Data Encoding

CYBR473 – Malware and Reverse Engineering (2024/T1)

Lecturers: Arman Khouzani, Alvin Valera

Victoria University of Wellington – School of Engineering and Computer Science

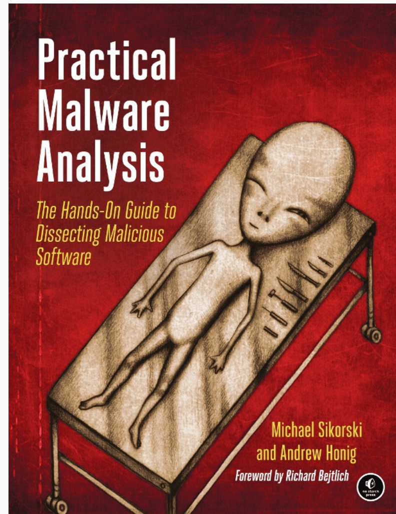
Table of contents

1. The Goal of Analyzing Encoding Algorithms
2. Simple Ciphers
3. Common Cryptographic Algorithms
4. Custom Encoding
5. Decoding

Reading Material

- Part IV: Malware Functionality
  - Ch.13: Data Encoding

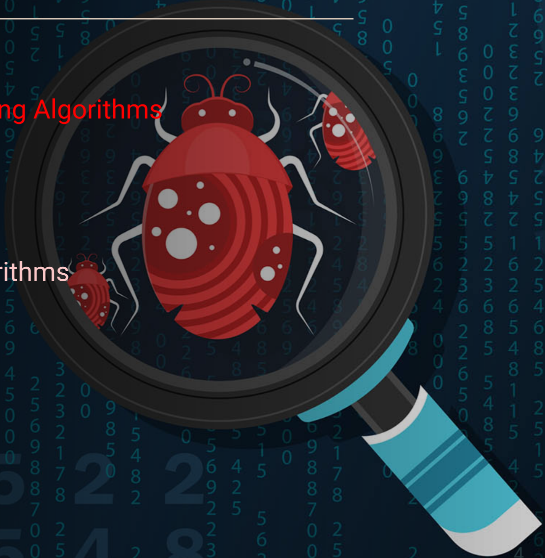
"Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software", Michael Sikorski and Andrew Honig, 2012



Objectives

- Why does malware use outdated encoding/encryption schemes.
- What are some common ones.
- How to identify their use.
- How to identify custom schemes.
- How to write decoders and decryptors.

The Goal of Analyzing Encoding Algorithms



The Goal of Analyzing Encoding Algorithms

Simple Ciphers

Common Cryptographic Algorithms

Custom Encoding

Decoding

The Goal of Analyzing Encoding Algorithms

In the context of malware analysis, the term data encoding refers to all forms of content modification for the purpose of hiding intent. E.g.:

- To hide configuration information, such as a command-and-control domain
- To save information to a staging file before stealing it
- To store strings used by the malware and decode them just before they are needed
- To disguise the malware as a legitimate tool

So their main purpose is *obfuscation* (to avoid detection or frustrating the analyst!), not encryption for confidentiality.

Simple Ciphers

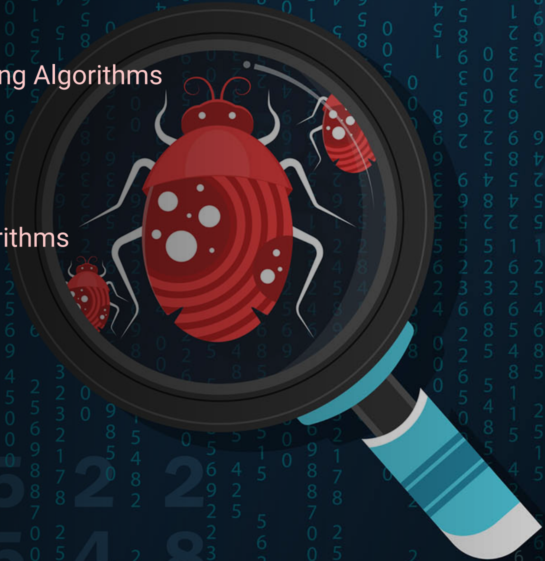
The Goal of Analyzing Encoding Algorithms

Simple Ciphers

Common Cryptographic Algorithms

Custom Encoding

Decoding



Simple Ciphers: Why are they used at all

Simple ciphers are often disparaged for being unsophisticated, but they offer many advantages for malware:

- They are small enough to be used in space-constrained environments such as exploit shellcode.
- They are less obvious than more complex ciphers.
- They have low overhead and thus little impact on performance.

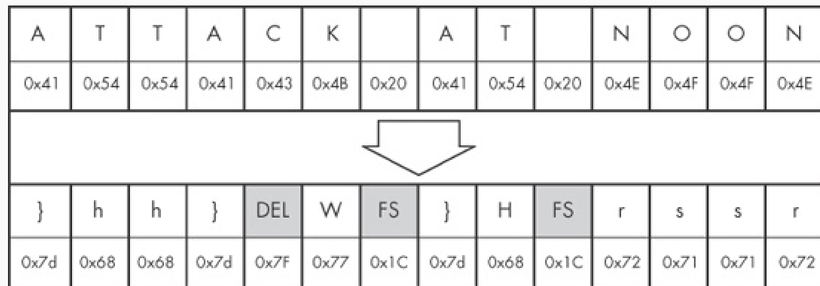
Their goal is not to be immune to detection; rather an easy way to prevent basic analysis from identifying their activities.

Simpler Ciphers: Caesar Cipher

```
THIS MALWARE IS WRITTEN BY JULIUS CAESAR
AOPZ THSDHYL PZ DYPAALU IF QBSPBZ JHLZHY
```

Simple Ciphers: XOR

Simple XOR cipher, aka *single-byte XOR encoding*: Each byte of plaintext is `XOR`ed with a constant byte value.



An example of single-byte XOR encoding. Can you tell what the XOR key is?

(**Q:** what are some good properties of XOR function for this purpose?)

Brute-Forcing XOR Encoding

```
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

A weakness of the single-byte XOR encoding: If the encoded content has a large number of NULL bytes, the“key” becomes obvious.

Original XOR

```
buf[i] ^= key;
```

NULL-preserving XOR

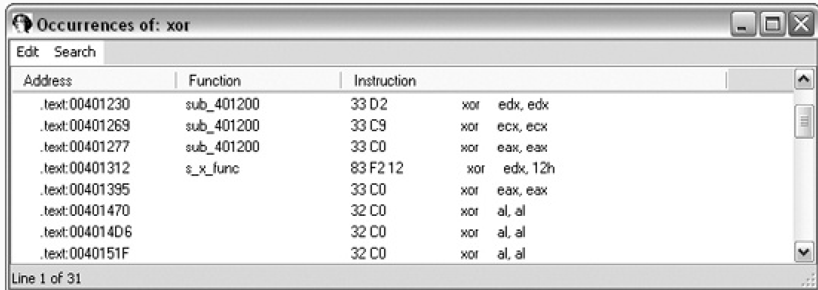
```
if (buf[i] != 0 && buf[i] != key)
    buf[i] ^= key;
```

Brute-Forcing 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

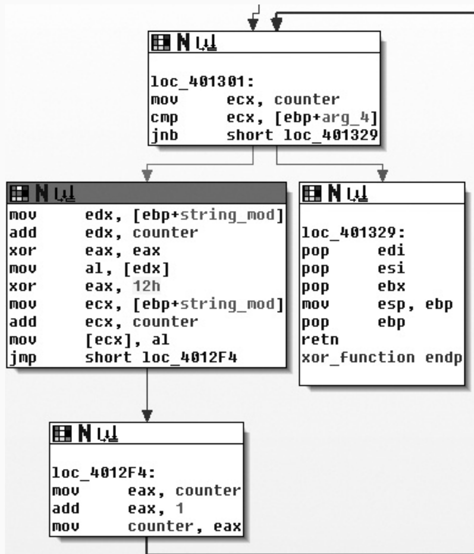
- In disassembly, XOR loops can be identified by small loops with an XOR instruction in the middle of a loop.
- The easiest way to find an XOR loop in IDA Pro is to search for all instances of the XOR instruction.



Address	Function	Instruction		
.text:00401230	sub_401200	33 D2	xor	edx, edx
.text:00401269	sub_401200	33 C9	xor	ecx, ecx
.text:00401277	sub_401200	33 C0	xor	eax, eax
.text:00401312	s_x_func	83 F2 12	xor	edx, 12h
.text:00401395		33 C0	xor	eax, eax
.text:00401470		32 C0	xor	al, al
.text:004014D6		32 C0	xor	al, al
.text:0040151F		32 C0	xor	al, al

Searching for XOR in IDA Pro.

Identifying XOR Loops in IDA Pro



Graphical view of single-byte XOR loop

Other Simple Encoding Schemes

`ADD, SUB`

`ADD` and `SUB` are not reversible, so they need to be used in tandem (one to encode and the other to decode).

`ROL, ROR`

Rotate the bits within a byte left, right (one to encode, one to decode)

Multibyte

Using longer than 1 byte key (often 4 or 8), typically `XOR`.

Chained or loopback

Uses the content itself as part of the key, e.g., the original key is applied at one side of the plaintext (start or end), and the encoded output is used as the key for the next block.

Base64

Base64 encoding: encoding data in base 64!

Symbols needed for each base:

- Base2: `0,1`
- Base8: `0,1,2,3,4,5,6,7`
- Base10: `0,1,2,3,4,5,6,7,8,9`
- Base16: `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`
- Base64: …?

Base64

Base64 encoding: encoding data in base 64!

Symbols needed for each base:

- Base2: `0,1`
- Base8: `0,1,2,3,4,5,6,7`
- Base10: `0,1,2,3,4,5,6,7,8,9`
- Base16: `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`
- Base64: `A-Z`, `a-z`, `0-9`, `+`, `/`

Base64

Base64 encoding: encoding data in base 64!

Symbols needed for each base:

- Base2: `0,1`
- Base8: `0,1,2,3,4,5,6,7`
- Base10: `0,1,2,3,4,5,6,7,8,9`
- Base16: `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`
- Base64: `A−Z`, `a−z`, `0−9`, `+`, /
    - used to represent binary data in an ASCII string format.
    - originally developed to encode email attachments for transmission (Multipurpose Internet Mail Extensions – MIME – standard), it is now widely used for HTTP and XML.
    - we also needed an additional character to indicate padding, often =

Base64

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		=									

Base64

Source	Text (ASCII)	M								a								n							
	Octets	77 (0x4d)								97 (0x61)								110 (0x6e)							
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Base64 encoded	Sextets	19						22						5						46					
	Character	T						W						F						u					
	Octets	84 (0x54)						87 (0x57)						70 (0x46)						117 (0x75)					

Source	Text (ASCII)	M								a															
	Octets	77 (0x4d)								97 (0x61)															
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	0						
Base64 encoded	Sextets	19						22						4						Padding					
	Character	T						W						E						=					
	Octets	84 (0x54)						87 (0x57)						69 (0x45)						61 (0x3D)					

Source	Text (ASCII)	M																							
	Octets	77 (0x4d)																							
Bits		0	1	0	0	1	1	0	1	0	0	0	0												
Base64 encoded	Sextets	19						16						Padding						Padding					
	Character	T						Q						=						=					
	Octets	84 (0x54)						81 (0x51)						61 (0x3D)						61 (0x3D)					

Base64

```
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

Identifying and Decoding Base64

```
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
```

Common Cryptographic Algorithms

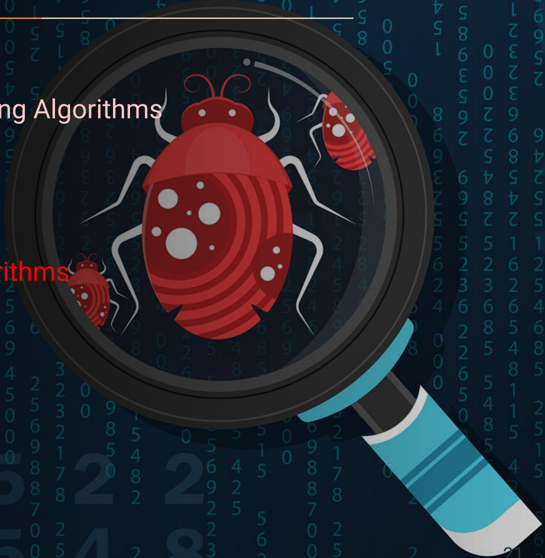
The Goal of Analyzing Encoding Algorithms

Simple Ciphers

Common Cryptographic Algorithms

Custom Encoding

Decoding



Common Cryptographic Algorithms

The simple cipher schemes are trivially susceptible to brute-force. Their main purpose is to obscure.

Why then, does malware not always take advantage of serious cryptography for hiding its sensitive information?

- ▷ simple cipher schemes are easy and often sufficient.
- ▷ Cryptographic libraries can be large
- ▷ Linking to code that exists on the host may reduce portability.
- ▷ Standard cryptographic libraries are easily detected (via imports, function matching, or identification of cryptographic constants).
- ▷ Symmetric encryption algorithms need to worry about how to hide the key.

Identifying standard cryptography: Strings and Imports

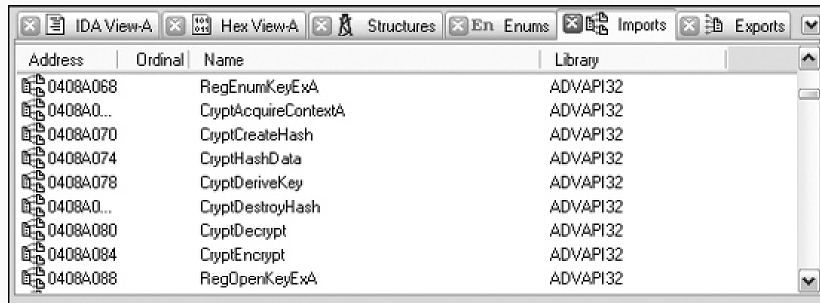
One way to identify standard cryptographic algorithms is by recognizing strings that refer to the use of cryptography. This can occur when cryptographic libraries such as OpenSSL are statically compiled into malware. Example:

```
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>
```

Identifying standard cryptography: Strings and Imports

Another way to look for standard cryptography is to identify imports that reference cryptographic functions, e.g. that provide services related to hashing, key generation, and encryption.

Most of the Microsoft functions that pertain to cryptography start with **`Crypt`**, **`CP`** (for Cryptographic Provider), or **`Cert`**.



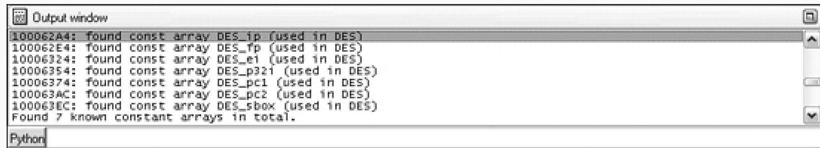
Address	Ordinal	Name	Library
0408A068		RegEnumKeyExA	ADVAPI32
0408A0...		CryptAcquireContextA	ADVAPI32
0408A070		CryptCreateHash	ADVAPI32
0408A074		CryptHashData	ADVAPI32
0408A078		CryptDeriveKey	ADVAPI32
0408A0...		CryptDestroyHash	ADVAPI32
0408A080		CryptDecrypt	ADVAPI32
0408A084		CryptEncrypt	ADVAPI32
0408A088		RegOpenKeyExA	ADVAPI32

IDA Pro imports listing showing cryptographic functions.

Identifying standard cryptography: Cryptographic Constants

Another method is via tools that search for commonly used cryptographic constants. This works because most cryptographic algorithms employ some type of magic constant (some fixed string of bits that is associated with the essential structure of the algorithm)

Example tools: IDA Pro's **`FindCrypt2`** and **`Krypto ANALyzer`**.

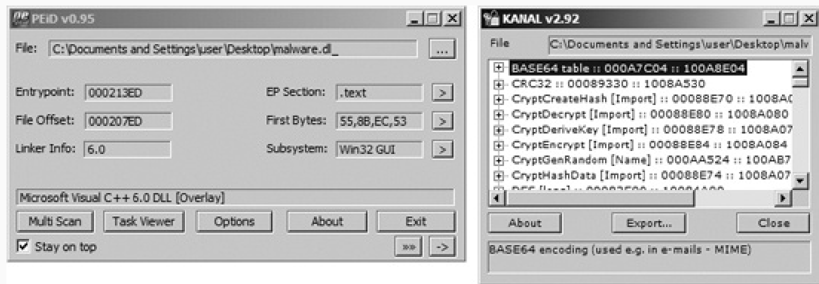


Example output of the IDA Pro's plugin: **`FindCrypt2`**.

NOTE: Some cryptographic algorithms, like **`IDEA`** and **`RC4`**, build their structures on the fly, and thus are not identified. **`RC4`**, because it is also small and easy to implement, is often employed by malware.

Identifying standard cryptography: Cryptographic Constants

Another tool that uses the same principles is the **Krypto ANALyzer (KANAL)**. **KANAL** is a plug-in for **PEiD**. It also recognizes Base64 tables and cryptography-related function imports.

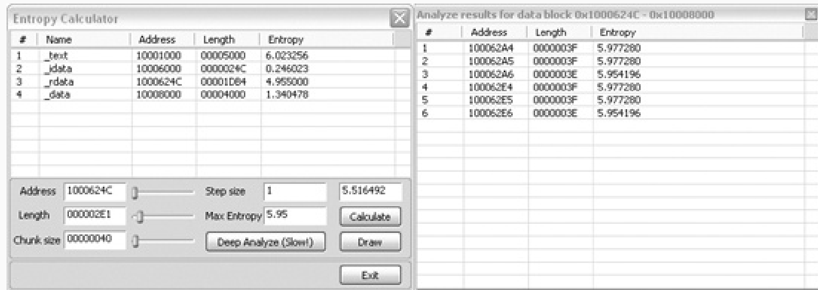


Example **PEiD** and **Krypto ANALyzer (KANAL)** output, finding a **Base64** table, a **CRC32** constant, and several **Crypt...** import functions in a malware.

Identifying standard cryptography: High-Entropy Content

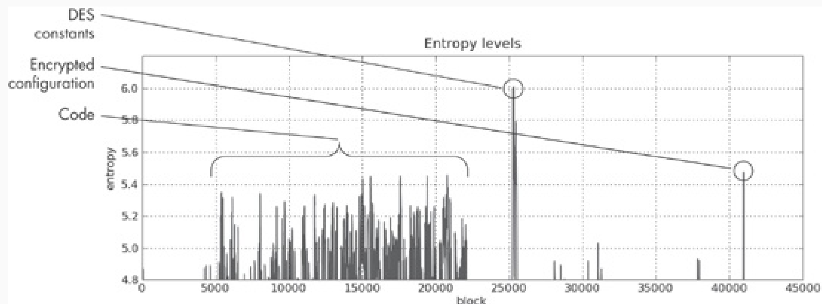
Another way to identify the use of cryptography is to search for high-entropy content. In addition to potentially highlighting cryptographic constants or cryptographic keys, this technique can also identify encrypted content itself.

The **IDA Entropy Plugin** is one tool that implements this technique:



IDA Pro Entropy Plugin

Identifying standard cryptography: High-Entropy Content



Entropy graph for a malicious executable.

Custom Encoding

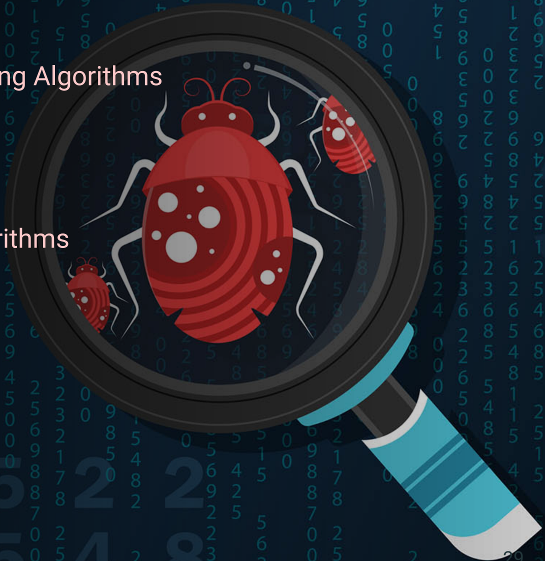
The Goal of Analyzing Encoding Algorithms

Simple Ciphers

Common Cryptographic Algorithms

Custom Encoding

Decoding



Identifying Custom Encoding

Malware often uses homegrown encoding schemes.

One such scheme is to layer multiple simple encoding methods. For example, malware may perform one round of `XOR` encryption and then perform `Base64` encoding on the result.

Another scheme is to simply develop a custom algorithm, possibly with similarities to a standard published cryptographic algorithm.

Identifying Custom Encoding

Finding the encoding algorithm the hard way entails tracing the thread of execution from the suspicious input or output.

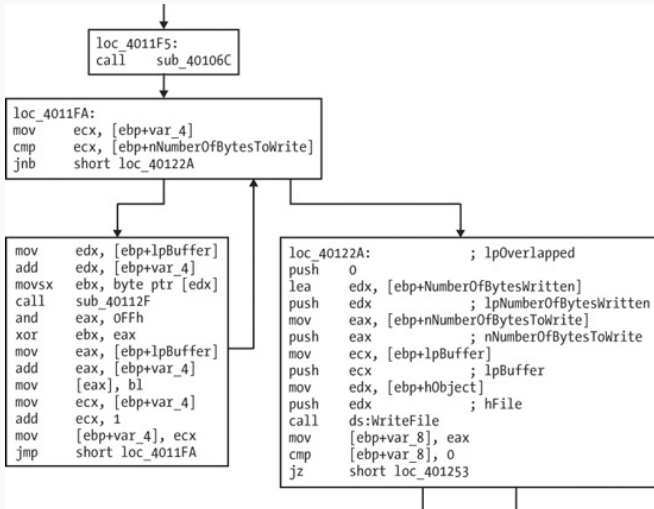
Inputs and outputs can be treated as generic categories. No matter whether the malware sends a network packet, writes to a file, or writes to standard output, those are all outputs.

If outputs are suspected of containing encoded data, then the encoding function will occur prior to the output.

Conversely, decoding will occur not far after an input. Follow the execution path forward.

Output functions are similar, except that the tracing must be done opposite the flow of execution.

Identifying Custom Encoding



Example function graph showing an encrypted write.

Advantages of Custom Encoding to the Attacker

For the attacker, custom-encoding methods have their advantages:

- they can retain the characteristics of simple encoding schemes (small size and non-obvious use of encryption).
- making the job of the reverse engineer more difficult.

With many types of standard cryptography, if the cryptographic algorithm is identified and the key found, it is fairly easy to write a decryptor using standard libraries.

With custom encoding, attackers can create any encoding scheme they want, which may or may not use an explicit key (the key can be embedded (and obscured) within the code itself).

Even if the attacker uses a key and the key is found, it is unlikely that a freely available library will be available to assist with the decryption.

Decoding

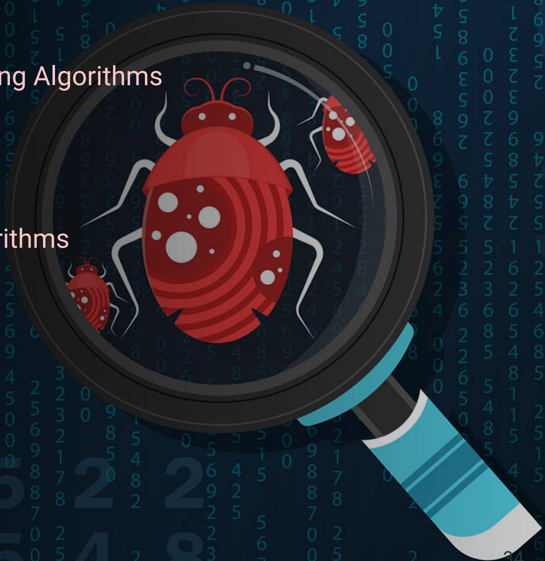
The Goal of Analyzing Encoding Algorithms

Simple Ciphers

Common Cryptographic Algorithms

Custom Encoding

Decoding



Decoding

Typically we want to decode the encoded content. Two ways:

- ▷ Reprogram the functions (build a decoder).
- ▷ Use the functions as they exist in the malware itself.

Self-Decoding

self-decoding: (in a debugger) let the program itself perform the decryption in the course of its normal activities

cheap and effective, but drawbacks:

- you must isolate the decryption function and set a breakpoint directly after the decryption routine.
- if the malware doesn't happen to decrypt the information you are interested in (or you cannot figure out how to coax the malware into doing so), you are out of luck.

Manual Programming of Decoding Functions

- For simple ciphers and encoding methods, you can often use the standard functions available within a programming language. For example, **`base64.b64decode(str)`** in python for Base64 decoding.
- For simple encoding methods that lack standard functions, such as XOR encoding or Base64 encoding that uses a modified alphabet, often the easiest course of action is to just program or script the encoding function in the language of your choice.

```
from Crypto.Cipher import DES
des = DES.new(key, DES.MODE_ECB)
cfile = open('encrypted_file','r')
cbuf = cfile.read()
print obj.decrypt(cbuf)
cfile.close()
```

Using Instrumentation for Generic Decryption

Instrumentation: executing a programme using directed input in order to diagnose errors and to write trace information.

It is useful when dealing with cases where the cryptography is too complex to emulate and is also nonstandard.

Once encoding or decoding routines are isolated and the parameters are understood, it is possible to exploit malware to decode arbitrary content using instrumentation, effectively using it against itself.

```
004011A9         push    ebp
004011AA         mov     ebp, esp
004011AC         sub     esp, 14h
004011AF         push    ebx
004011B0         mov     [ebp+counter], 0
004011B7         mov     [ebp+NumberOfBytesWritten], 0
...
004011F5 loc_4011F5:  ; CODE XREF: encrypted_Write+46j
004011F5         call    encrypt_Init
004011FA
```

Using Instrumentation for Generic Decryption

```
004011FA loc_4011FA:      ; CODE XREF: encrypted_Write+7Fj
004011FA         mov      ecx, [ebp+counter]
004011FD         cmp      ecx, [ebp+nNumberOfBytesToWrite]
00401200         jnb      short loc_40122A
00401202         mov      edx, [ebp+lpBuffer]
00401205         add      edx, [ebp+counter]
00401208         movsx    ebx, byte ptr [edx]
0040120B         call     encrypt_Byte
00401210         and      eax, 0FFh
00401215         xor      ebx, eax
00401217         mov      eax, [ebp+lpBuffer]
0040121A         add      eax, [ebp+counter]
0040121D         mov      [eax], bl
0040121F         mov      ecx, [ebp+counter]
00401222         add      ecx, 1
00401225         mov      [ebp+counter], ecx
00401228         jmp      short loc_4011FA
0040122A
0040122A loc_40122A:      ; CODE XREF: encrypted_Write+57j
0040122A         push     0        ; lpOverlapped
0040122C         lea      edx, [ebp+NumberOfBytesWritten]
```

Using Instrumentation for Generic Decryption

We can find out a couple of key information through analysis:

- The function **`sub_40112F`** initializes the encryption, and is the start of the encryption routine, called at address **`0x4011F5`**. In the listing, this function is labeled **`encrypt_Init`**.
- The encryption has completed when we reach at **`0x40122A`**.
- We know several of the variables and arguments that are used in the encryption function. These include the counter and two arguments: the buffer (**`lpBuffer`**) to be encrypted or decrypted and the length (**`nNumberOfBytesToWrite`**) of the buffer.

Our high-level goal is to instrument the malware so that it takes the encrypted file, the malware itself, and runs it through the same routine it used for encryption. (We are assuming based on the use of **`XOR`** that the function is reversible.)

Using Instrumentation for Generic Decryption

This high-level goal can be broken down into a series of tasks:

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

The following **Immunity Debugger** python script does these tasks.

Using Instrumentation for Generic Decryption

```
import immlib
def main ():
    imm = immlib.Debugger()
    cfile = open("C:\\enc_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 variab
    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
```

Using Instrumentation for Generic Decryption

- **`immlib.Debugger`** provides programmatic access to debugger.
- Note the **`rb`** option on the python **`open`** command that ensures that data is interpreted correctly as “binary” when reading.
- **`imm.remoteVirtualAlloc`** allocates memory within the malware process space inside the debugger. This is the memory that can be directly referenced by the malware.
- **`imm.getRegs`** gets the current register values so that **`EBP`** can be used to locate the two arguments: the memory buffer to be decrypted and its size. **`imm.writeLong`** sets these arguments.

Using Instrumentation for Generic Decryption

The actual running of the code is done in two stages:

- The initial portion of code run is the start of the function, which sets up the stack frame and sets the counter to zero. This is from **`0x004011A9`** (where **`EIP`** is set) until **`0x004011b7`** (where a breakpoint, set by **`imm.setBreakpoint`**, stops execution).
- The second part is the actual encryption loop, from **`0x004011f5`** (where **`EIP`** is set), through the loop one time for each byte decrypted, until the loop is exited and **`0x0040122a`** is reached (where the second breakpoint stops execution).
  - Finally, the same buffer is read out of the process memory (using **`imm.readMemory`**) and output to a file (using **`pfile.write`**).

Using Instrumentation for Generic Decryption

In this example, the encryption function stood alone: It didn't have any dependencies and was fairly straightforward.

However, Some encoding functions require initialization, possibly with a key, which may not even reside in the malware, but acquired from an outside source over the network.

In order to support decoding in these cases, it is necessary to first have the malware properly prepared:

- it may merely mean that the malware needs to start up in the normal fashion, if, e.g., it uses an embedded password as a key.
- in other cases, it may be necessary to customize the external environment. For example, it may be necessary either to script the key-setup algorithm with the appropriate key material or to simulate the server sending the key.

Next: Malware-Focused Network Signatures