New Zealand Cyber Security Challenge 2018
Round One Writeup

1: The provided file is a disk image; looking inside it, there are several images, one of which cannot be opened. The first four bytes of this file appear to be a ZIP file, while the rest are an image; by replacing the first four bytes with the correct magic bytes for the file, the image becomes viewable, and the flag can be seen in the image.

2: Only some data appears to be returned by the server; by looking at the response code, we see the server is responding with 206 Partial Content, which indicates that there is more data to be sent. We can specify a range of data from the server by including a HTTP Range request. We can modify the HTTP request to include `Range: bytes=0-` , the server will then respond with a complete message, which includes the flag.

Read: https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests

3: The provided file is a Wireshark packet capture file; looking at its contents, we see it is some type of file transfer. We see that both a username and password were transmitted in plaintext, as well as some ZIP file. If we extract the ZIP file, we see that it is encrypted, and requires a password; by trying the password we found earlier in the capture, we can decrypt it and retrieve the flag.

4: Several hashes are provided to the user; they are 256-bits, so the hash function used is likely SHA-256. If we guess that the flag itself has been hashed to give us one of these hashes, we can then immediately try to use a rainbow table. Using a dictionary attack will not work, since the input values are almost-random. Most online rainbow tables will find the input values used.

5: When the challenge loads it decrypts the list of messages. Inside this is the flag but due to a bug the flag is overwritten when the password is checked. The easiest way to retrieve the flag is to put a breakpoint just before the code that overwrites it. The console message "Error Decrypting Flag" points to one such location to put a breakpoint.

6: The provided file is a disk image; Rename Customers.xlsx to Customer.zip, Unzip Customer.zip, open xl\config.xml and get the flag

7: The provided script is incomplete; several constant values contain the character `?`, and part of a function has been left unencrypted. To begin, the partially defined function should be implemented; the required line would be similar to `` `(a * seed + c) % m` ``. To fix the rest of the script, the user needs to loop through 0 .. (1 << 24), and setting the lower bits of `x0` and `a` to be these values. If these values are used in the LCG, and generate the second value, `x1`, we can recover the LCG used to generate the flag. By returning this LCG, the rest of the script will run and generate the flag, printing to stdout.

8: User input was being passed through to the shell, in the form: `` `passthru("grep -i \"$data\" log.txt");` ``. Key characters such as `'` and `;` were blocked, preventing users from escaping out of the shell and running actual commands. The log file contained a hint as to where the flag could be found; challenge8/flag.txt. This file could not be directly read since .htaccess specifically blocked it. However, we did not block subshell usage, meaning if input like `` `flag$(grep -v flag:guessthevalue flag.txt)` `` was given, the user could instead make the server read the flag.txt file, and report back to the user if 'guessthevalue' was a correct guess. Since grep supports regex, it is possible to guess one character at a time.

9: Several known plaintexts are given, all ending with "Over.". A base64 glob is given, which is said to be data encrypted with a 64-bit XOR cipher. We can use "Over." to recover the last 5 bytes of the key, leaving 3 bytes unknown. At this point, it is not practical to brute-force the rest of the key, however if the partially decrypted message is inspected, the fragment "fla" occurs. By guessing that the next characters are likely "g:", another 2 bytes of the key can be recovered. The final byte can either be bruteforced, or by noticing the fragment "ET?:1200", and guessing the missing letter to be an A, the entire key can be recovered.

10: The provided script has a known plaintext and ciphertext pair, along with the encrypted flag. The aim of this challenge is to recover the key, and decrypt the flag. The encryption scheme is as follows: the input is encrypted with one 16-bit key, and then again with another 16-bit key (see Double-DES); this is vulnerable to a meet-in-the-middle attack, which can be summarized as follows:

```
    middle = dict([ (encrypt(known_plaintext, key1), key1) for key1 in
make_keys() ])
    for key2 in make_keys():
        tmp = decrypt(known_ciphertext, key2)
        if tmp in middle:
            key1 = middle[tmp]
            print(decrypt(decrypt(flag_ciphertext, key2), key1))
            break
```

11: The general method behind this challenge was to find the use of the flag in the executable and, using that pointer, reveal the flag. It is also possible to remove the assembly instructions that edit the flag causing it to print to standard out.

12: The method for solving this challenge was to remove the alpha challenge of the image revealing a pattern encoded into the blue channel. This pattern was a four pixel separate most significant bit encoding that can be directly read as a bit pattern and mapped to ASCII by hand. It is also possible to automate this process although it is unnecessary.