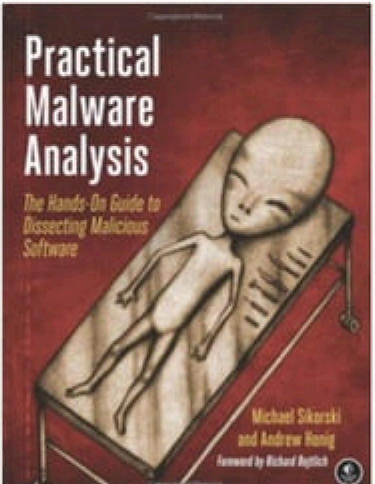
Practical Malware Analysis

Ch 13: Data Encoding



Revised 4-25-16

The Goal of Analyzing Encoding Algorithms

Reasons Malware Uses Encoding

- Hide configuration information
  - Such as C&C domains
- Save information to a staging file
  - Before stealing it
- Store strings needed by malware
  - Decode them just before they are needed
- Disguise malware as a legitimate tool
  - Hide suspicious strings

Simple Ciphers

Why Use Simple Ciphers?

- They are easily broken, but
  - They are small, so they fit into space-constrained environments like exploit shellcode
  - Less obvious than more complex ciphers
  - Low overhead, little impact on performance
- These are *obfuscation*, not *encryption*
  - They make it difficult to recognize the data, but can't stop a skilled analyst

Caesar Cipher

- Move each letter forward 3 spaces in the alphabet

  ABCDEFGHIJKLMNOPQRSTUVWXYZ

  DEFGHIJKLMNOPQRSTUVWXYZABC

- Example

  ATTACK AT NOON

  DWWDFN DW QRRQ

XOR

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0

- Uses a key to encrypt data
- Uses one bit of data and one bit of the key at a time
- Example: Encode HI with a key of 0x3c

HI = 0x48 0x49 (ASCII encoding)

Data: 0100 1000 0100 1001

Key: 0011 1100 0011 1100

Result: 0111 0100 0111 0101

A	T	T	A	C	K		A	T		N	O	O	N
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E
}	h	h	}	DEL	W	FS	}	H	FS	r	s	s	r
0x7d	0x68	0x68	0x7d	0x7F	0x77	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72

Figure 14-1. The string ATTACK AT NOON *encoded with an XOR of 0x3C (original string at the top; encoded strings at the bottom)*

XOR Reverses Itself

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0

- Example: Encode HI with a key of 0x3c

  HI = 0x48 0x49 (ASCII encoding)

  Data: 0100 1000 0100 1001

  Key: 0011 1100 0011 1100

- Encode it again

  Result: 0111 0100 0111 0101

  Key: 0011 1100 0011 1100

  Data: 0100 1000 0100 1001

Brute-Forcing XOR Encoding

- If the key is a single byte, there are only 256 possible keys
  - Error in book; this should be "a.exe"
  - PE files begin with MZ

Example 14-1. First bytes of XOR-encoded file a.gif

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12    _HB.............
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12    ........R.......
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12    ................
12 12 12 12 12 12 12 12 12 12 12 12 12 13 12 12    ................
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82    ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61    Fz{a2b`}u`s.2.ga
```

MZ = 0x4d 0x5a

Table 14-1. Brute-Force of XOR-Encoded Executable

XOR key value	Initial bytes of file	MZ header found?
Original	5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12	No
XOR with 0x01	5e 49 43 13 11 13 13 13 17 13 1c 13 ec ec 13	No
XOR with 0x02	5d 4a 40 10 12 10 10 10 14 10 1f 10 ef ef 10	No
XOR with 0x03	5c 4b 41 11 13 11 11 11 15 11 1e 11 ee ee 11	No
XOR with 0x04	5b 4c 46 16 14 16 16 16 12 16 19 16 e9 e9 16	No
XOR with 0x05	5a 4d 47 17 15 17 17 17 13 17 18 17 e8 e8 17	No
...	...	No
XOR with 0x12	4d 5a 50 00 02 00 00 00 04 00 0f 00 ff ff 00	Yes!

Example 14-2. First bytes of the decrypted PE file

```
4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00    MZP.............
B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00    ........@.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00    ................
BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90    ........!..L.!..
54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73    This program mus
```

Link Ch 13a

computer-forensics.sans.org/blog/2013/05/14/tools-for-examining-xor-obfuscation-for-malware-analysis

Tools for Examining XOR Obfuscation for Malware Analysis

2 comments Posted by Lenny Zeltser
Filed under Computer Forensics, Incident Response, Malware Analysis, Reverse Engineering

There are numerous ways of concealing sensitive data and code within malicious files and programs. Fortunately, attackers use one particular XOR-based technique very frequently, because offers sufficient protection and is simple to implement. Here's a look at several tools for deobfuscating XOR-encoded data during static malware analysis. We'll cover XORSearch, XORStrings, xorBruteForcer, brutexor and NoMoreXOR.

Brute-Forcing Many Files

- Look for a common string, like "This Program"

Table 14-2. Creating XOR Brute-Force Signatures

XOR key value	"This program"
Original	54 68 69 73 20 70 72 6f 67 72 61 6d 20
XOR with 0x01	55 69 68 72 21 71 73 6e 66 73 60 6c 21
XOR with 0x02	56 6a 6b 71 22 72 70 6d 65 70 63 6f 22
XOR with 0x03	57 6b 6a 70 23 73 71 6c 64 71 62 6e 23
XOR with 0x04	50 6c 6d 77 24 74 76 6b 63 76 65 69 24
XOR with 0x05	51 6d 6c 76 25 75 77 6a 62 77 64 68 25
...	...
XOR with 0xFF	ab 97 96 8c df 8f 8d 90 98 8d 9e 92 df

XOR and Nulls

- A null byte reveals the key, because
  - 0x00 xor KEY = KEY
- Obviously the key here is 0x12

Example 14-1. First bytes of XOR-encoded file a.gif

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12    _HB.............
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12    ........R.......
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12    ................
12 12 12 12 12 12 12 12 12 12 12 12 12 13 12 12    ................
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82    ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61    Fz{a2b`}u`s.2.ga
```

NULL-Preserving Single-Byte XOR Encoding

- Algorithm:
  - Use XOR encoding, EXCEPT
  - If the plaintext is NULL or the key itself, skip the byte

Table 14-3. Original vs. NULL-Preserving XOR Encoding Code

Original XOR	NULL-preserving XOR
`buf[i] ^= key;`	`if (buf[i] != 0 && buf[i] != key)` `    buf[i] ^= key;`

Example 14-1. First bytes of XOR-encoded file a.gif

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12    _HB.............
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12    ........R.......
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12    ................
12 12 12 12 12 12 12 12 12 12 12 12 12 13 12 12    ................
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82    ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61    Fz{a2b`}u`s.2.ga
```

Example 14-3. First bytes of file with NULL-preserving XOR encoding

```
5F 48 42 00 10 00 00 00 16 00 1D 00 ED ED 00 00    _HB.............
AA 00 00 00 00 00 00 00 52 00 08 00 00 00 00 00    ........R.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 13 00 00    ................
A8 02 00 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82    ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61    Fz{a2b`}u`s.2.ga
```

Identifying XOR Loops in IDA Pro

- Small loops with an XOR instruction inside
  1. Start in "IDA View" (seeing code)
  2. Click **Search, Text**
  3. Enter **xor** and **Find all occurrences**

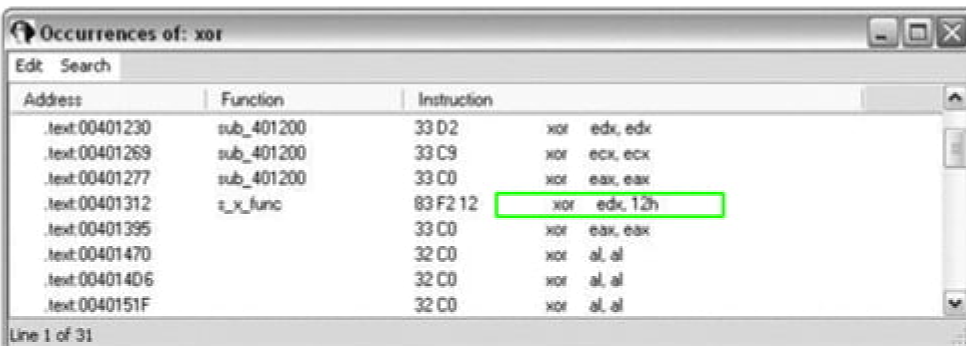


Figure 14-2. Searching for XOR in IDA Pro

Three Forms of XOR

- XOR a register with itself, like **xor edx, edx**
  - Innocent, a common way to zero a register
- XOR a register or memory reference with a constant
  - May be an encoding loop, and key is the constant
- XOR a register or memory reference with a different register or memory reference
  - May be an encoding loop, key less obvious

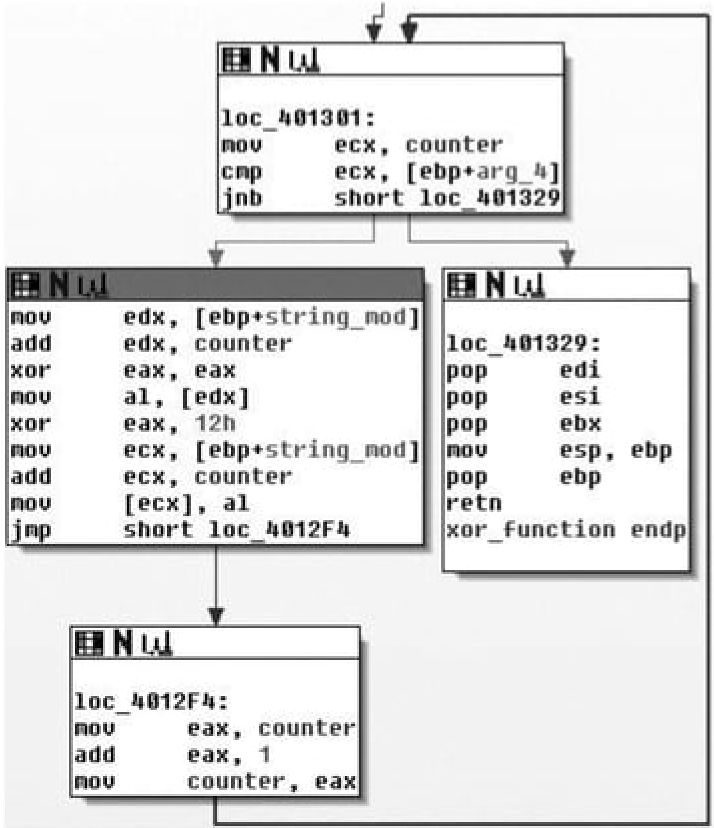


Figure 14-3. Graphical view of single-byte XOR loop

Table 14-4. Additional Simple Encoding Algorithms

Encoding scheme	Description
ADD, SUB	Encoding algorithms can use ADD and SUB for individual bytes in a manner that is similar to XOR. ADD and SUB are not reversible, so they need to be used in tandem (one to encode and the other to decode).
ROL, ROR	Instructions rotate the bits within a byte right or left. Like ADD and SUB, these need to be used together since they are not reversible.
ROT	This is the original Caesar cipher. It's commonly used with either alphabetical characters (*A–Z* and *a–z*) or the 94 printable characters in standard ASCII.
Multibyte	Instead of a single byte, an algorithm might use a longer key, often 4 or 8 bytes in length. This typically uses XOR for each block for convenience.
Chained or loopback	This algorithm uses the content itself as part of the key, with various implementations. Most commonly, the original key is applied at one side of the plaintext (start or end), and the encoded output character is used as the key for the next character.

Base64

- Converts 6 bits into one character in a 64-character alphabet
- There are a few versions, but all use these 62 characters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
```

- MIME uses + and /
  - Also = to indicate padding

Example 14-4. Part of raw email message showing Base64 encoding

```
Content-Type: multipart/alternative;
    boundary="_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_"
MIME-Version: 1.0
--_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: base64

SWYgeW91IGFyZSByZWFkaW5nIHRoaXMsIHlvdSBwcm9iYWJseSBzaG91bGQganVzdCBza2lwIHRoaX
MgY2hhcHRlciBhbmQgZ28gdG8gdGhlIG5leHQgb25lLiBEbyB5b3UgcmVhbGx5IGhhdmUgdGhlIHRp
bWUgdG8gdHlwZSB0aGlzIHdob2xlIHN0cmluZyBpbj8gWW91IGFyZSBvYnZpb3VzbHkgdGFsZW50ZW
QuIE1heWJlIHlvdSBzaG91bGQgY29udGFjdCB0aGUgYXV0aG9ycyBhbmQgc2VlIGlmIH
```

Transforming Data to Base64

- Use 3-byte chunks (24 bits)
- Break into four 6-bit fields
- Convert each to Base64

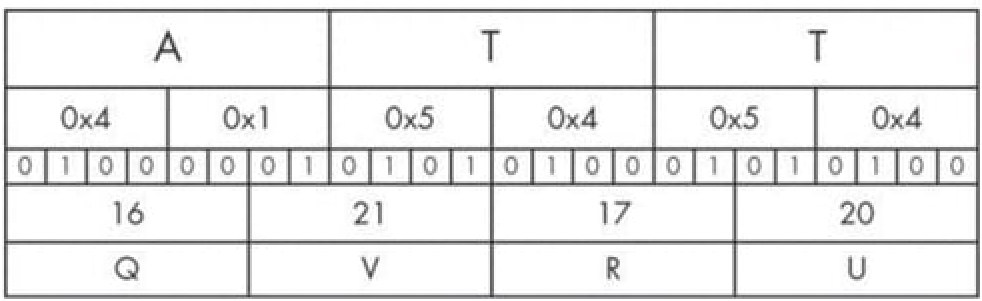
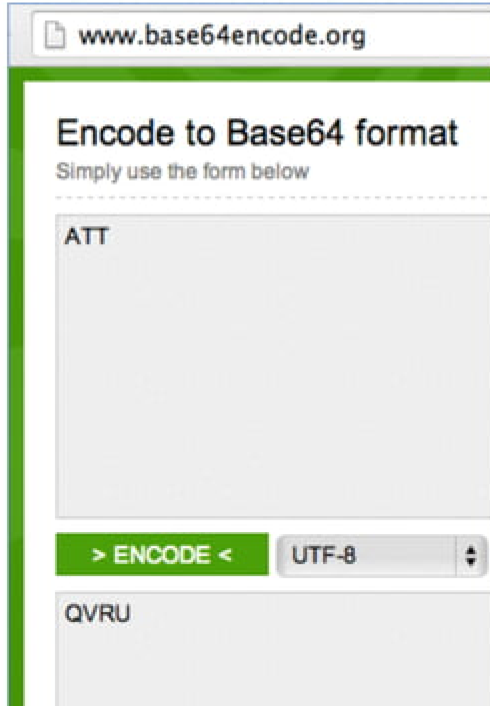


Figure 14-4. Base64 encoding of ATT

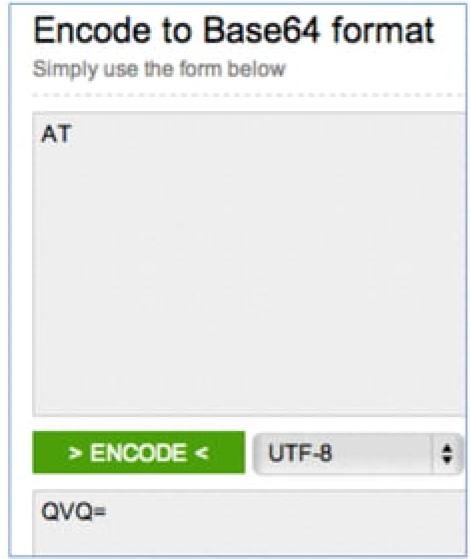
base64encode.org
base64decode.org

- 3 bytes encode to 4 Base64 characters



Padding

- If input had only 2 characters, an = is appended



Padding

- If input had only 1 character, == is appended

Example

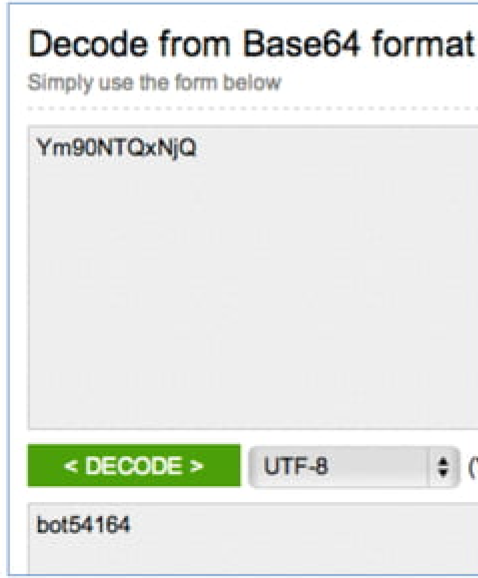
- URL and cookie are Base64-encoded

Example 14-5. Sample malware traffic

```
GET /X29tbVEuYC8=/index.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
Cookie: Ym90NTQxNjQ

GET /c2UsYi1kYWM0cnUjdFlvbiAjb21wbFU0YP==/index.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
Cookie: Ym90NTQxNjQ
```

Cookie: Ym90NTQxNjQ

- This has 11 characters—padding is omitted
- Some Base64 decoders will fail, but this one just automatically adds the missing padding



Finding the Base64 Function

- Look for this "indexing string"

  `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghi`
  `jklmnopqrstuvwxyz0123456789+/`

- Look for a lone padding character (typically =) hard-coded into the encoding function

Decoding the URLs

Figure 14-7. Unsuccessful attempt to decode Base64 string due to nonstandard indexing string

- Custom indexing string

  `aABCDEFGHIJKLMNOPQRSTUVWXYZbcdefghijk`
  `lmnopqrstuvwxyz0123456789+/`

- Look for a lone padding character (typically =) hard-coded into the encoding function

Figure 14-8. Successful decoding of Base64 string using custom indexing string

Common Cryptographic Algorithms

Strong Cryptography

- Strong enough to resist brute-force attacks
  - Ex: SSL, AES, etc.
- Disadvantages of strong encryption
  - Large cryptographic libraries required
  - May make code less portable
  - Standard cryptographic libraries are easily detected
    - Via function imports, function matching, or identification of cryptographic constants
  - Symmetric encryption requires a way to hide the key

Recognizing Strings and Imports

- Strings found in malware encrypted with OpenSSL

```
OpenSSL 1.0.0a
SSLv3 part of OpenSSL 1.0.0a
TLSv1 part of OpenSSL 1.0.0a
SSLv2 part of OpenSSL 1.0.0a
You need to read the OpenSSL FAQ,
http://www.openssl.org/support/faq.html
%s(%d): OpenSSL internal error, assertion failed: %s
AES for x86, CRYPTOGAMS by <appro@openssl.org>
```

Recognizing Strings and Imports

- Microsoft crypto functions usually start with **`Crypt`** or **`CP`** or **`Cert`**

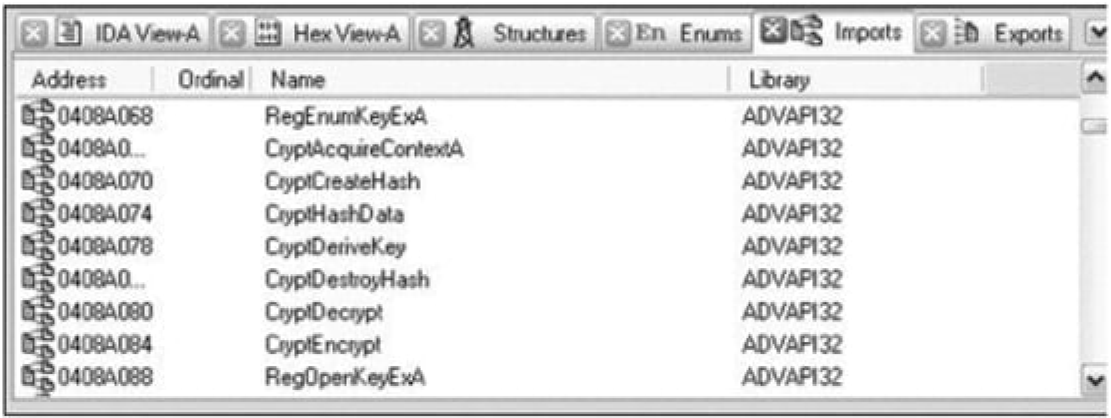


Figure 14-9. IDA Pro imports listing showing cryptographic functions

Searching for Cryptographic Constants

- IDA Pro's FindCrypt2 Plug-in (Link Ch 13c)
  - Finds *magic constants* (binary signatures of crypto routines)
  - Cannot find RC4 or IDEA routines because they don't use a magic constant
  - RC4 is commonly used in malware because it's small and easy to implement

FindCrypt2

- Runs automatically on any new analysis
- Can be run manually from the Plug-In Menu

Figure 14-10. IDA Pro FindCrypt2 output

Krypto ANALyzer (PEiD Plug-in)

- Download from link Ch 13d
- Has wider range of constants than FindCrypt2
  - More false positives
- Also finds Base64 tables and crypto function imports

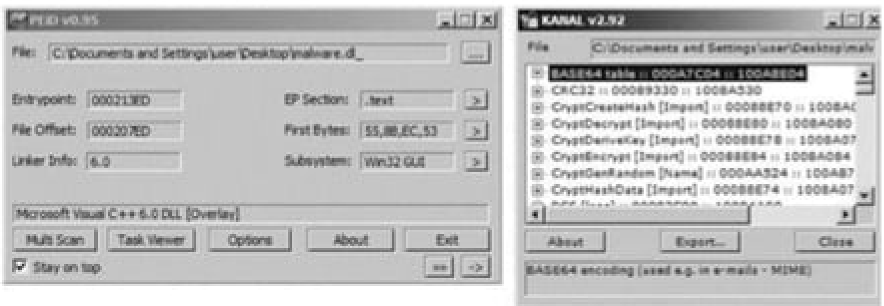


Figure 14-11. PEiD and Krypto ANALyzer (KANAL) output

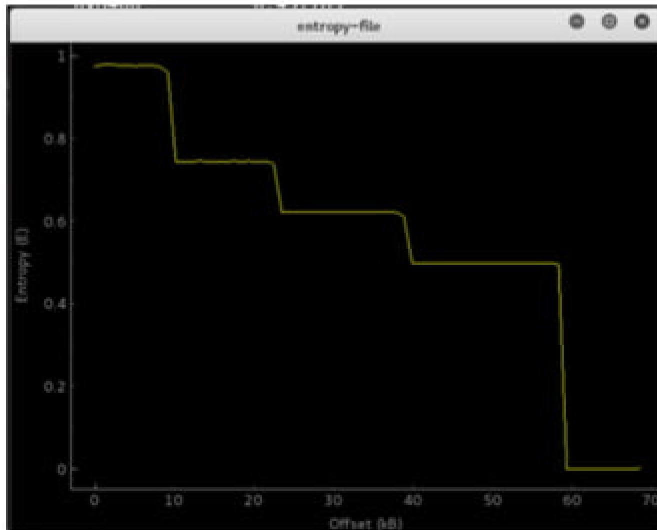
Entropy

- Entropy measures disorder
- To calculate it, just count the number of occurrences of each byte from 0 to 255
  – Calculate Pi = Probability of value i
  – Then sum Pi log(Pi) for I = 0 to 255 (Link 13e)
- If all the bytes are equally likely, the entropy is 8 (maximum disorder)
- If all the bytes are the same, the entropy is zero

Entropy Demo

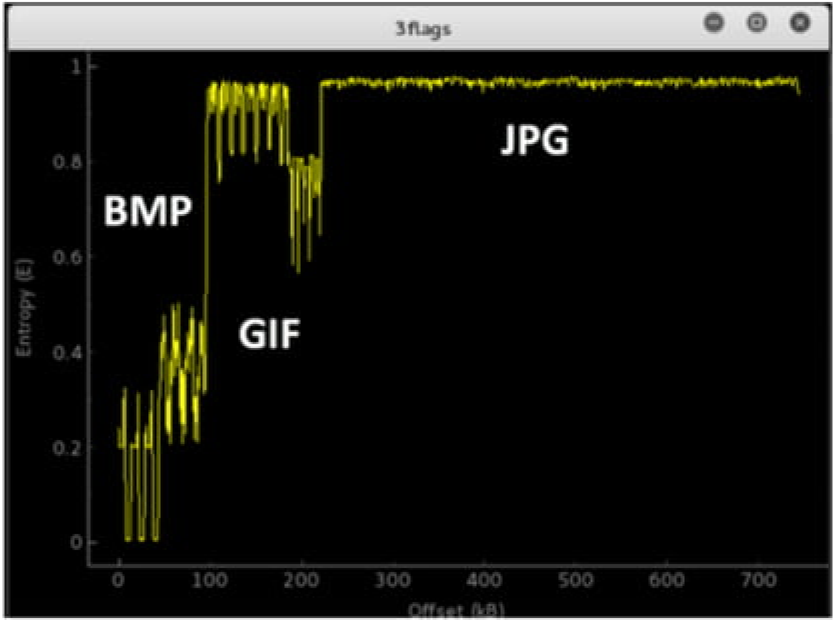
- Put output in a file
- Use bin walk -E to analyze the file
- Multiply vertical axis by 8

```
#!/usr/bin/python
import base64, random

a = ''
for i in range(0, 10000):
  a += chr(random.randint(0,255))

b = base64.b64encode(a)
c = base64.b32encode(a)
d = base64.b16encode(a)
e = 'A' * 10000

print a + b + c + d + e
```



Entropy Demo

- Concatenate three images in different formats



Searching for High-Entropy Content

- IDA Pro Entropy Plugin
- Finds regions of high entropy, indicating encryption (or compression)

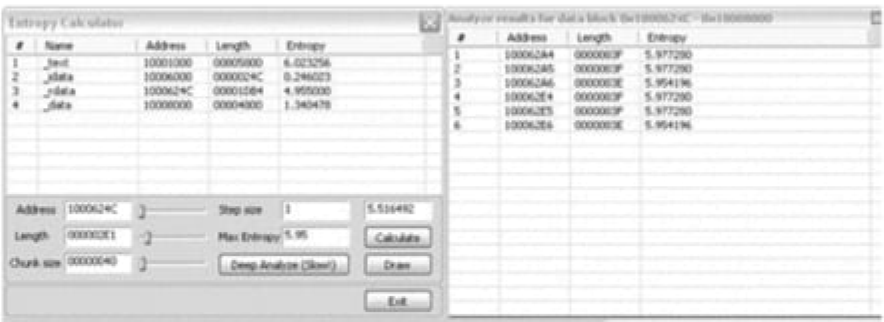


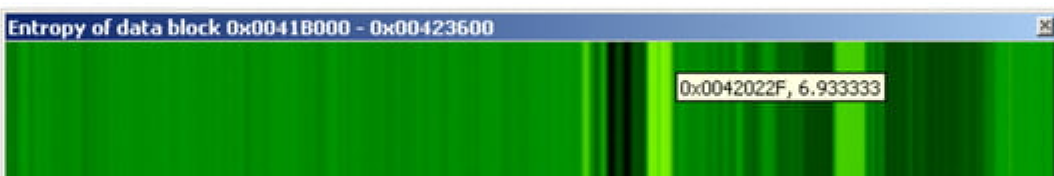
Figure 14-12. IDA Pro Entropy Plugin

Recommended Parameters

- Chunk size: 64 Max. Entropy: 5.95
  - Good for finding many constants,
  - Including Base64-encoding strings (entropy 6)
- Chunk size: 256 Max. Entropy: 7.9
  - Finds very random regions

Entropy Graph

- IDA Pro Entropy Plugin
  - Download from link Ch 13g
  - Use StandAlone version
  - Double-click region, then Calculate, Draw
  - Lighter regions have high entropy
  - Hover over graph to see numerical value



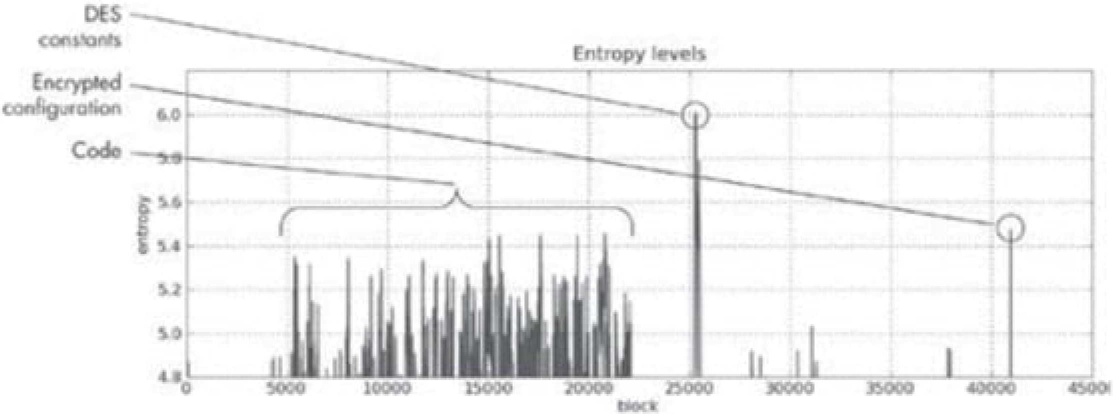


Figure 14-13. Entropy graph for a malicious executable

Custom Encoding

Homegrown Encoding Schemes

- Examples
  - One round of XOR, then Base64
  - Custom algorithm, possibly similar to a published cryptographic algorithm

Identifying Custom Encoding

Example 14-6. First bytes of an encrypted file

```
88 5B D9 02 EB 07 5D 3A 8A 06 1E 67 D2 16 93 7F    .[....]:...g....
43 72 1B A4 BA B9 85 B7 74 1C 6D 03 1E AF 67 AF    Cr......t.m...g.
98 F6 47 36 57 AA 8E C5 1D 70 A5 CB 38 ED 22 19    ..G6W....p..8.".
86 29 98 2D 69 62 9E C0 4B 4F 8B 05 A0 71 08 50    .).-ib..KO...q.P
92 A0 C3 58 4A 48 E4 A3 0A 39 7B 8A 3C 2D 00 9E    ...XJH...9{.<-..
```

- This sample makes a bunch of 700 KB files
- Figure out the encoding from the code
- Find **`CreateFileA`** and **`WriteFileA`**
  - In function **sub_4011A9**
- Uses XOR with a pseudorandom stream

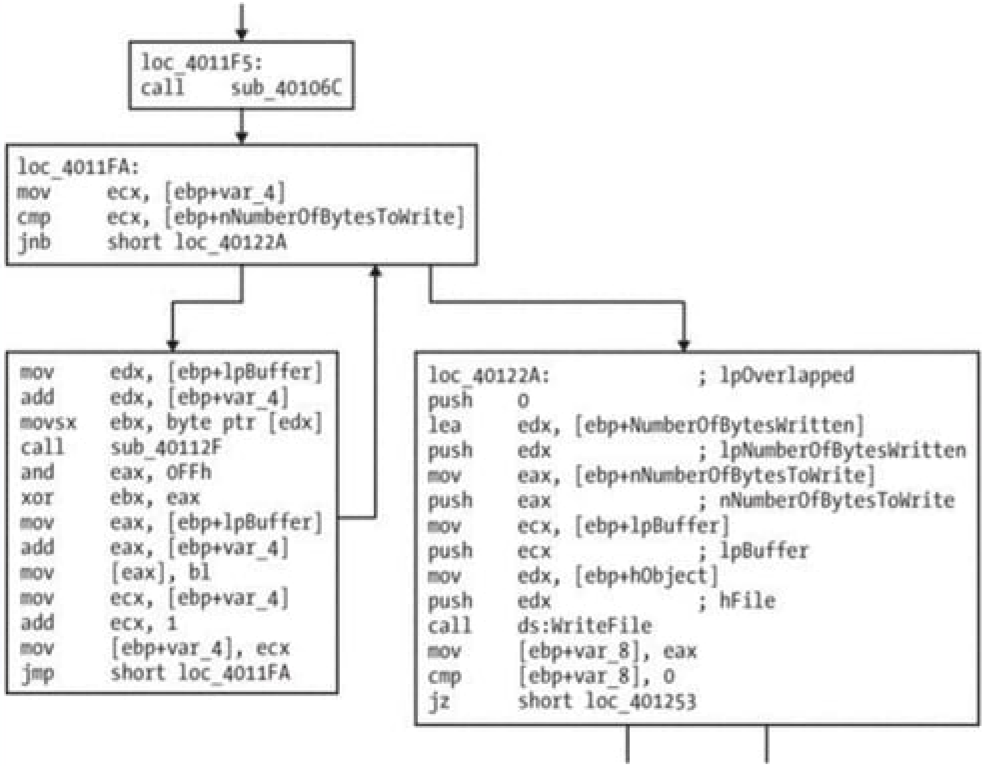


Figure 14-14. Function graph showing an encrypted write

Advantages of Custom Encoding to the Attacker

- Can be small and nonobvious
- Harder to reverse-engineer

Decoding

Two Methods

- Reprogram the functions
- Use the functions in the malware itself

Self-Decoding

- Stop the malware in a debugger with data decoded
- Isolate the decryption function and set a breakpoint directly after it
- BUT sometimes you can't figure out how to stop it with the data you need decoded

Manual Programming of Decoding Functions

- Standard functions may be available

Example 14-7. Sample Python Base64 script

```
import string
import base64

example_string = 'VGhpcyBpcyBhIHRlc3Qgc3RyaW5n'
print base64.decodestring(example_string)
```

Example 14-8. Sample Python NULL-preserving XOR script

```
def null_preserving_xor(input_char,key_char):
    if (input_char == key_char or input_char == chr(0x00)):
        return input_char
    else:
        return chr(ord(input_char) ^ ord(key_char))
```

Example 14-9. Sample Python custom Base64 script

```
import string
import base64

s = ""
custom = "9ZABCDEFGHIJKLMNOPQRSTUVWXYabcdefghijklmnopqrstuvwxyz012345678+/"
Base64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

ciphertext = 'TEgobxZobxZgGFPkb2O='


for ch in ciphertext:
    if (ch in Base64):
        s = s + Base64[string.find(custom,str(ch))]
    elif (ch == '='):
        s += '='

result = base64.decodestring(s)
```

PyCrypto Library

- Good for standard algorithms

Example 14-10. Sample Python DES script

```
from Crypto.Cipher import DES
import sys

obj = DES.new('password',DES.MODE_ECB)
cfile = open('encrypted_file','r')
cbuf = f.read()
print obj.decrypt(cbuf)
```

How to Decrypt Using Malware

1. Set up the malware in a debugger.
2. Prepare the encrypted file for reading and prepare an output file for writing.
3. Allocate memory inside the debugger so that the malware can reference the memory.
4. Load the encrypted file into the allocated memory region.
5. Set up the malware with appropriate variables and arguments for the encryption function.
6. Run the encryption function to perform the encryption.
7. Write the newly decrypted memory region to the output file.

Example 14-12. ImmDbg sample decryption script

```
import immlib

def main ():
    imm = immlib.Debugger()
    cfile = open("C:\\encrypted_file","rb") # Open encrypted file for read
    pfile = open("decrypted_file", "w")     # Open file for plaintext
    buffer = cfile.read()                   # Read encrypted file into buffer
    sz = len(buffer)                        # Get length of buffer
    membuf = imm.remoteVirtualAlloc(sz)     # Allocate memory within debugger
    imm.writeMemory(membuf,buffer)          # Copy into debugged process's memory

    imm.setReg("EIP", 0x004011A9)           # Start of function header
    imm.setBreakpoint(0x004011b7)           # After function header
    imm.Run()                               # Execute function header

    regs = imm.getRegs()
    imm.writeLong(regs["EBP"]+16, sz)       # Set NumberOfBytesToWrite stack variable
    imm.writeLong(regs["EBP"]+8, membuf)    # Set lpBuffer stack variable

    imm.setReg("EIP", 0x004011f5)           # Start of crypto
    imm.setBreakpoint(0x0040122a)           # End of crypto loop
    imm.Run()                               # Execute crypto loop

    output = imm.readMemory(membuf, sz)     # Read answer
    pfile.write(output)                     # Write answer
```