

Serpentine

Finally, a function `str_xor(secret, key)` took the secret and key, and applied a repeating XOR between the two. This encoded flag was stored as a concatenation of `chr()` values, thus obfuscating the contents from immediate viewing. The decoding function then created a key of the same length with the secret and then performed the XOR on a byte basis.

To finally extract the flag, I have isolated the minimum components needed for decoding – the `str_xor` function and the `flag_enc` variable. Finally, the entire byte sequence was passed through the `str_xor` function and the known key, `enkidu`, to convert it to ASCII characters and reveal the full flag.

It can also be passed as a one-shot Python command straight to the terminal:

```
python3 - << 'EOF'
```

```
def str_xor(secret, key):
```

```
    new_key = key
```

```
    i = 0
```

```
    while len(new_key) < len(secret):
```

```
        new_key = new_key + key[i]
```

```
        i = (i + 1) % len(key)
```

```
    return "".join([chr(ord(s)^ord(k)) for s,k in zip(secret,new_key)])
```

```
flag_enc =
```

```
chr(0x15)+chr(0x07)+chr(0x08)+chr(0x06)+chr(0x27)+chr(0x21)+chr(0x23)+chr(0x15)+chr(0x5c)+chr(0x01)+chr(0x57)+chr(0x2a)+chr(0x17)+chr(0x5e)+chr(0x5f)+chr(0x0d)+chr(0x3b)+chr(0x19)+chr(0x56)+chr(0x5b)+chr(0x5e)+chr(0x36)+chr(0x53)+chr(0x07)+chr(0x51)+chr(0x18)+chr(0x58)+chr(0x05)+chr(0x57)+chr(0x11)+chr(0x3a)+chr(0x0f)+chr(0x0a)+chr(0x5b)+chr(0x57)+chr(0x41)+chr(0x55)+chr(0x0c)+chr(0x59)+chr(0x14)
```

```
flag = str_xor(flag_enc, 'enkidu')
```

```
print(flag)
```

```
EOF
```

This approach to the CTF challenge highlights the importance of analyzing the code, identifying the transformation logic rather than attempting to brute force it, and the potential use of XOR-based encoding with CTF practices.