Malware Obfuscation Techniques

- *obfuscation*: the process of obscuring meaningful information.

In addition to obfuscation, attackers also use encoding/encryption techniques, which makes reversing much more difficult.

Reasons for encoding and encrypting:

- Conceal command and control communication
- Hide from signature-based solutions, such as IPS
- Obscure the content of the configuration file used by the malware
- Encrypt information to be exfiltrated from the victim system
- Obfuscate strings, to hide from static analysis

Definitions:

- *Plaintext:* Unencrypted message
- *Cyphertext:* Encrypted message

Malware encrypts the *plaintext*, by passing it as input along with the *key* to an encryption function, which produces a *cyphertext*. The resulting *cyphertext* is then used to write to file or send over the network.

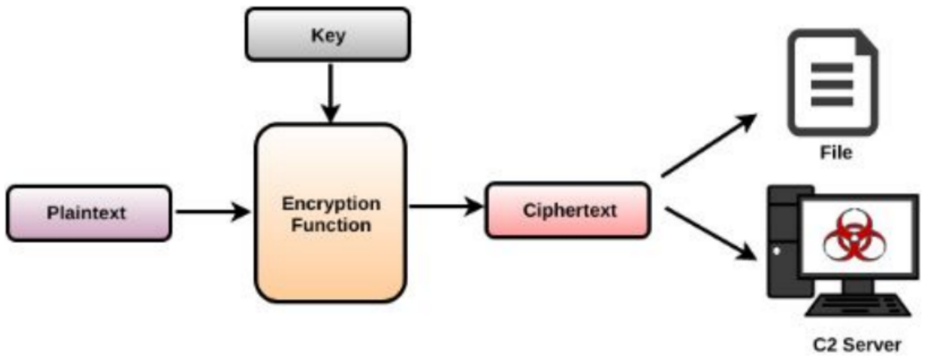


Figure 1: Malware Encryption

Malware may receive an encrypted content from the C2 server and then decrypt it by passing the encrypted content and the key to the decryption function.

To understand how a particular content is encrypted or decrypted, you will mainly focus on identifying the encryption or decryption functions and the key used to encrypt or decrypt the content.

1. Simple Encoding

Most of the time, attackers use very simple encoding, such as `Base64 encoding` or `xor encryption` to obscure the data. This is because they use very little system resources and are just enough to obscure the content from the security products and the security analyst.

1.1 Caesar Cypher

Also known as shift-cypher. Encodes the message by shifting each letter in the plain text with some fixed number of positions down the alphabet.

The following diagram shows the character set, the encryption, and the decryption of the text “ZEUS” using 3 as the key.

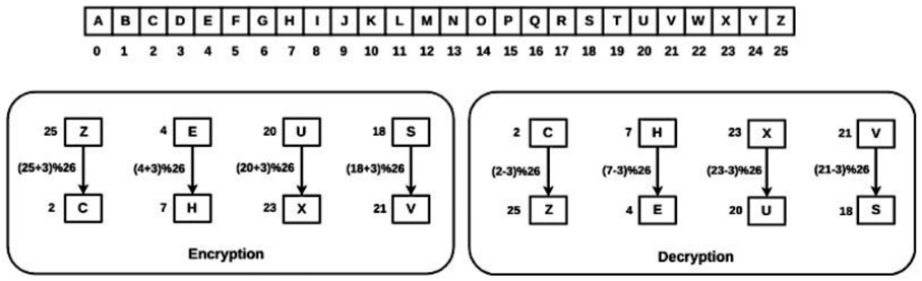


Figure 2: Example of Caesar cypher

1.1.2 Decryption of Caesar Cypher in Python

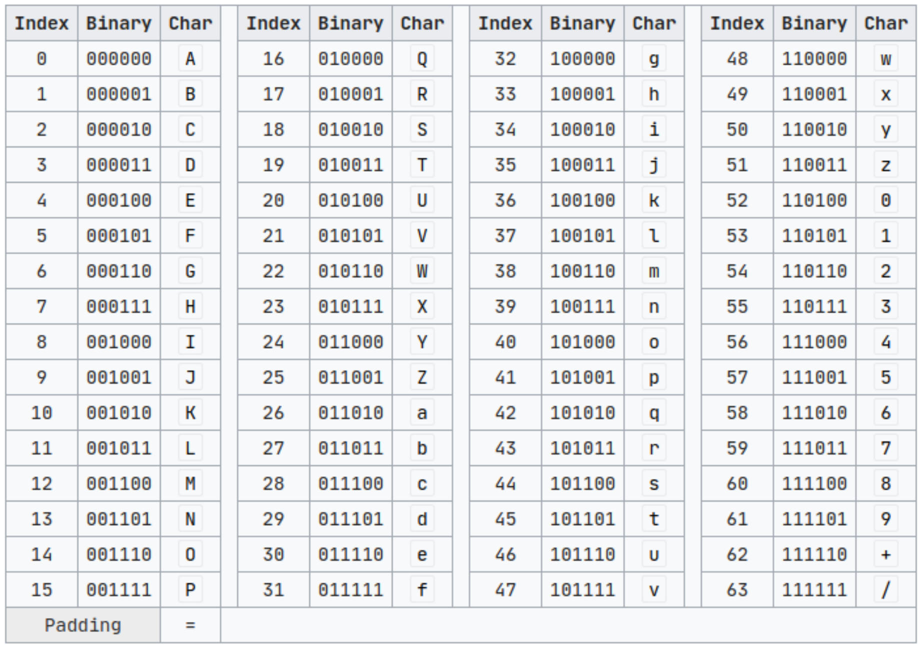
```
chr_set = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
key = 3
cypher_text = "CHXV"
plain_text = ""
for ch in cypher_text:
    j = chr_set.find(ch.upper())
    plain_index = (j - key) % len(chr_set)
    plain_text += chr_set[plain_index]
print plain_text
```

1.2 Base64 Encoding

Unlike Caesar cypher, Base64 **can** encode binary data. It allows an attacker to encode binary data to an ASCII string format.

1.2.1 Translating data to Base64

Standard Base64 encoding consists of the following 64-character set. Each 3 bytes (24 bits) of the binary data that you want to encode is translated into four characters from the character set mentioned above. Each translated character is 6 bits in size. In addition to the following characters, the = character is used for padding.



Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Figure 3: Base64 Index Table

If we want to encode text `One`, we need to convert the letters to their corresponding bit values:

```
O -> 0x4f -> 01001111 n -> 0x6e -> 01101110 e -> 0x65 ->
01100101
```

The Base64 algorithm processes 3 bytes at a time; In this case, we have exactly `24` bits that are placed next to each other:

```
010011110110111001100101
```

The `24` bits are then split in four parts, each consisting of `6` bits and converted to its equivalent decimal value. The decimal value is then used as an index to find the corresponding value in the `Base64` index table, so the text `One` encodes to `T251`:

```
010011 -> 19 -> Base64 table lookup -> T    110110 -> 54
-> Base64 table lookup -> 2 111001 -> 57 -> Base64 table
lookup -> 5 100101 -> 37 -> Base64 table lookup -> 1
```

1.2.2 Encoding and Decoding Base64

In Python:

```
# Encode text in Base64

import base64
plain_text = "One"
encoded = base64.b64encode(plain_text)
print encoded

# Decode text in Base64

import base64
encoded = "T251"
decoded = base64.b64encode(encoded)
print decoded
```

1.2.3 Decoding Custom Base64

Some malware samples delete the padding character `=` from the end. When decoding the encoded string, there will be a `Padding error`. Upon inspection, it is clear that the `len(encoded_string)` is not a multiple of 4. To obtain the encoded string, add back the `=` characters to make it a multiple of 4 again.

Sometimes, characters are replaced. For example, `+ /` characters are replaced with `- _` characters. Replace the characters in the encoded string and decode to obtain the decoded text.

Sometimes, the authors reorder the character in the character set. To decode this:

```
import base64
chr_set = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
non_chr_set = "0123456789+/ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz="
encoded = "G6JgP6w="
re_encoded = ""
for en_ch in encoded:
    re_encoded += en_ch.replace(en_ch, chr_set[non_chr_set.find(en_ch)])
decoded = base64.b64encode(re_encoded)
print decoded
```

1.2.4 Identifying Base64

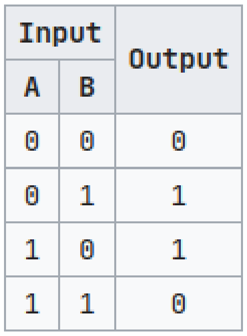
You can identify a binary using Base64 encoding by looking for a long string comprising the Base64 character set.

Using `IDA PRO` *string cross-reference* feature, you can land at the place the string is used. In the case the malware authors encrypt the C2 network traffic with other encryption algorithms, and then uses `Base64` encoding, looking at the `Base64` character set, will probably land you at the `Base64` function. Further analysis of the function, may help you identify the functions that call the `Base64` function.This will probably land you at the encryption function.

Another method of detecting the presence of the `Base64` character set is using a YARA rule.

1.3 XOR Encoding

Bitwise operation performed on the corresponding bits of the operands.



Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4: XOR Truth Table

1.3.1 Single Byte XOR

Each byte from the plaintext is `XORed` with the encryption key.

Encrypting `cat` with a key of `0x40`, then, each character from the text is `XORed` with `0x40`, which results in the cipher-text `#!4`.

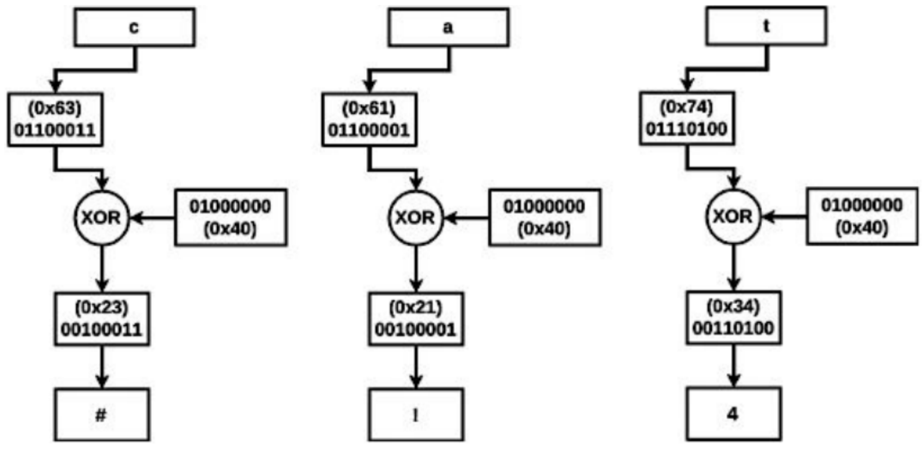
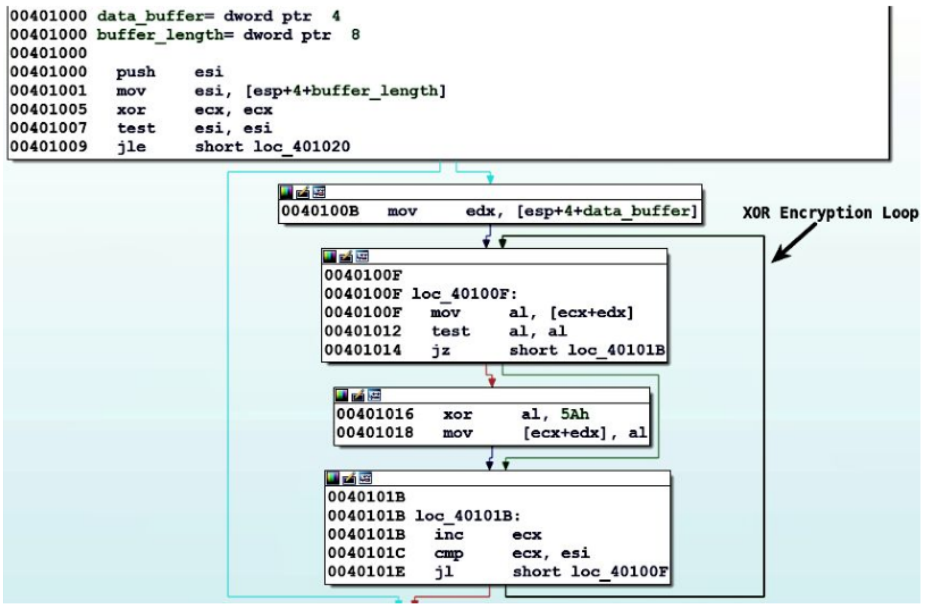


Figure 5: XOR example with cat

When you XOR the cypher text with the same `XOR` key, you get back the plain text.



```
00401000 data_buffer= dword ptr  4
00401000 buffer_length= dword ptr  8
00401000
00401000   push    esi
00401001   mov     esi, [esp+4+buffer_length]
00401005   xor     ecx, ecx
00401007   test    esi, esi
00401009   jle     short loc_401020

0040100B   mov     edx, [esp+4+data_buffer]

0040100F
0040100F loc_40100F:
0040100F   mov     al, [ecx+edx]
00401012   test    al, al
00401014   jz      short loc_40101B

00401016   xor     al, 5Ah
00401018   mov     [ecx+edx], al

0040101B
0040101B loc_40101B:
0040101B   inc     ecx
0040101C   cmp     ecx, esi
0040101E   jl      short loc_40100F
```

Figure 6: XOR encryption loop disassembled

1.3.2 Finding XOR Key Through Brute-Force

In a single byte `XOR`, the length of the key is one byte, so there are only `255` possible keys (`0x0 - 0xff` with the exception of `0` as the `key` because `XORing` any value with `0` will give the same value as output.

This technique is useful when you know what to find in the decrypted data. The following example shows a `brute-force` expecting `mymachine` as output.

```
def xor_brute-force(content, to_match):
    for key in range(256):
        translated = ""
        for ch in content:
            translated += chr(ord(ch) ^ key)
        if to_match in translated:
            print "Key %s(0x%x): %s" % (key, jey, translated)

xor_brute-force("lkwpjeia>i}ieglmja", "mymachine")

key 4(0x4): hostname:mymachine
```

1.3.3 NULL Ignoring XOR Encoding

In `XOR` encoding, when a null byte `(0x00)` is `XORed` with a key, you get back the key as shown here:

```
ch = 0x00

key = 4

ch ^ key

4
```

You can use this to find the key if there is a buffer with a large number of null bytes encoded. This property makes it easy to spot the key if the null bytes are not ignored. To avoid the null byte problem, malware authors ignore the null byte.

1.3.4 Multi-byte XOR Encoding

More common among attackers because it provides better defense against brute-force technique. An attacker can encrypt a PE file with a 4-bit `XOR` key and decrypt at compile time.

1.3.5 Identifying XOR Encoding

Load the binary in IDA and search for the `XOR` instruction. It is very common to see `XOR` operations where the operands are the same memory register. This is used by the compiler to zero out register values. To identify `XOR` encoding, look for:

- `XOR` of a register (or memory reference) with a constant value:
  - `XOR esi 0EAD4AA34h`
- `XOR` of a register with a different register (or memory reference)
  - `XOR esi edi`

Some programs used to determine the `XOR` key:

- *CyberChef*
- *XORSearch by Didier Stevens*
- *Balbuzard*
- *unXOR*
- *rrxor.py*
- *NoMoreXOR.py*