
}
```

By using the printf function this way, users would be able to read the program's memory and write to arbitrary addresses via the use of format specifiers. This means that any format specifiers in **output** would be interpreted as formatting information. As a result, writing %x into output would cause the printf function to print out an element on the stack as a hexadecimal character even if it isn't one.

The exploit is done by storing the shell code in the environment variable unlike sploit1 since we do not have a buffer to store it in this time.

Next, we leverage the use of format specifiers to write our return address into the saved EIP. The return address we are writing would be **0xffbfdf76** which is the location of our environment variable.

The format specifier %n allows us to write 32 bits into an address contained in the stack. By using %.08x we can print out the stack's memory contents. Printing it 10 times shows us that the start of the **output** string starts at the 10th stack element relative to the printf starting pointer.

First, we separate our the Saved EIP's address at **0xffbfdccc** into its two lower and upper bytes – **0xffbfdccc** and **0xffbfdcce**. Doing so allow us to write the lower two bytes first followed by the two upper bytes with the following string:

```
args[0] = " \xcc\xdc\xbf\xff\xce\xdc\xbf\xff%57200c%10$n%8255c%11$n";
```

Note that alignment is done in 16 bytes by default in gcc. Since our format string starts with 9 characters "<space>\xcc\xdc\xbf\xff\xce\xdc\xbf\xff", padding is added by the compiler to align the stack.

The reason why we write the lower address bits first is because the address we wanted to write is **0xffbfdf68**. 0xffbf is bigger than 0xdf76. %n writes a value equivalent to the number of characters printed before the specifier. This means that in our format string, we wrote the lower bytes first as 0xdf68 is smaller than 0xffbf.

Due to the padded bytes, the number of characters we had to write were off by a few bytes. The actual value needed to be written ended up being 57200 + number of padded bytes. This was done by observing the written values before adjustment using gdb.

Sploit 2 fix

The most common method of avoiding format string vulnerabilities is to never allow the use of a string as an input. If that cannot be avoided, we can extract the format specifiers out from the string to "sterilize" it