CSec15233
Malicious Software Analysis

Malware Encoding

Qasem Abu Al-Haija

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 and can 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 alphabet**

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
```

- **Example**

```
ATTACK AT NOON
DWWDFN DW QRRQ
```

XOR Cipher

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 key at a time
- Example: Encode HI with a key of 0x3C

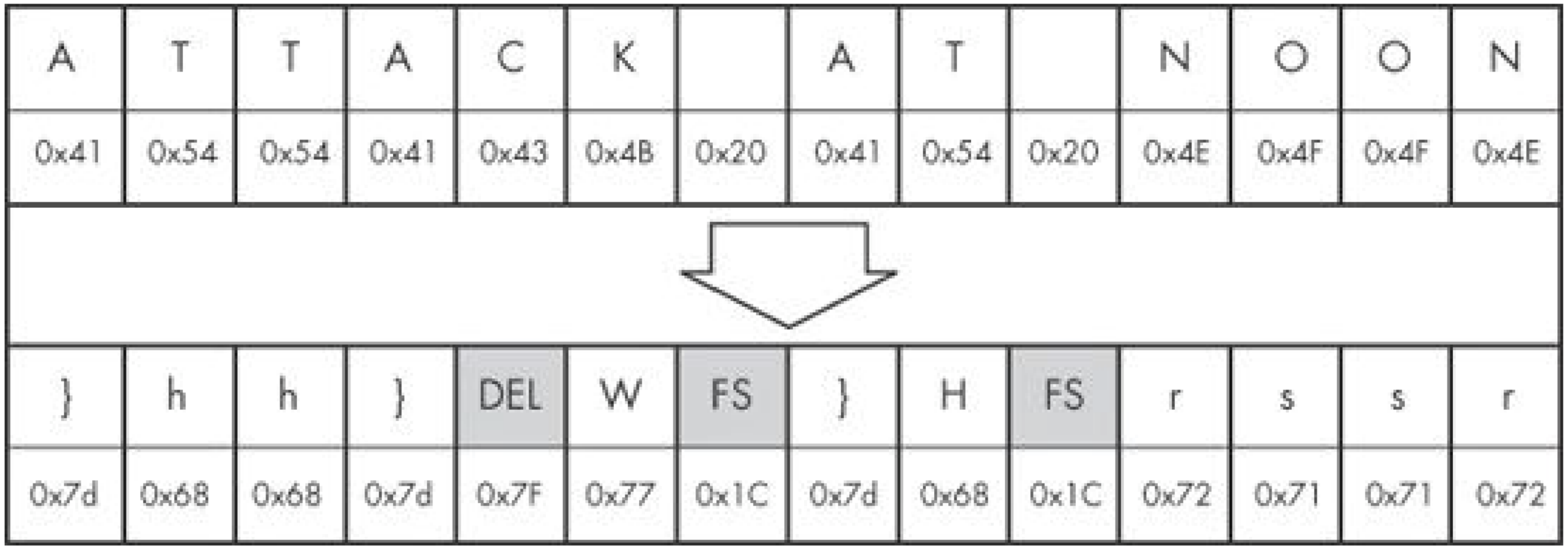
HI = 0x48 0x49 (ASCII encoding)

Data:	0100 1000 0100 1001
Key:	0011 1100 0011 1100
Result:	0111 0100 0111 0101

$\rightarrow$ 0x74 0x75 $\rightarrow$ tu

XOR Cipher

Example



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
Result:	0111 0100 0111 0101

- 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
  - PE files begin with MZ

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

```
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

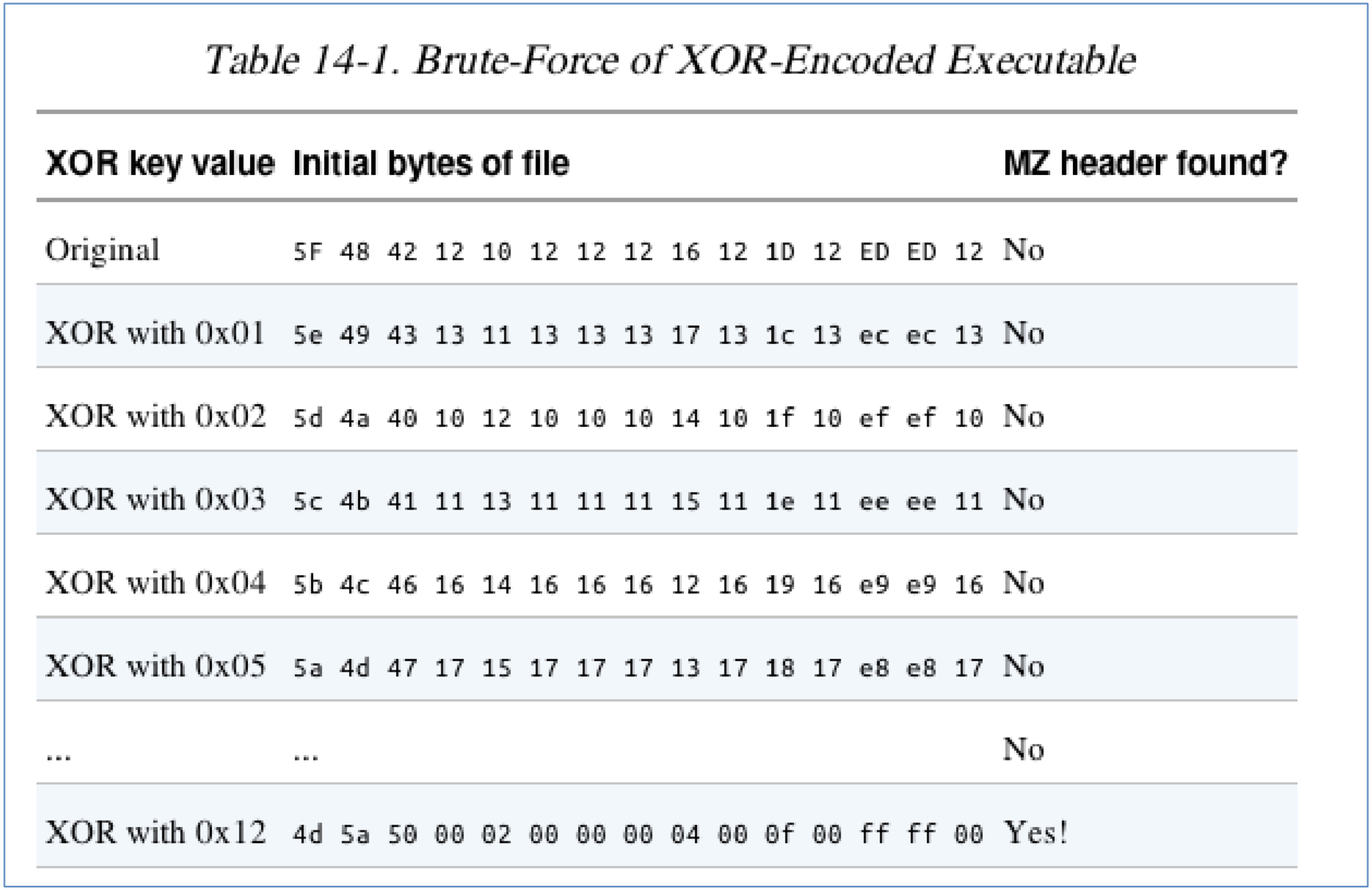


Table 14-1. Brute-Force of XOR-Encoded Executable

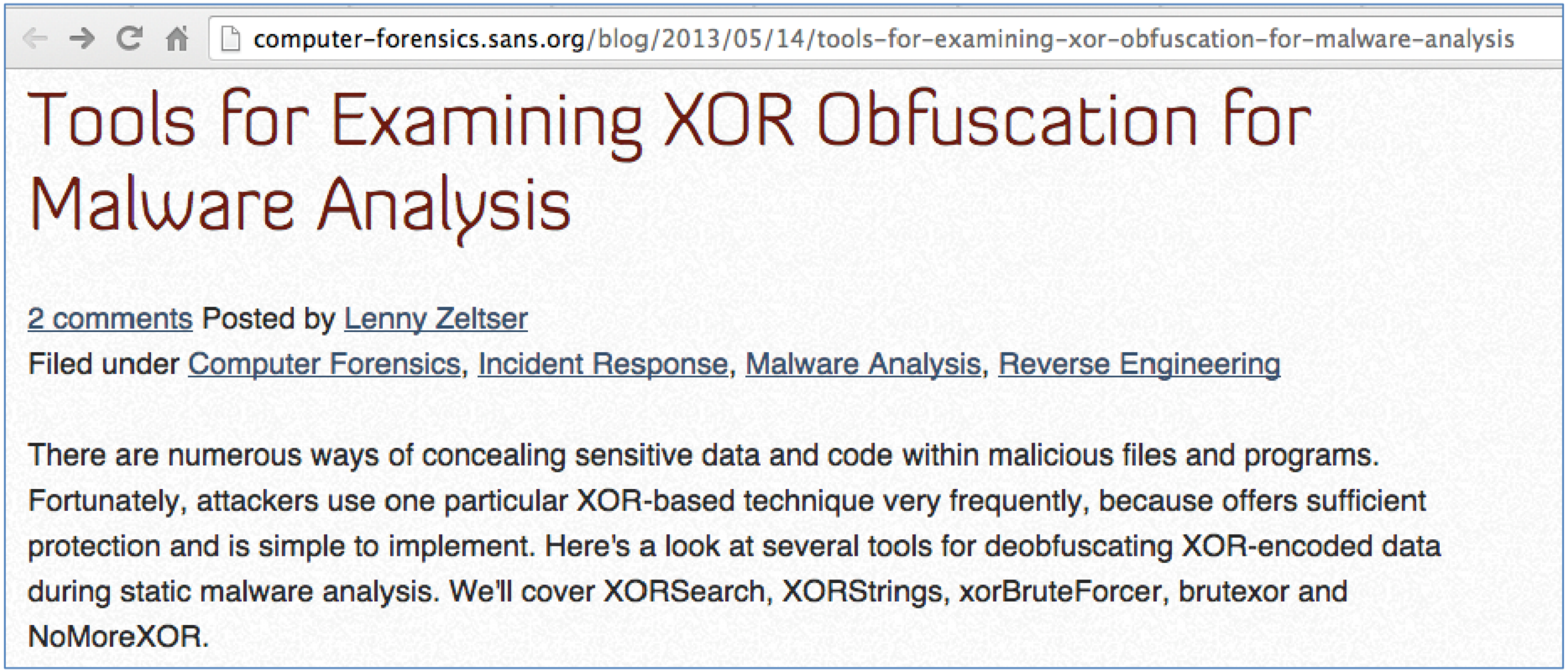
XOR key value	Initial bytes of file	MZ header found?
Original	`5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12`	No
XOR with 0x01	`5e 49 43 13 11 13 13 13 17 13 1c 13 ec ec 13`	No
XOR with 0x02	`5d 4a 40 10 12 10 10 10 14 10 1f 10 ef ef 10`	No
XOR with 0x03	`5c 4b 41 11 13 11 11 11 15 11 1e 11 ee ee 11`	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!

After Decoding

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
```

Tools to de-obfuscate XOR-Encoded Data


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

- Proactive brute-forcing
  - Look for a common string, like "This Program"
  - Search files for encoded strings

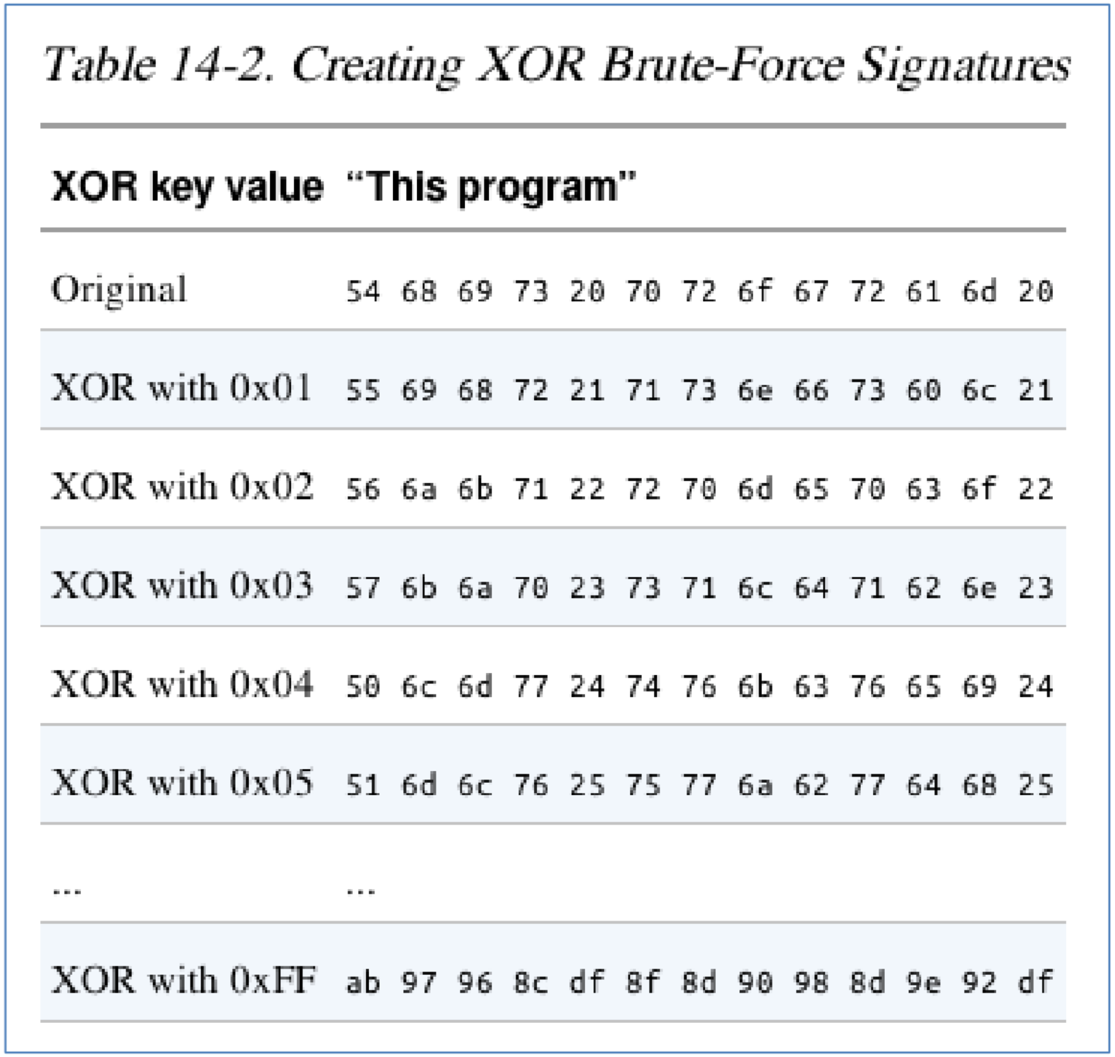


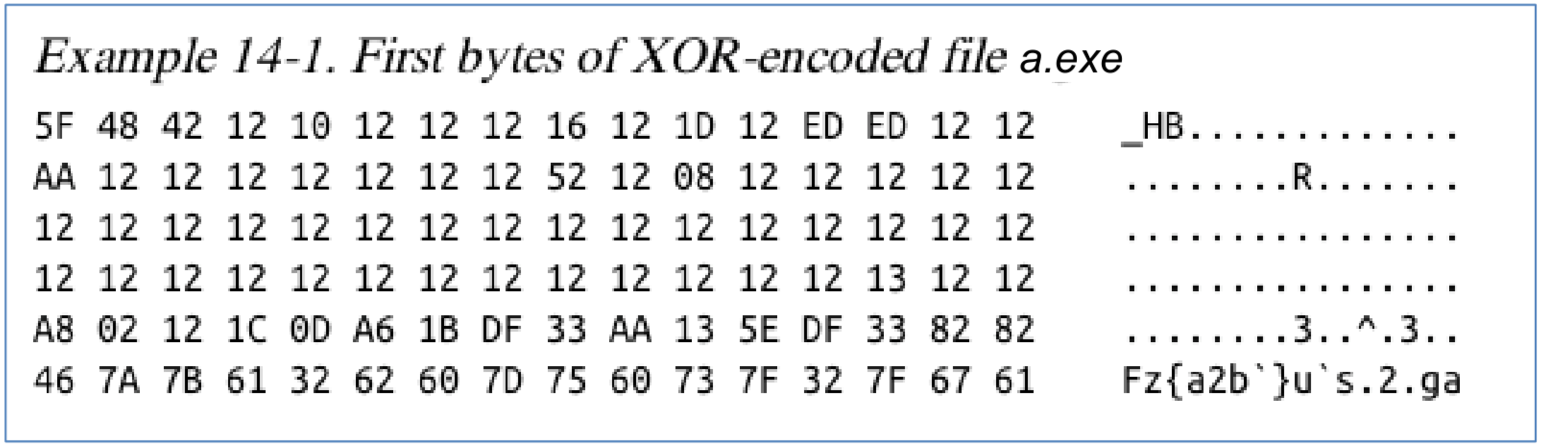
Table 14-2. Creating XOR Brute-Force Signatures

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.exe

```
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

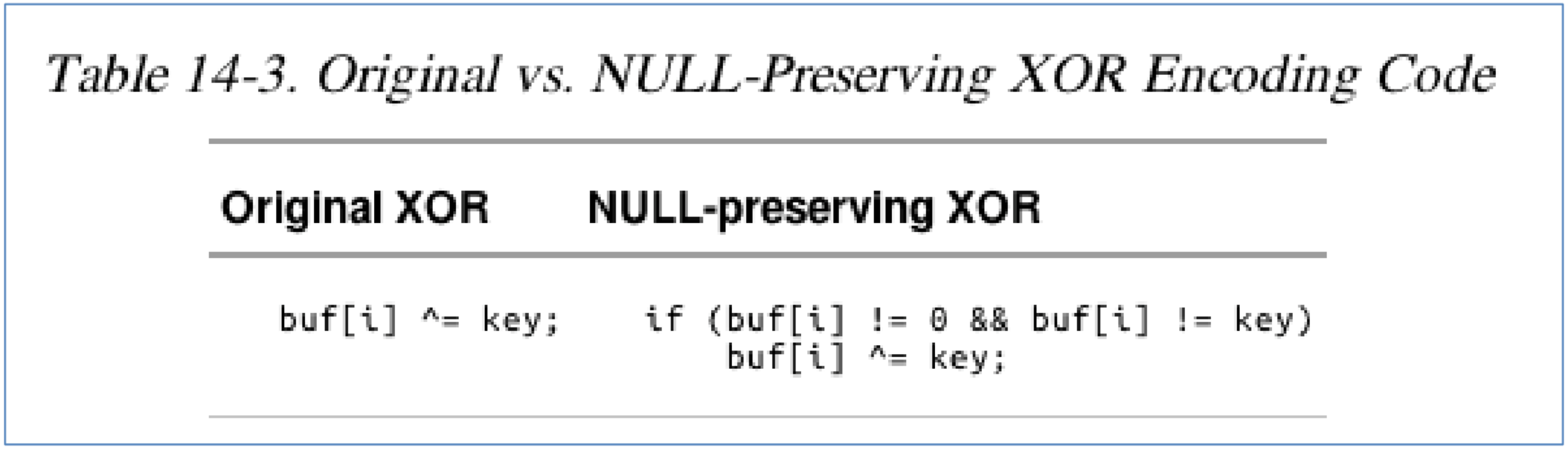
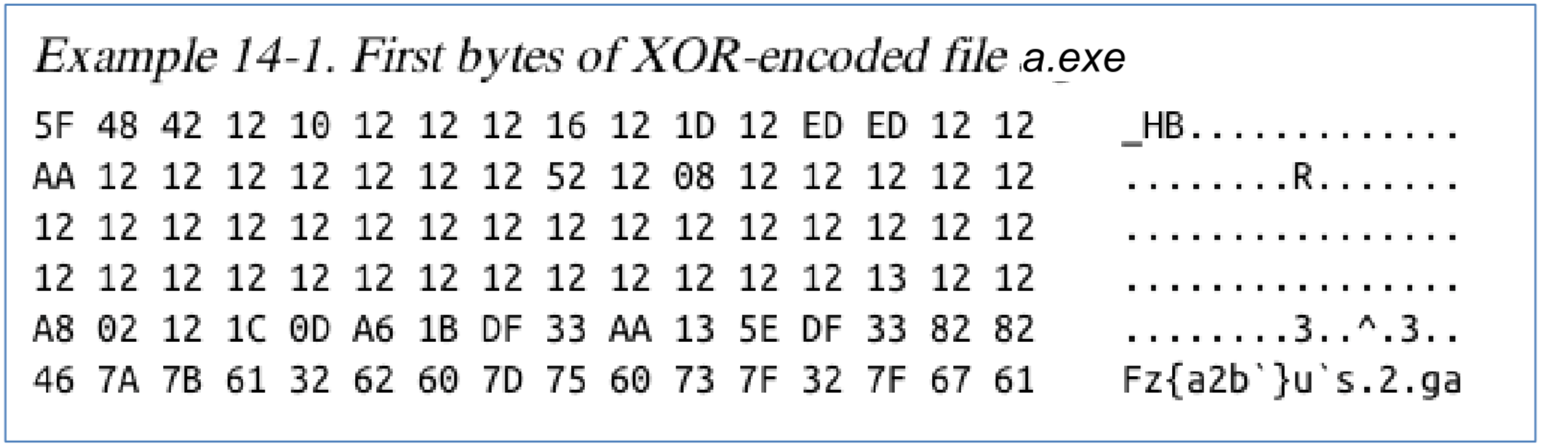


Table 14-3. Original vs. NULL-Preserving XOR Encoding Code

Original XOR	NULL-preserving XOR

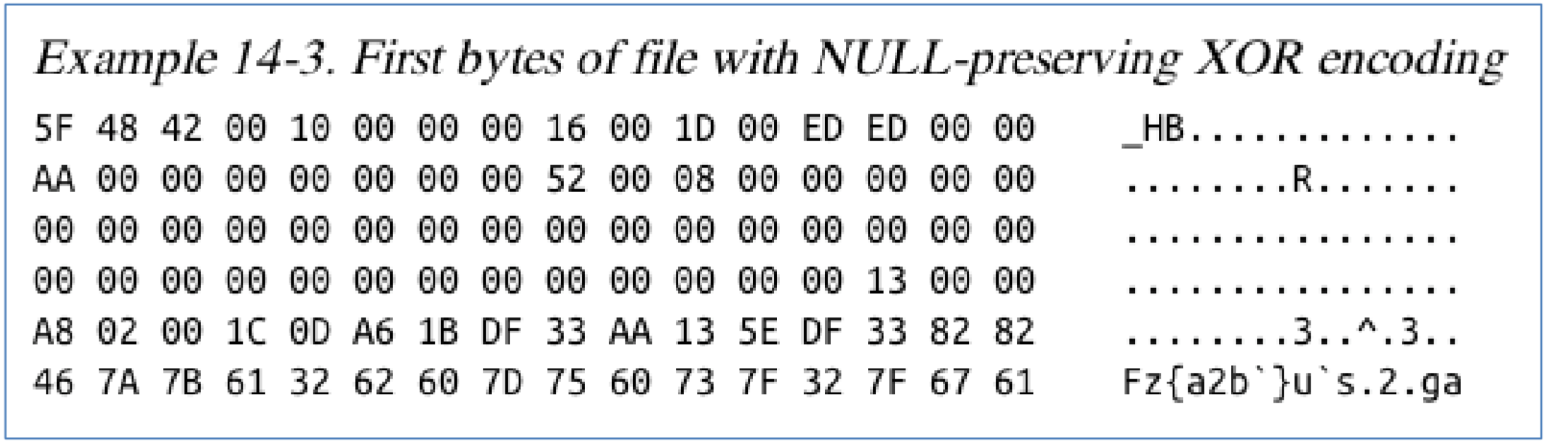
`buf[i] ^= key;`	`if (buf[i] != 0 && buf[i] != key)`
	`buf[i] ^= key;`

Comparison



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

```
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
```

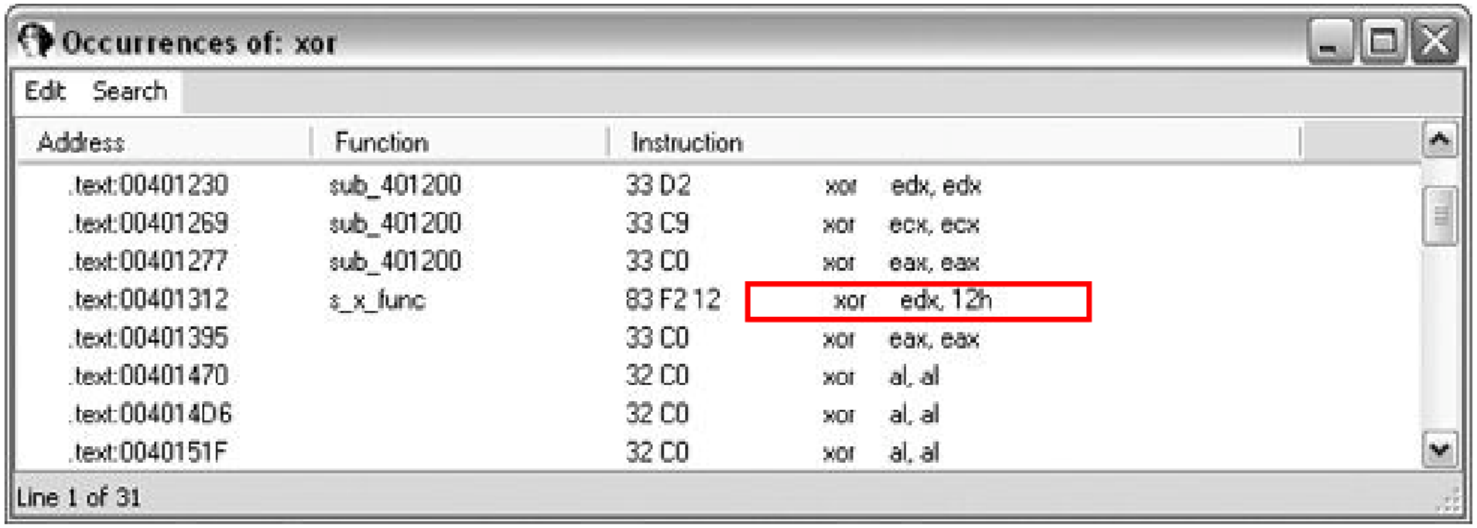


Example 14-3. First bytes of file with NULL-preserving XOR encoding

```
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**

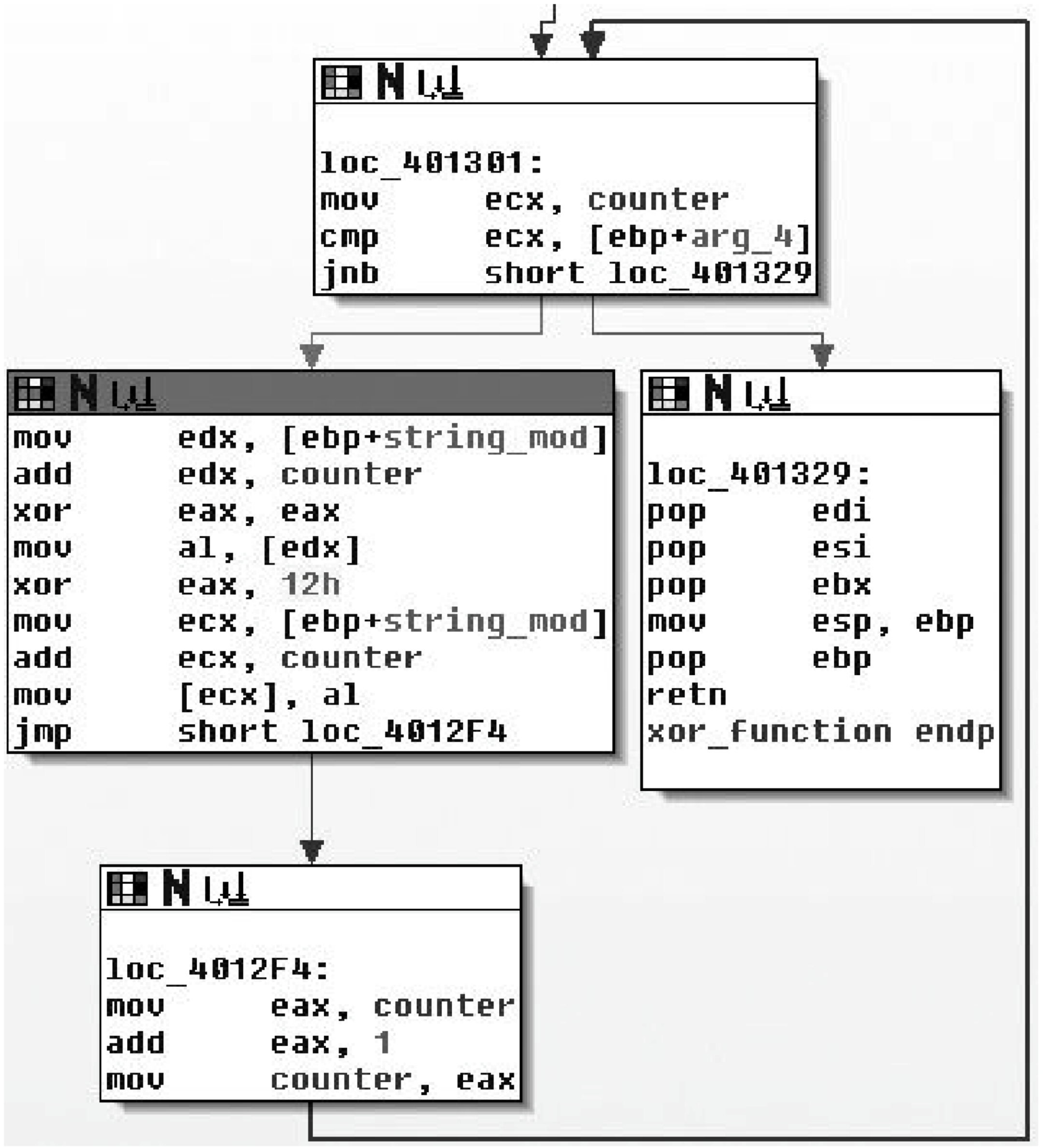


Address	Function	Instruction		
.text:00401230	sub_401200	33 D2	xor	edx, edx
.text:00401269	sub_401200	33 C9	xor	ecx, ecx
.text:00401277	sub_401200	33 C0	xor	eax, eax
.text:00401312	s_x_func	83 F2 12	xor	edx, 12h
.text:00401395		33 C0	xor	eax, eax
.text:00401470		32 C0	xor	al, al
.text:004014D6		32 C0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

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 the key is the constant
- XOR a reg/mem with a different reg/mem
  - May be an encoding loop, keyless obvious



```
loc_401301:
mov      ecx, counter
cmp      ecx, [ebp+arg_4]
jnb      short loc_401329
```

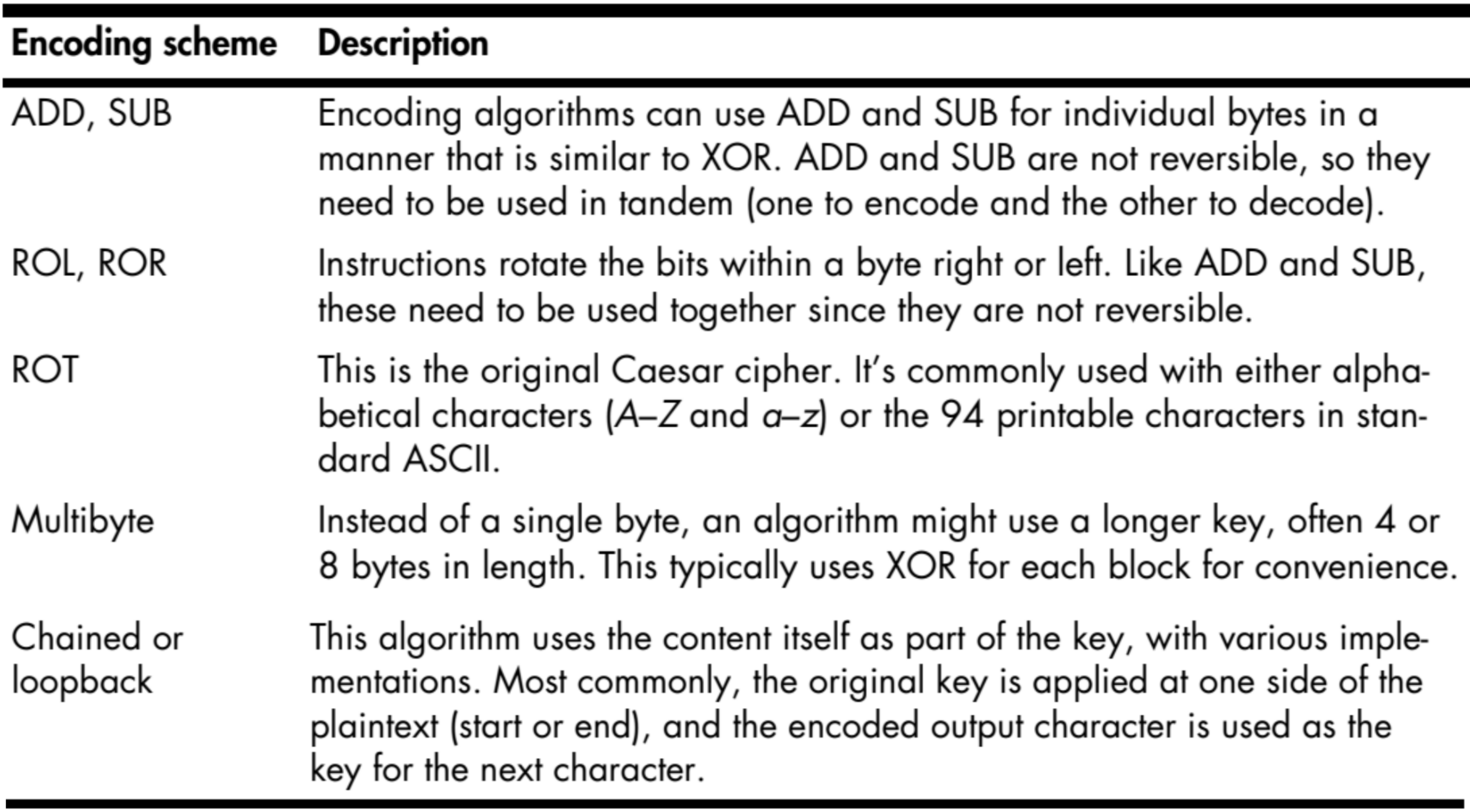
```
mov      edx, [ebp+string_mod]
add      edx, counter
xor      eax, eax
mov      al, [edx]
xor      eax, 12h
mov      ecx, [ebp+string_mod]
add      ecx, counter
mov      [ecx], al
jmp      short loc_4012F4
```

```
loc_401329:
pop      edi
pop      esi
pop      ebx
mov      esp, ebp
pop      ebp
retn
xor_function endp
```

```
loc_4012F4:
mov      eax, counter
add      eax, 1
mov      counter, eax
```

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

Other Simple Encoding Schemes



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 Encoding

- Used to represent binary data in an ASCII string format.
- Commonly found in malware.
  - So, you need to know how to recognize it
- The term Base64 is taken from the MIME standard
  - MIME: Multipurpose Internet Mail Extensions
  - While originally developed to encode email attachments for transmission, it is now widely used for HTTP and XML

Base64 Encoding

- Converts 6 bits into one character (8 bits) in a 64-character alphabet
  - Base64-encoded data is longer than original data
  - Every 3 bytes of binary data → 4 bytes of Base64-encoded data

- MIME's Base64 uses 64 characters:
  - ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - abcdefghijklmnopqrstuvwxyz
  - 0123456789 +/
  - Besides, the '=' is used to indicate padding

```
function base64_encode (s)
{
  // the result/encoded string, the padding string, and the pad count
  var base64chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
  var r = "";
  var p = "";
  var c = s.length % 3;

  // add a right zero pad to make this string a multiple of 3 characters
  if (c > 0) {
    for (; c < 3; c++) {
      p += '=';
      s += "\0";
    }
  }

  // increment over the length of the string, three characters at a time
  for (c = 0; c < s.length; c += 3) {

    // we add newlines after every 76 output characters, according to the MIME specs
    if (c > 0 && (c / 3 * 4) % 76 == 0) {
      r += "\r\n";
    }

    // these three 8-bit (ASCII) characters become one 24-bit number
    var n = (s.charCodeAt(c) << 16) + (s.charCodeAt(c+1) << 8) + s.charCodeAt(c+2);

    // this 24-bit number gets separated into four 6-bit numbers
    n = [(n >>> 18) & 63, (n >>> 12) & 63, (n >>> 6) & 63, n & 63];

    // those four 6-bit numbers are used as indices into the base64 character list
    r += base64chars[n[0]] + base64chars[n[1]] + base64chars[n[2]] + base64chars[n[3]];
  }
  // add the actual padding string, after removing the zero pad
  return r.substring(0, r.length - p.length) + p;
}
```

MIME Base64 Encoding Example

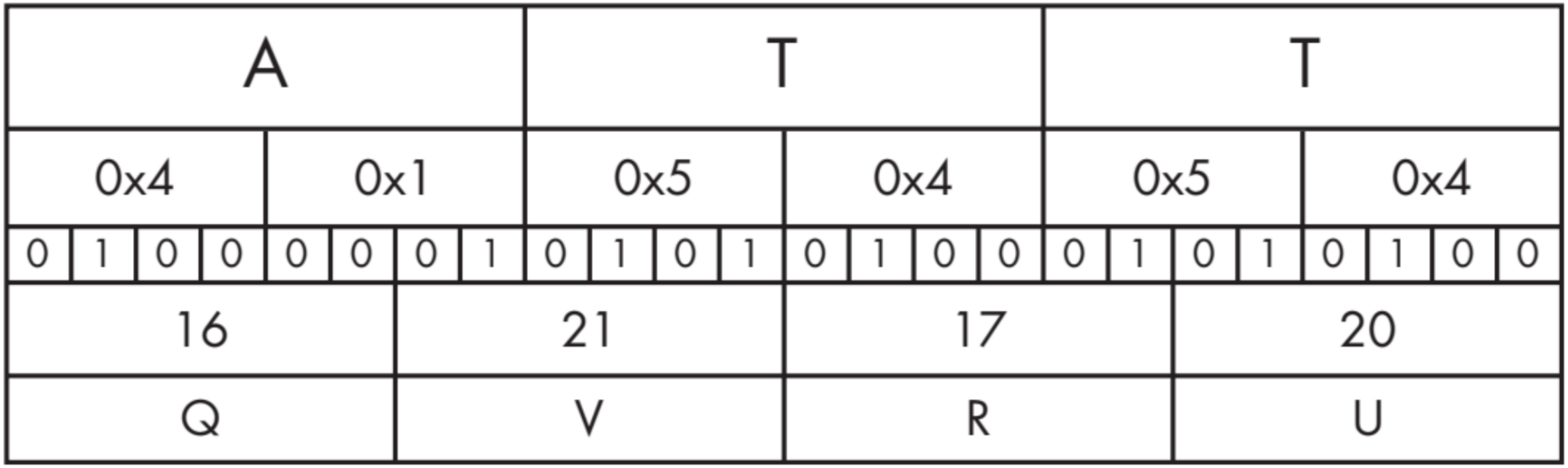
- Following is a part of a raw email file.
  - It uses Base64 encoding.
  - The top few lines show email headers followed by a blank line, with the Base64-encoded data at the bottom.

```
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
```

Listing 13-4: Part of raw email message showing Base64 encoding

Transforming Data to Base64

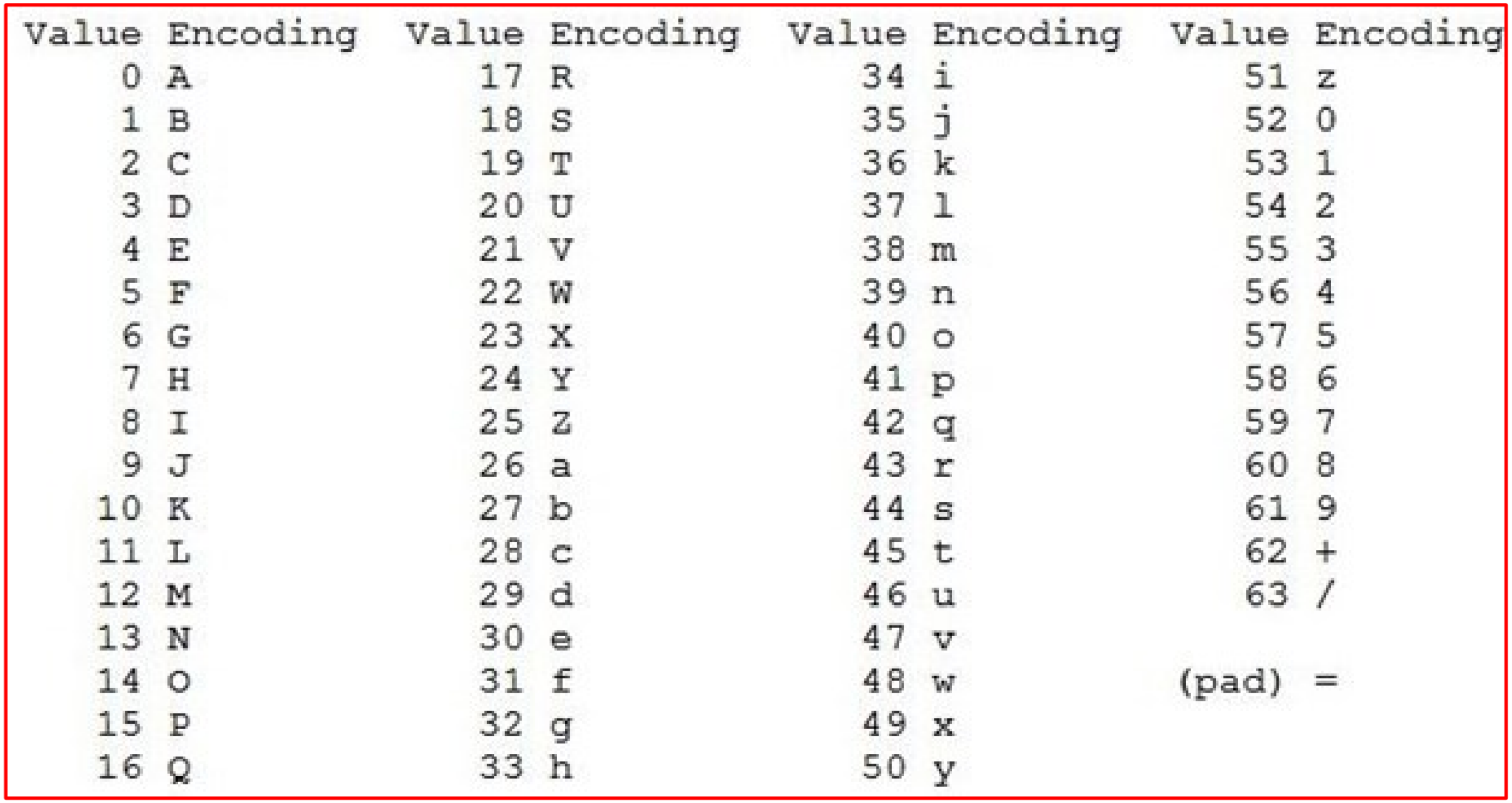
- Use 3-byte chunks (24 bits)
- Break into four 6-bit fields
- Convert each to Base64



A								T								T							
0x4				0x1				0x5				0x4				0x5				0x4			
0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0
16						21						17						20					
Q						V						R						U					

Figure 13-4: Base64 encoding of ATT

The Base64 Alphabet



Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

The ASCII Table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

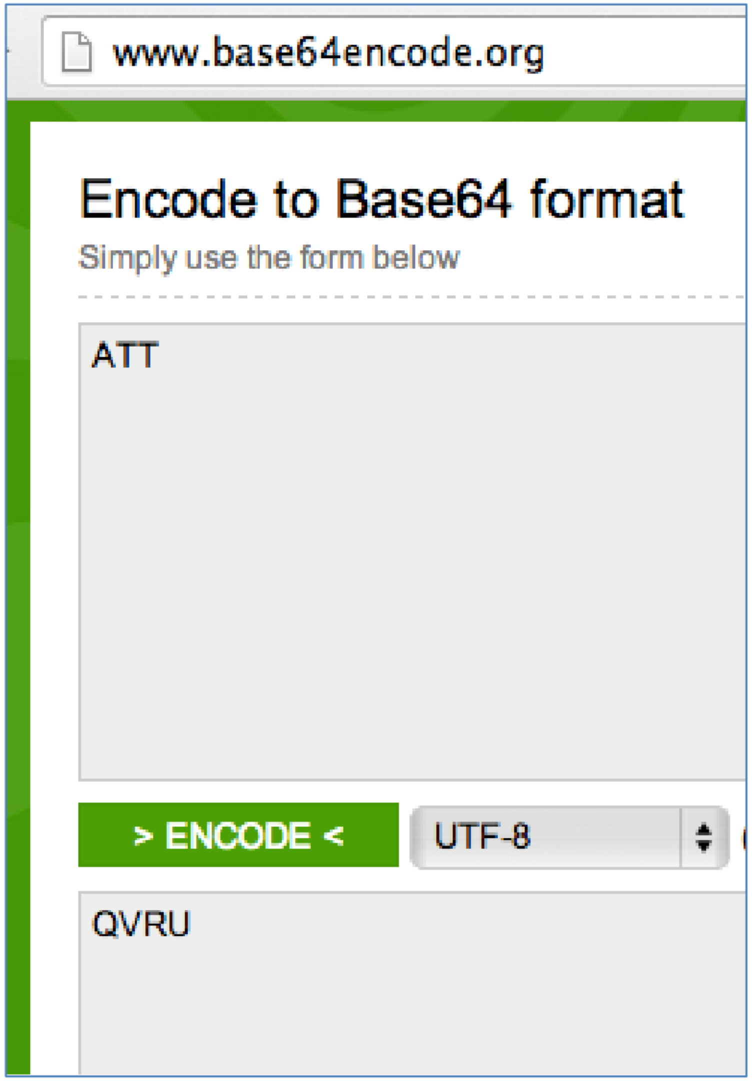
Online Encoding Tool

www.base64decode.org/

3 ASCII Bytes(Characters)

are encode to

4 Base64 Bytes(Characters)


www.base64encode.org

Encode to Base64 format

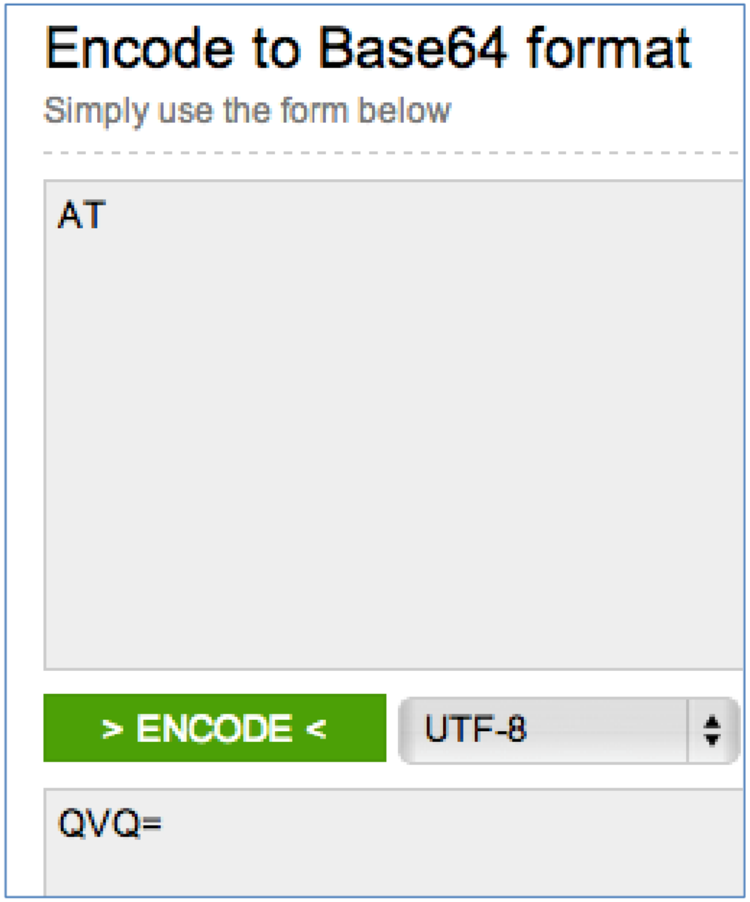
Simply use the form below

ATT

QVRU

Padding

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

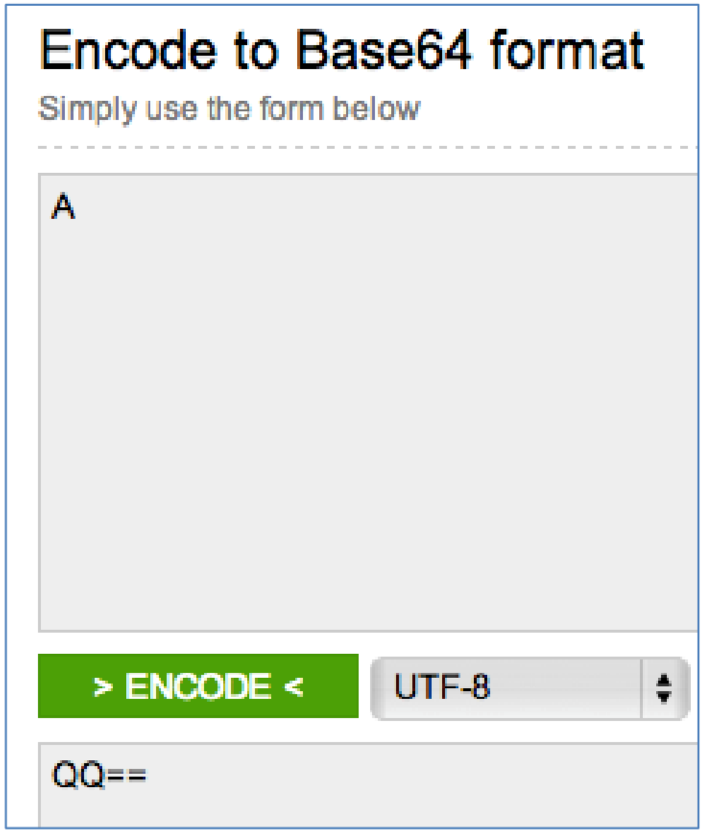


```
AT
```

```
QVQ=
```

Padding

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



Encode to Base64 format

Simply use the form below

A

UTF-8

QQ==

Example

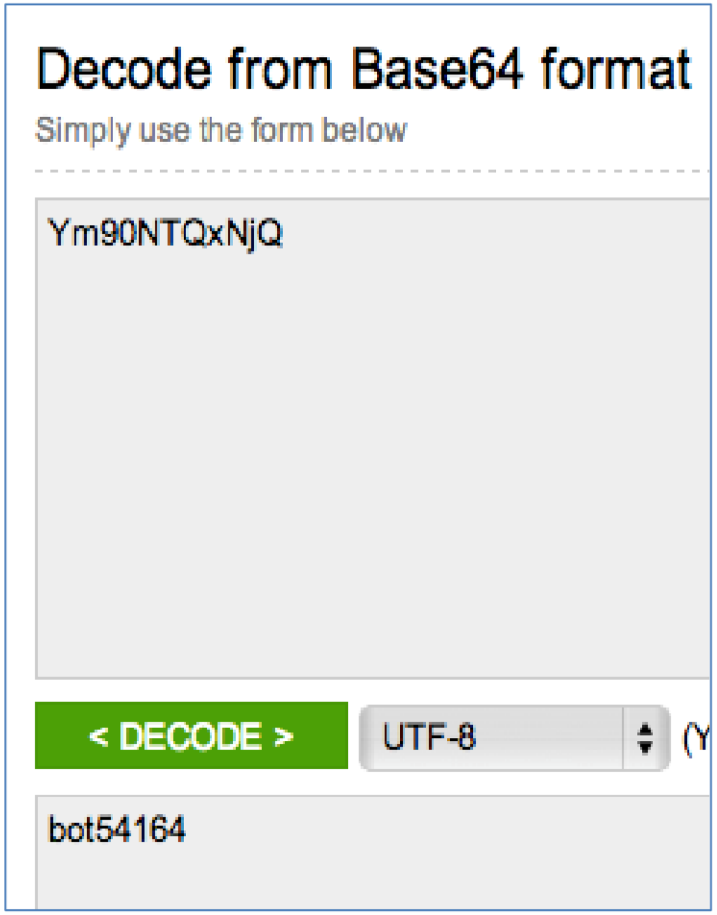
- URL and cookie are Base64-encoded

```
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: Ym9ONTQxNjQ

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: Ym9ONTQxNjQ
```

Listing 13-5: Sample malware traffic

Cookie: Ym9ONTQxNjQ

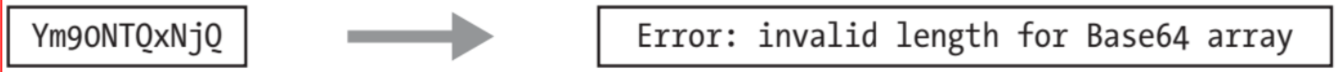
- This has 11 characters—padding is omitted
- Some Base64 decoders will fail, but this one just automatically adds the missing padding.
- Apparently, the attacker is tracking his bots by giving them identification numbers and Base64-encoding that into a cookie.


Decode from Base64 format

Simply use the form below

```
Ym90NTQxNjQ
```

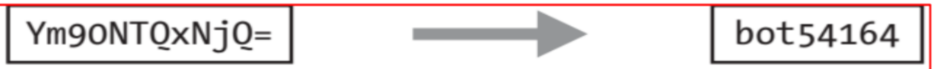
```
bot54164
```



```
Ym9ONTQxNjQ
```

```
Error: invalid length for Base64 array
```

Figure 13-5: Unsuccessful attempt to decode Base64 string



```
Ym9ONTQxNjQ=
```

```
bot54164
```

Figure 13-6: Successful decoding of Base64 string due to addition of padding character

Finding the Base64 Function

- Look for this "indexing string."

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijkl
mnopqrstuvwxyz0123456789+/
```

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

Decoding the URLs

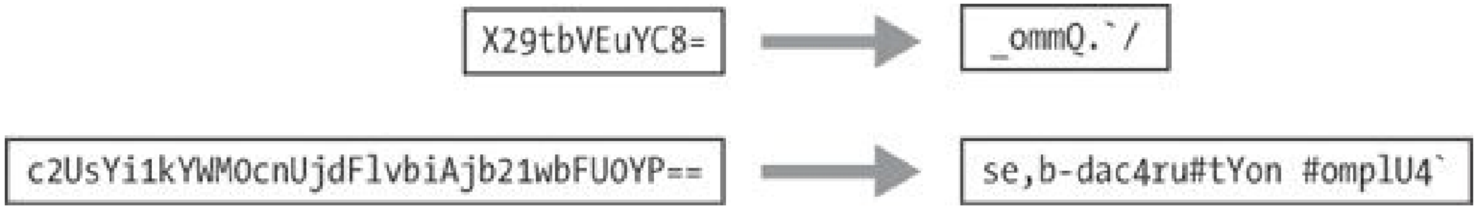

X29tbVEuYC8=
_ommQ.`/
c2UsYi1kYWM0cnUjdFlvbiAjb21wbFU0YP==
se,b-dac4ru#tYon #omplU4`

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

- Custom indexing string

  `aABCDEFGHIJKLMNOPQRSTUVWXYZbcdefghijklmnopqrstuvwxyz0123456789+/`

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

Decoding the URLs

- Malware designers can use non-standard Base64 encoding!
  - New indexing can be done by only changing the indexing string.
  - Relocate some of the characters to the front of the string.

- E.g.: moving the character ‘a’ to the front of the string:

```
aABCDEFGHIJKLMNOPQRSTUVWXYZbcdefghijklmnopqrstuvwxyz0123456789+/
```

  - This string creates a new key for the encoded string, which is difficult to decode without knowledge of this string.
  - Malware designers use this technique to make its output appear to be Base64, while cannot be decoded using standard Base64 indexing.

With Right Custom indexing string

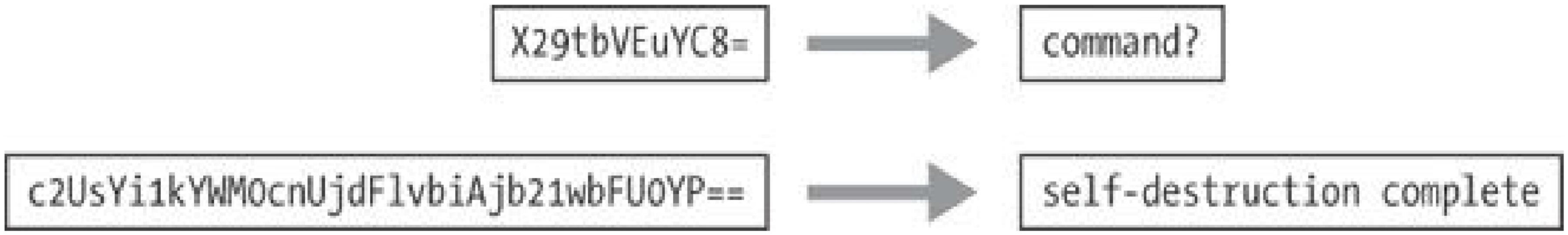

```
X29tbVEuYC8=                              → command?
c2UsYi1kYWM0cnUjdFlvbiAjb21wbFU0YP==      → self-destruction complete
```

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.

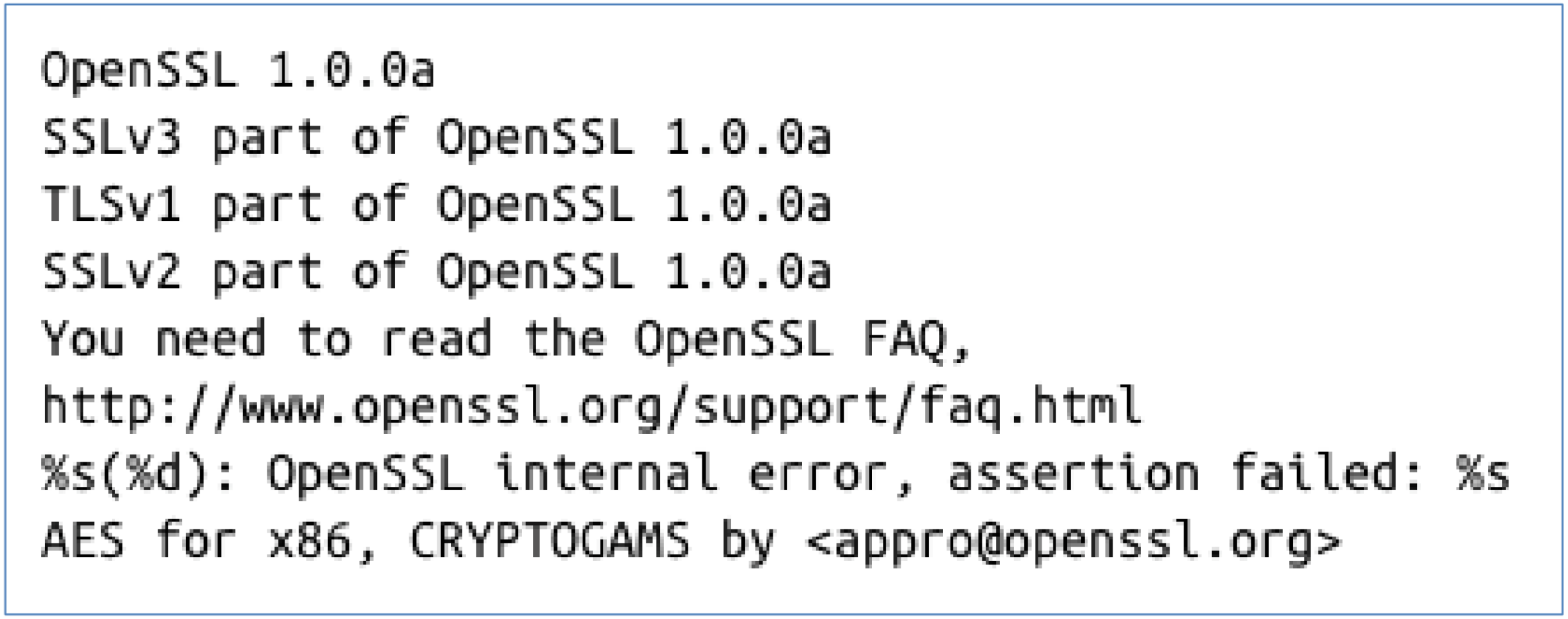
- Disadvantages of strong encryption
  - Large cryptographic libraries required
  - May make code less portable
  - Standard cryptographic libraries are easily detected
    - Via function imports, function matching, or identification of cryptographic constants
  - Symmetric encryption requires a way to hide the key

Strong Cryptography

- Several ways to identify the use of standard cryptography.
  - Recognizing strings that reference cryptographic functions
  - Recognizing Imports that reference cryptographic functions
  - Using tools to Searching Cryptographic Constants.
  - Using tools to Searching for High-Entropy Content

Recognizing Strings

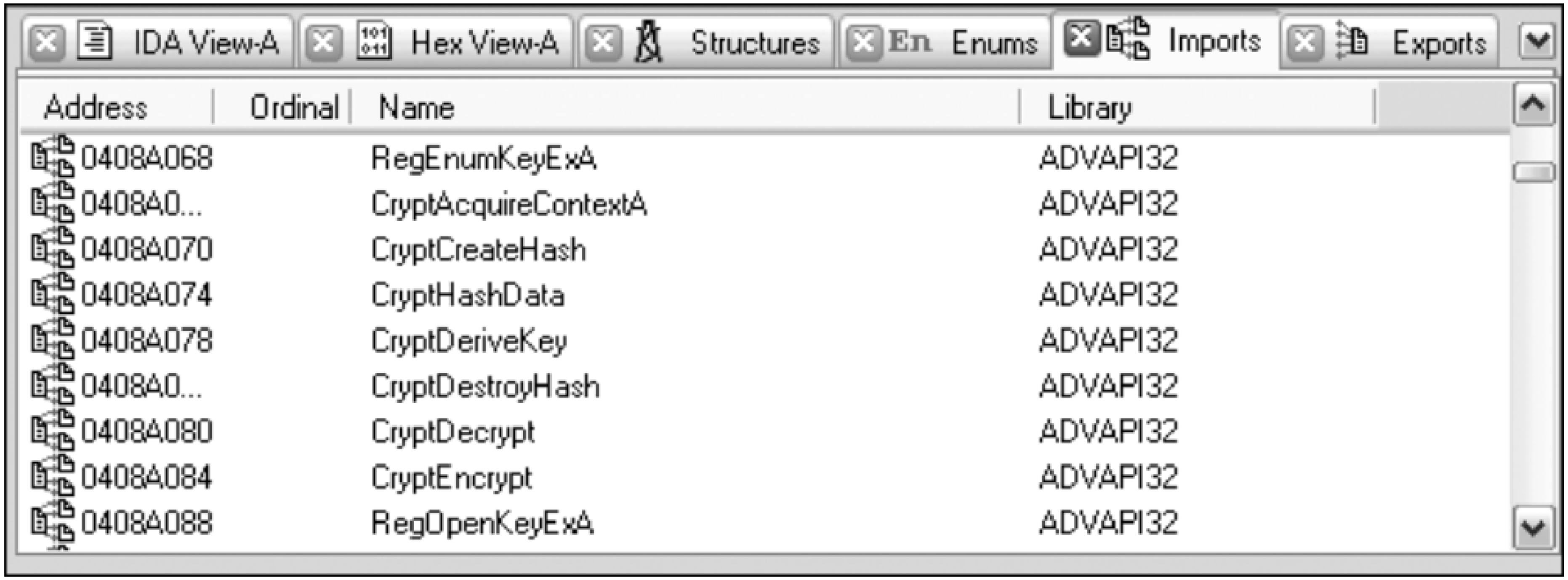
- Strings found in malware encrypted with OpenSSL



```
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 Imports

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

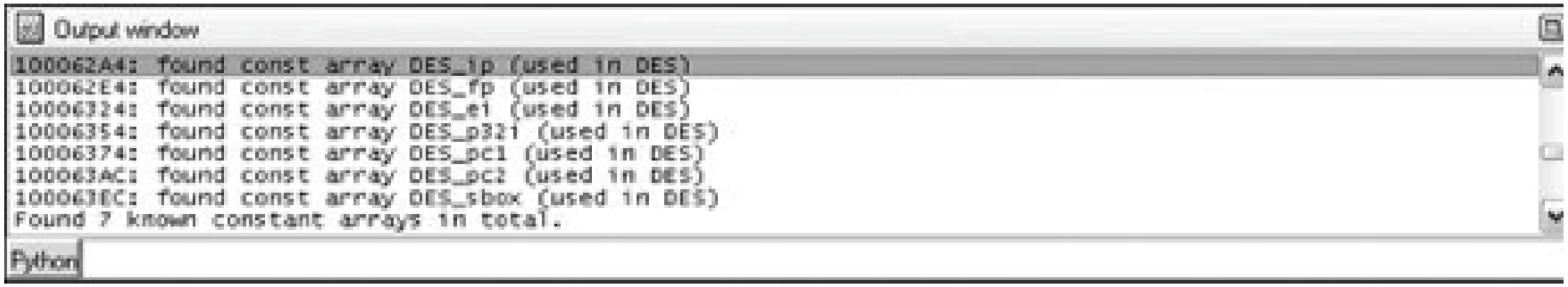
Figure 13-9: IDA Pro imports listing showing cryptographic functions

Searching for Cryptographic Constants

- IDA Pro's FindCrypt2 Plug-in.
  - 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



Output window

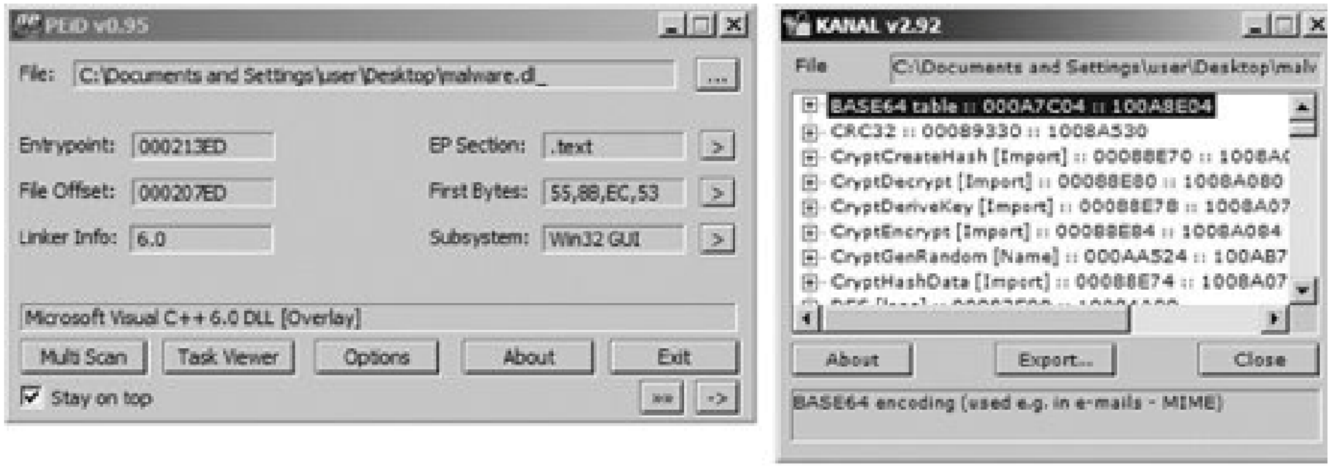
```
100062A4: found const array DES_ip (used in DES)
100062E4: found const array DES_fp (used in DES)
10006324: found const array DES_e1 (used in DES)
10006354: found const array DES_p32i (used in DES)
10006374: found const array DES_pc1 (used in DES)
100063AC: found const array DES_pc2 (used in DES)
100063EC: found const array DES_sbox (used in DES)
Found 7 known constant arrays in total.
```

Python

Figure 14-10. IDA Pro FindCrypt2 output

Krypto ANALyzer (PEiD Plug-in)

- Has a wider range of constants than FindCrypt2
  - More false positives
- Also finds Base64 tables and crypto function imports



```
BASE64 table :: 000A7C04 :: 100A8E04
CRC32 :: 00089330 :: 1008A530
CryptCreateHash [Import] :: 00088E70 :: 1008A
CryptDecrypt [Import] :: 00088E80 :: 1008A080
CryptDeriveKey [Import] :: 00088E78 :: 1008A07
CryptEncrypt [Import] :: 00088E84 :: 1008A084
CryptGenRandom [Name] :: 000AA524 :: 100AB7
CryptHashData [Import] :: 00088E74 :: 1008A07
```

BASE64 encoding (used e.g. in e-mails - MIME)

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

Searching for High-Entropy Content

- Entropy measures disorder in a set of data.

- To calculate it, count the number of occurrences of each byte from 0 to 255
  - Calculate Pi = Probability of value i
  - Then sum Pi log(1/Pi) for i = 0 to 255.

$$H(x) = \sum_{i=1}^{n} [P(x_i) \cdot \log_b(\frac{1}{P(x_i))}]$$

Calculator $Log_2(x)$

- Higher entropy indicates higher uncertainty and a more chaotic system.
  - If all the bytes are equally likely, the entropy is 8 (maximum disorder)
  - If all the bytes are the same, the entropy is zero

Example

1. You have a sequence of numbers:

1 0 3 5 8 3 0 7 0 1

2. Each distinct character has a different probability associated with

- $p(1) = 2/10$.
- $p(0) = 3/10$.
- $p(3) = 2/10$.
- $p(5) = 1/10$.
- $p(8) = 1/10$.
- $p(7) = 1/10$.

3. Shannon entropy equals:

$$H = p(1) * log_2(\frac{1}{p(1)}) + p(0) * log_2(\frac{1}{p(0)}) + p(3) * log_2(\frac{1}{p(3)}) + p(5) * log_2(\frac{1}{p(5)}) + p(8) * log_2(\frac{1}{p(8)}) + p(7) * log_2(\frac{1}{p(7)})$$

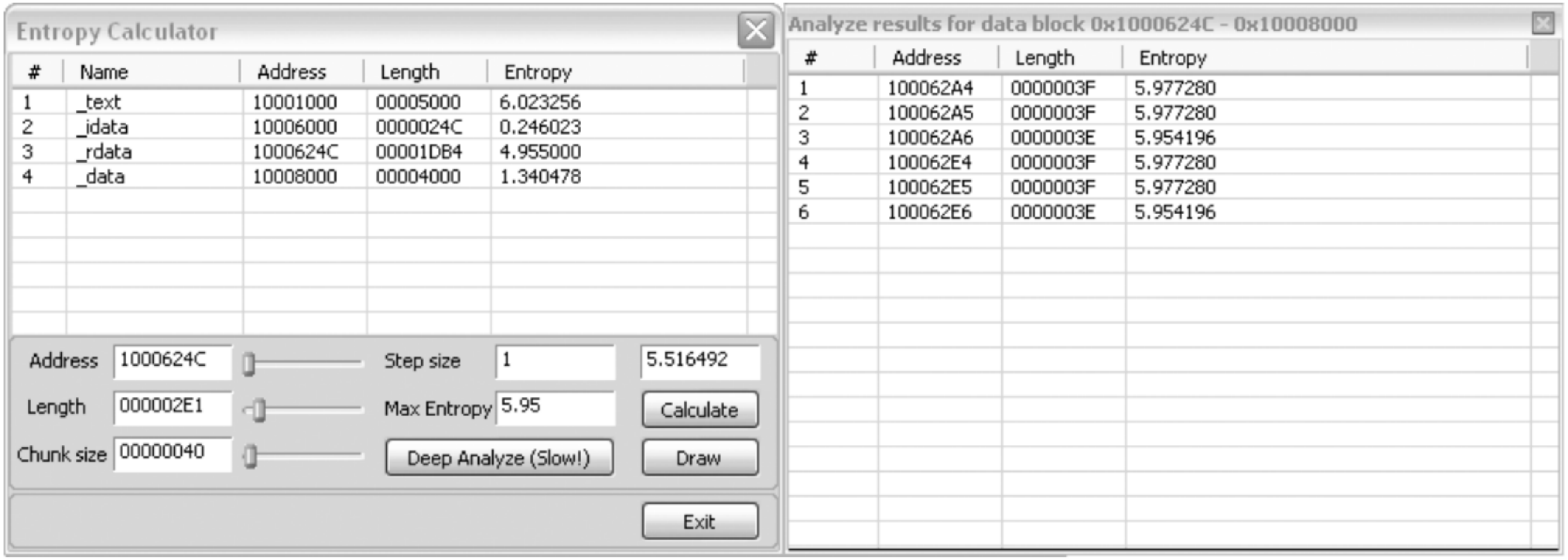
- After inserting the values:

$$H = 0.2 * log_2(\frac{1}{0.2}) + 0.3 * log_2(\frac{1}{0.3}) + 0.2 * log_2(\frac{1}{0.2}) + 0.1 * log_2(\frac{1}{0.1}) + 0.1 * log_2(\frac{1}{0.1}) + 0.1 * log_2(\frac{1}{0.1})$$

- $H = 2.44644$.

Searching for High-Entropy Content

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



#	Name	Address	Length	Entropy
1	_text	10001000	00005000	6.023256
2	_idata	10006000	0000024C	0.246023
3	_rdata	1000624C	00001DB4	4.955000
4	_data	10008000	00004000	1.340478

Address: 1000624C
Length: 000002E1
Chunk size: 00000040
Step size: 1
Max Entropy: 5.95
5.516492

Analyze results for data block 0x1000624C - 0x10008000

#	Address	Length	Entropy
1	100062A4	0000003F	5.977280
2	100062A5	0000003F	5.977280
3	100062A6	0000003E	5.954196
4	100062E4	0000003F	5.977280
5	100062E5	0000003F	5.977280
6	100062E6	0000003E	5.954196

Figure 13-12: IDA Pro Entropy Plugin

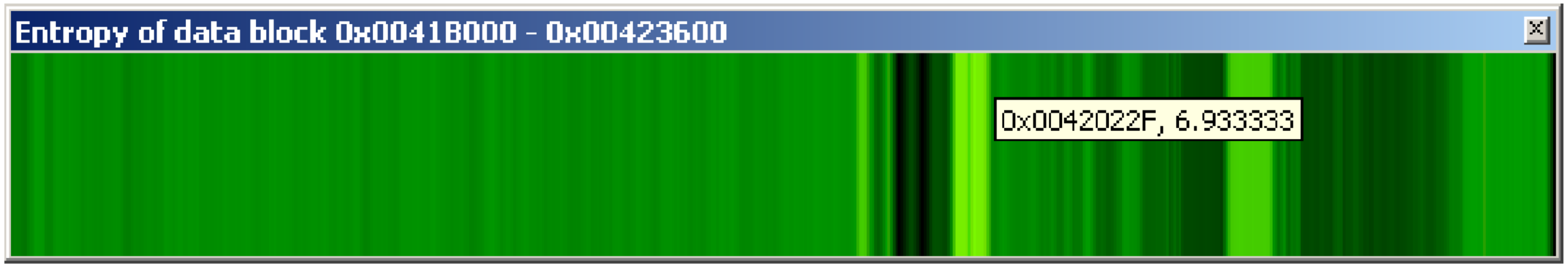
Recommended Parameters

- 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 it.
  - Use the StandAlone version
  - Double-click region, then Calculate, Draw
  - Lighter regions have high entropy
  - Hover over the graph to see the numerical value


Entropy of data block 0x0041B000 - 0x00423600

0x0042022F, 6.933333

Entropy Graph for Malicious Executable

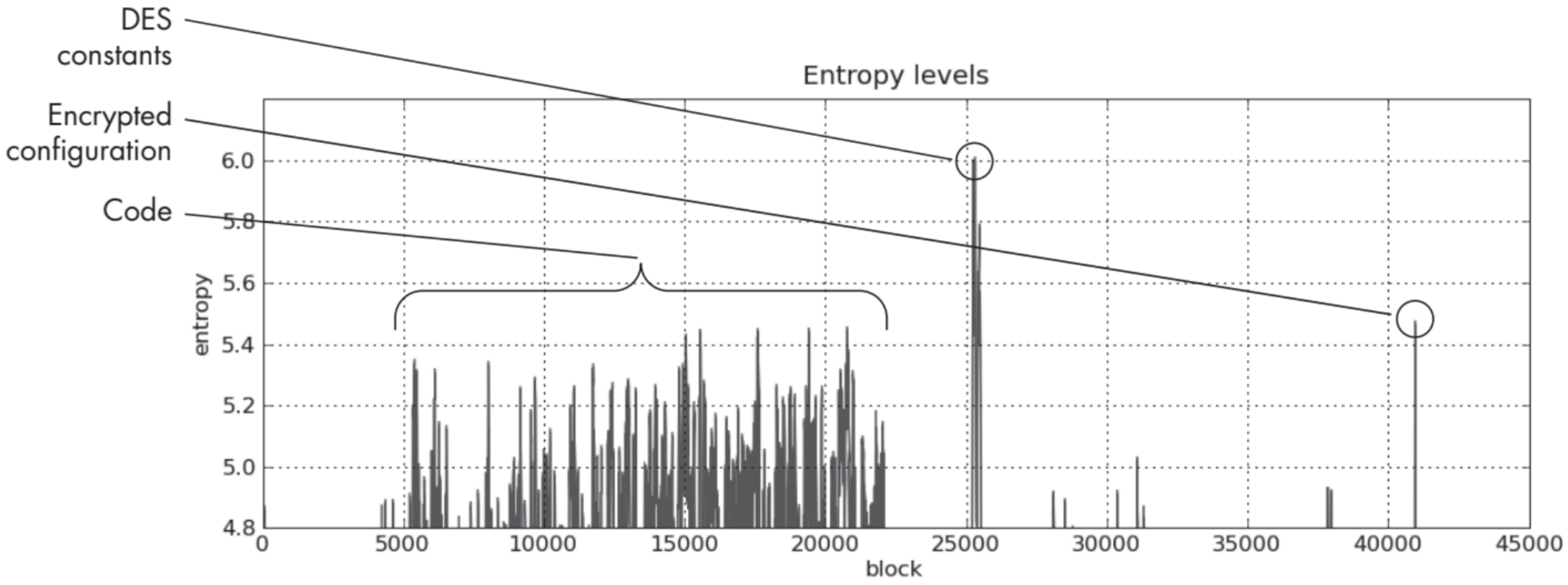


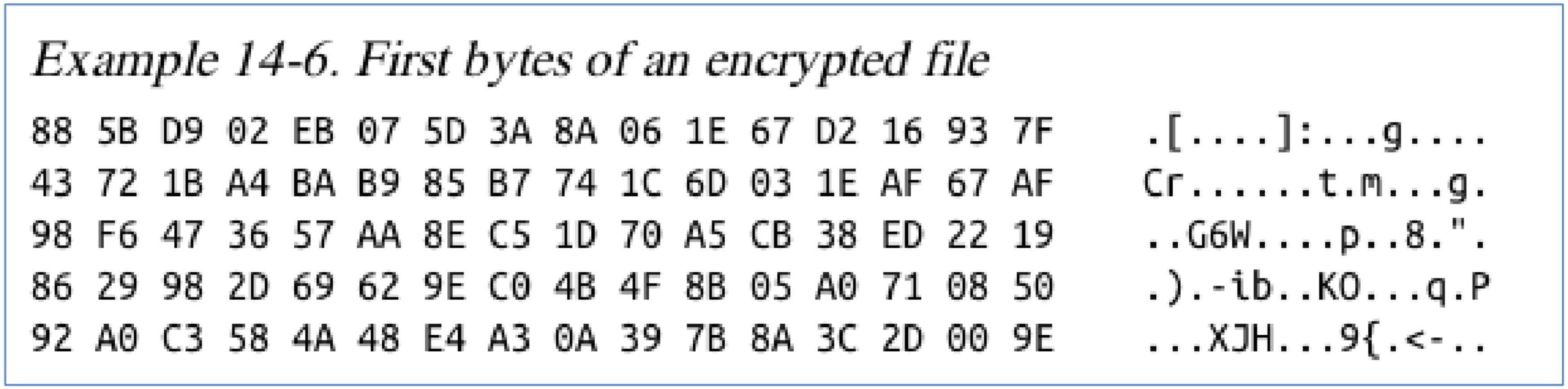
Figure 13-13: Entropy graph for a malicious executable

Custom Encoding

Homegrown Encoding Schemes

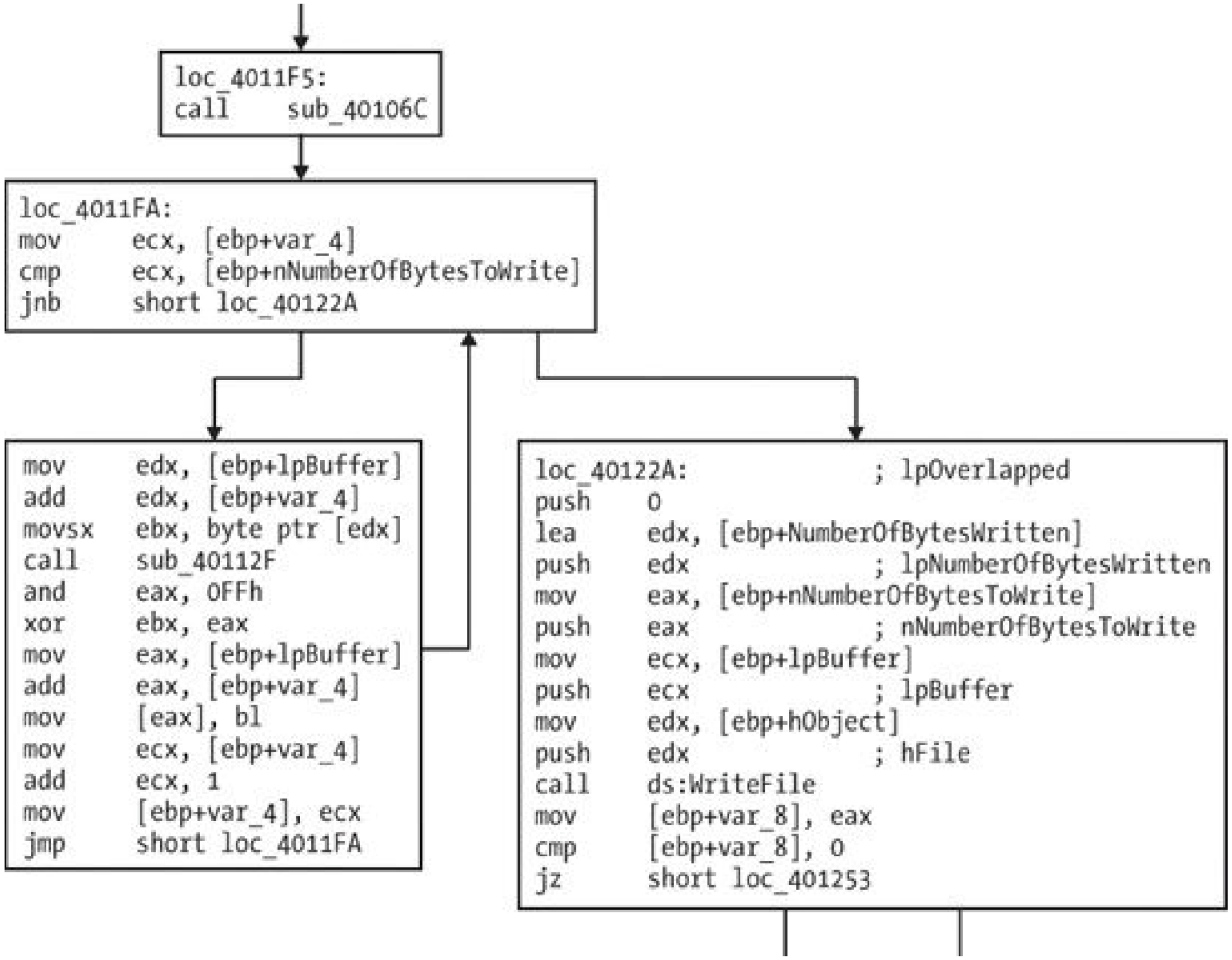
- Examples
  - One round of XOR, then Base64
  - Custom algorithm, possibly similar to a published cryptographic algorithm

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



```
loc_4011F5:
call    sub_40106C
```

```
loc_4011FA:
mov     ecx, [ebp+var_4]
cmp     ecx, [ebp+nNumberOfBytesToWrite]
jnb     short loc_40122A
```

```
mov     edx, [ebp+lpBuffer]
add     edx, [ebp+var_4]
movsx   ebx, byte ptr [edx]
call    sub_40112F
and     eax, 0FFh
xor     ebx, eax
mov     eax, [ebp+lpBuffer]
add     eax, [ebp+var_4]
mov     [eax], bl
mov     ecx, [ebp+var_4]
add     ecx, 1
mov     [ebp+var_4], ecx
jmp     short loc_4011FA
```

```
loc_40122A:                     ; lpOverlapped
push    0
lea     edx, [ebp+NumberOfBytesWritten]
push    edx                     ; lpNumberOfBytesWritten
mov     eax, [ebp+nNumberOfBytesToWrite]
push    eax                     ; nNumberOfBytesToWrite
mov     ecx, [ebp+lpBuffer]
push    ecx                     ; lpBuffer
mov     edx, [ebp+hObject]
push    edx                     ; hFile
call    ds:WriteFile
mov     [ebp+var_8], eax
cmp     [ebp+var_8], 0
jz      short loc_401253
```

Figure 14-14. Function graph showing an encrypted write

Advantages of Custom Encoding

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

Decoding

Two Methods

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

Self-Decoding

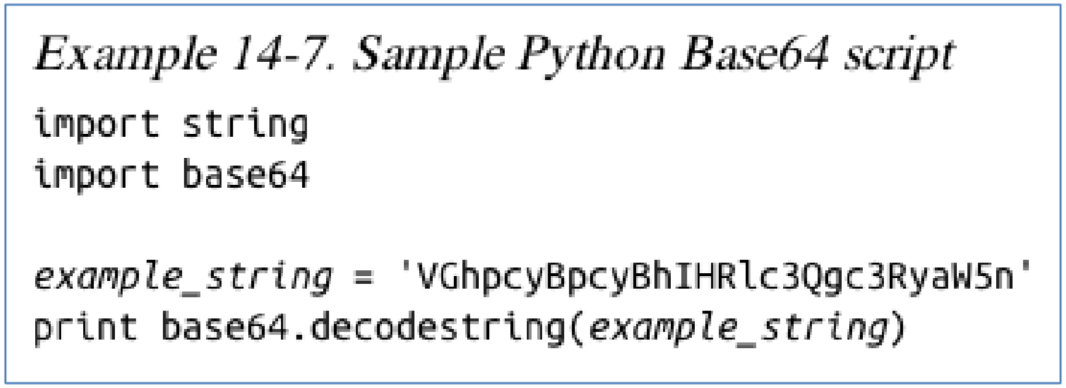
- Stop the malware in a debugger with data decoded

- 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 to be decoded

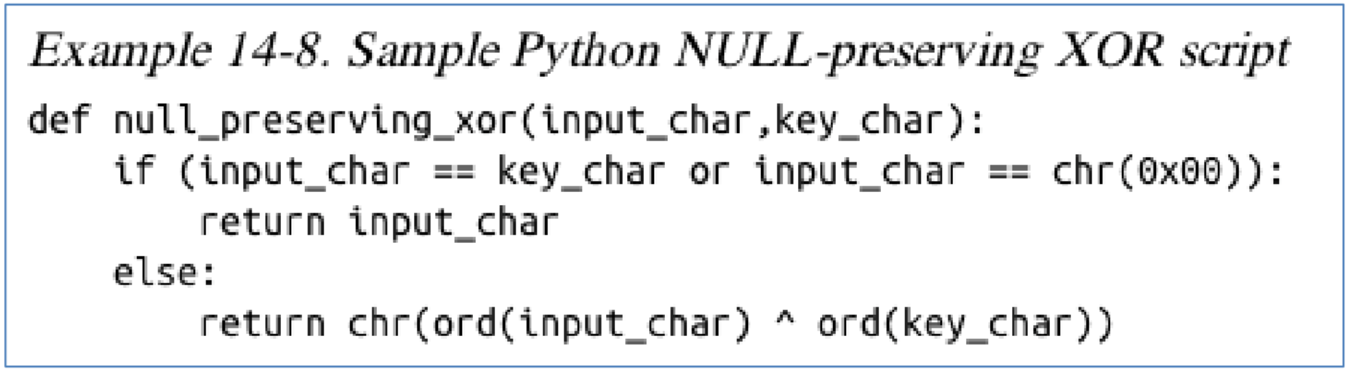
Manual Programming of Decoding Functions

- 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

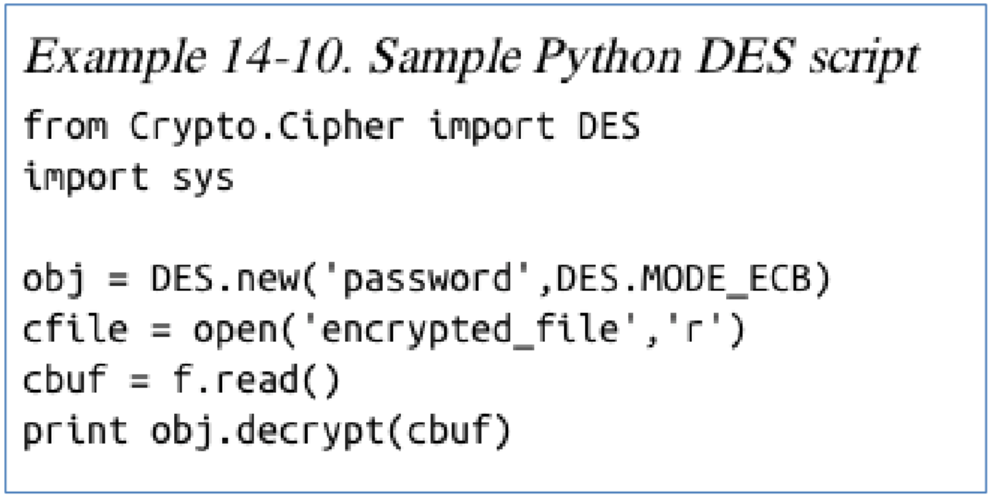
```
def null_preserving_xor(input_char,key_char):
    if (input_char == key_char or input_char == chr(0x00)):
        return input_char
    else:
        return chr(ord(input_char) ^ ord(key_char))
```

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

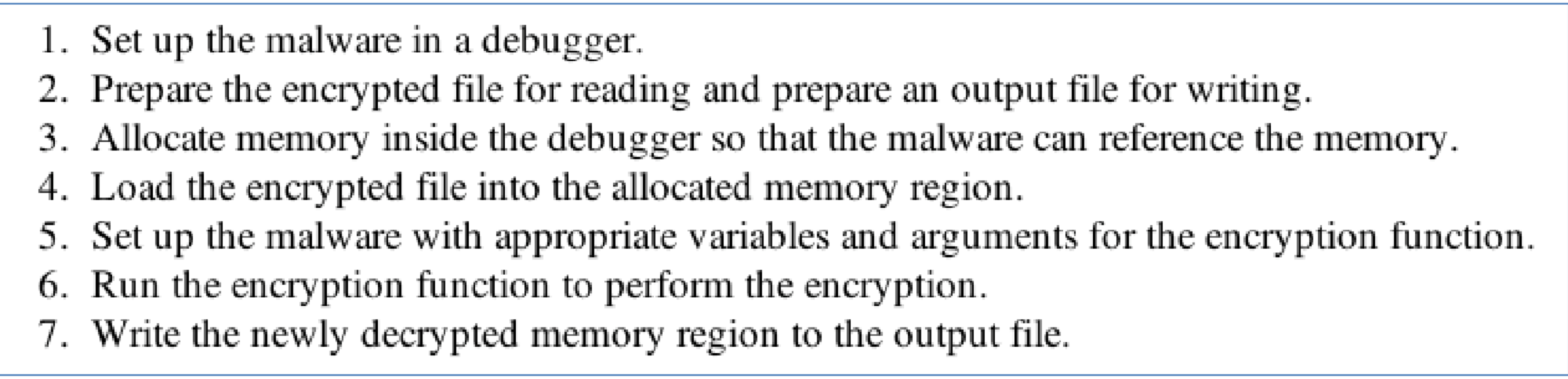
- Good for standard algorithms



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
```

Main Sources for these slides

- *Michael Sikorski and Andrew Honig, "Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software"; ISBN-10: 1593272901.*
- *Xinwen Fu, "Introduction to Malware Analysis," University of Central Florida*
- *Sam Bowne, "Practical Malware Analysis," City College San Francisco*
- *Abhijit Mohanta and Anoop Saldanha, "Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware," ISBN: 1484261925.*

Thank you