Obfuscation used by an HTTP Bot

Author: Daryl Ashley

Email: ashley@infosec.utexas.edu

Date: September 20, 2010

Information Security Office
The University of Texas at Austin
SECURUS // VIGILARE // INSANUS

Introduction

The purpose of this paper is to analyze the obfuscation technique used by an HTTP Bot (aka Web bot). The name of the malware specimen analyzed in this paper is micupdate.exe. The md5 hash of this file is dc21cf8b9a8b9573fa433d0a002d26f1. The original malware was patched to remove the name of the command and control (C&C) website that was encoded in the malware. A sample of the malware can be requested by sending an email to security@utexas.edu.

The malware uses the Base64 algorithm to decode the commands it receives from the command and control (C&C) website. However, the malware does not use a standard Base64 chart to perform the decoding. Instead, it makes use of a "scrambled" chart. Once the contents of the scrambled chart are known, an Intrusion Detection System (IDS) signature can be written to detect specific commands received by an infected host from the C&C website. This paper also discusses ideas for detecting variants of the malware.

Overview of Base64 Algorithm

Base64 encoding was first used to convert binary data into 7-bit ASCII characters. The use of 7-bit ASCII characters was a requirement for certain transfer protocols, such as SMTP (uCertify 2006). The algorithm makes use of a 64 character alphabet to encode a given piece of data (IETF 2006). The MIME version of Base64 algorithm uses the characters A-Z, a-z, and 0-9 as the first 62 characters. The last two characters may be different depending on the variant used (uCertify 2006). The characters and corresponding numeric values can be arranged in a chart, such as the chart shown in Figure 1. The chart maps 64 ASCII characters to numeric values ranging from 0 - 63.

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

Figure 1: Base 64 Encoding Chart (Tschabitscher 2010)

The Base64 encoding algorithm translates three 8-bit numbers into four 6-bit numbers. The Base64 encoding chart is used to determine the ASCII character that corresponds to each 6-bit numeric value (IETF 2006). This will yield 4 ASCII characters that can be represented by 7 bits of data (uCertify 2006). Therefore, three characters in the original message are represented as four characters after encoding.

Figure 2 shows how to encode the ASCII string “sle”. The first step is to translate the three characters into their ASCII codes. In Step 2, the ASCII codes are represented as three 8-bit binary numbers. In Step 3, the three 8-bit numbers are partitioned to form four 6-bit numbers. The 6-bit numbers are then converted back to decimal format in Step 4. In Step 5, the 6-bit numbers are mapped to characters using the chart shown in Figure 1. The characters “sle” are encoded into the characters “c2xl”.

ASCII characters: "sle"	
Step 1	's' = 115, 'l' = 108, 'e' = 101
Step 2	01110011 01101100 01100101
Step 3	011100 110110 110001 100101
Step 4	28 54 49 37
Step 5	28 = 'c', 54 = '2', 49 = 'x', 37 = 'l'
Output of algorithm: "c2xl"	

Figure 2: Encoding "sle" into "c2xl"

After the first three bytes of data are encoded, the next three bytes are encoded in the same way. The process is repeated until all of the data is encoded. If the length of the data is not a multiple of three bytes, the algorithm must append padding characters to the end of the encoded message (IETF 2006). The padding procedure will not be discussed in this paper.

The algorithm for decoding an encoded message is the reverse of the algorithm described above. When decoding an encoded message, four letters are decoded into three letters. Figure 3 shows how to decode the "c2xl" characters back to "sle". In Step 1, the numeric value for each letter is determined using the chart in Figure 1. The four numeric values are expressed as 6-bit binary numbers in Step 2. The four 6-bit numbers are used to form three 8-bit binary numbers in Step 3. In Step 4, the three 8-bit numbers are converted into ASCII characters. The algorithm is repeated until all of the data has been decoded.

ASCII characters: "c2xl"

Step 1 'c' = 28, '2' = 54, 'x' = 49, 'l' = 37

Step 2 011100 110110 110001 100101

Step 3 01110011 01101100 01100101

Step 4 115 = 's', 108 = 'l', 101 = 'e'

Output of algorithm: "sle"

Figure 3: Decoding "c2xl" into "sle"

Malware Implementation of Scrambled Chart

The malware uses the Base64 algorithm to decode commands received from the C&C website. However, it does not use the Base64 chart shown in Figure 1. Instead, it makes use of a "scrambled" Base64 chart. Figure 4 shows the portion of the malware that contains the characters used to construct the scrambled chart and Figure 5 shows what the scrambled chart looks like. The characters shown in Figure 4 occupy 64 bytes of memory. Each byte of memory occupies a position ranging from 0 – 63 from offset 00403010. The position determines what value each letter decodes to. For example, 'A' occupies the 0th position of the memory location, so the value for 'A' in the Base64 chart is 0, 'F' is in the 3rd position of the memory location, so the value for 'F' in the Base64 chart is 3.

```
.data:00403000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:00403010  41 42 43 46 47 48 49 4A  64 65 66 67 68 69 6A 6B  ABCFGHIJdefghijk
.data:00403020  4B 4C 4D 4E 4F 50 56 78  79 7A 30 31 32 33 34 57  KLMNOPVxyz01234W
.data:00403030  58 59 5A 61 62 63 6C 6D  51 52 53 54 44 45 55 6E  XYZabclmQRSTDEUn
.data:00403040  6F 70 71 72 73 74 75 76  77 35 36 37 38 39 2B 3D  opqrstuvw56789+=
```

Figure 4: Scrambled ASCII Chart in Malware Memory

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

Figure 5: Base64 chart used by micupdate.exe

Since the infected machine receives encoded commands from the C&C website, the malware must decode the command before the command can be processed. To decode the command, the command must be partitioned into sets of four letters. Each set of four letters is then used as an input for the decoding algorithm. So, the malware must have a method for determining the numeric value that corresponds to each of the four letters.

Each character shown in Figure 4 has a corresponding ASCII code, a number between 0 and 255. For example, the ASCII code for the letter 'P' is 80. The ASCII code for a character can be used as an index into an array that stores the numeric values of the chart. For example, the highest ASCII code for the characters shown in the figure is 122. An array of 122 characters can be used to store the numeric values corresponding to each ASCII code. The following C code fragment can be used to allocate a character array. Since software developers seem to like numbers that are a power of 2, the array will contain 128 characters instead of 122 characters:

```
int chart[128];
```

The ASCII code for each character can then be used as an index into the array. The

numeric value associated with the character can be stored within the array at that index. For example, since the ASCII code for 'P' is 80, the following C code fragment would assign the numeric value 15 to the chart array at index 80:

```
int chart[80] = 15;       // Equivalent to chart['P'] = 15; - the letter 'P' maps to a numeric value 15
```

To decode a command that contains the letter 'P':

1. Convert 'P' to its ASCII value (80).
2. Determine the value stored in the chart array at index 80

The C code fragment below will accomplish these two steps:

```
int val = chart['P'];
```

An array with numeric values from the chart in Figure 1 is listed in Appendix 1 and an array with the scrambled chart is listed in Appendix 2. For the letter 'P', the numeric value associated with chart[80] is 15 in the standard chart. However, it is 21 in the scrambled chart. Once all four numeric values have been determined, the decoding algorithm proceeds as outlined in the previous section.

IDA Pro was used to produce the disassembly of the instructions shown in Figure 6. In order to load the chart into memory, all 64 characters in Figure 4 must be processed by the malware. Recall the position of each character is also the character's numeric value within the scrambled chart. This means that a register used as an index into the array of characters will also contain the numeric value that corresponds to the letter. The EAX register is used for both purposes.

The instruction at offset 00401026 sets the EAX register to 0. This ensures that the 0^{th} position of the 64 character array is processed first. The instruction at offset 0040102A uses the EAX register as an index in the memory location storing the contents of the scrambled chart. The instruction at offset 00401034 increments the value in the EAX register so the next position in the array is processed. Finally, the instructions at offsets 00401035 and 00401038 cause the program to loop as long as the value in the EAX register is less than 40h (decimal 64).

Each letter in the scrambled chart is stored in the ECX register and processed within the loop. The instruction at offset 00401030 uses the ASCII code in the ECX register as an index into an array stored at ESP+114h+var_100. The array at this location stores the numeric values corresponding to each letter in the scrambled chart.

```
.text:00401026                  xor     eax, eax
.text:00401028
.text:00401028 loc_401028:                              ; CODE XREF: sub_401000+38↓j
.text:00401028                  xor     ecx, ecx
.text:0040102A                  mov     cl, byte_403010[eax]
.text:00401030                  mov     [esp+ecx+114h+var_100], al
.text:00401034                  inc     eax
.text:00401035                  cmp     eax, 40h
.text:00401038                  jl      short loc_401028
```

Figure 6: Assembler instructions that load chart into memory

Figure 7 shows a C code fragment that may produce the assembly listing shown in Figure 6. Note that since the ECX and EAX registers contain only 8-bit values during the loop, only the lower 8 bits of each register will contain a non-zero value. This means that the value stored in the ECX register will be equal to the value stored in the CL register because CL is the lower 8 bits of the ECX register. The same can be said about the EAX and AL registers. Therefore, the C code fragment replaces cl and al in the assembly listing with ecx and eax.

```
//Code snippet to load scrambled Base64 chart into memory

char key[64] =
"ABCFGHIJdefghijkKLMNOPVxyz01234WXYZabclmQRSTDEUnopqrstuvw56789+=";

void sub_401000 (char *encoded, char *decoded)
{
        …
        int eax;
        unsigned char ecx;
        char chart[128];

        …
        eax = 0;
        while (eax < 0x40) {
                ecx = key[eax];          // load letter from key[63] into ecx register
                chart[ecx] = eax;        // chart['P'] = offset of 'P' in key[] array
                eax++;
        }
        …
}
```

Figure 7: Possible C code snippet

OllyDbg was used to verify that an array of characters was used to store the numeric values of the scrambled chart. Figure 8 shows the disassembly for the same set of instructions when running the malware in OllyDbg. The instruction at offset 00401030 indicates that the address of the array that stores the numeric values of the scrambled chart is located at ESP+14h.

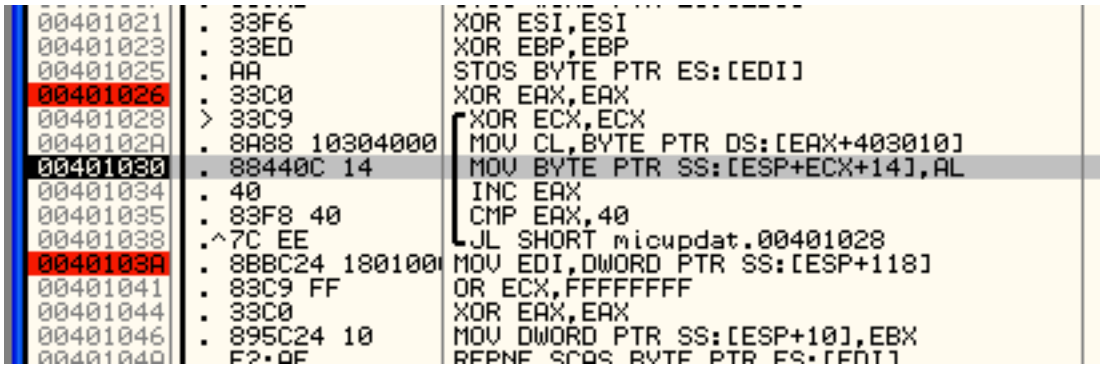

Figure 8: OllyDbg Assembler listing

Figure 9 shows that the ESP register was set to 0x12FE04 before the loop was executed. So, the address of the array should be 0x12FE18. After the loop had completed execution, the contents of memory at this address were inspected. Figure 10 shows the contents of the array at this address. The contents matched the chart shown in Appendix 2.

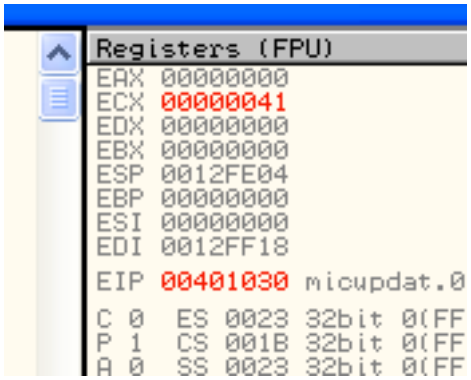

Figure 9: OllyDbg State of registers before loading chart into memory

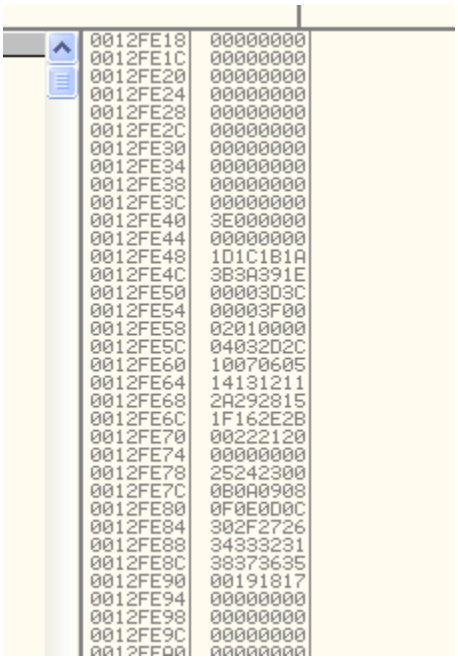

Figure 10: OllyDbg contents of memory after chart loaded into memory

Malware Implementation of Decoding Algorithm

The following algorithm describes how the micupdate.exe malware decodes a set of four letters into the original three letters. The disassembly will look different because the implementation is setup to loop through all of the letters of the encoded message. However, the steps outlined below should make the disassembled instructions easier to understand. The letters "c2xl" will be used in the algorithm outline.

Step 1: Lookup the numeric value for 'c' and convert it into a 6-bit binary value:

'c' -> 28 -> 011100

At this point, there are not enough bits to decode using the ASCII chart because an 8-bit number is needed.

Step 2: Lookup the numeric value for '2' and convert it into a 6-bit binary value:

'2' -> 54 -> 110110

Concatenate the 6-bit numbers from steps 1 and 2. This will produce a 12-bit binary number. This number must be saved so it can be used in the remaining steps of the algorithm.

011100110110

Copy the 12-bit number into a temporary location. Shift the number four bits to the right. This will produce an 8-bit number that can be decoded to an ASCII value. This is the first letter of the decoded command.

01110011 -> 115 -> 's'

Step 3: Lookup the numeric value for 'x' and convert it into a 6-bit binary value:

'x' -> 49 -> 110001

Append the 6-bit number to the 12-bit number from step 2. This will produce an 18-bit number. This number must be saved so it can be used in the final step of the algorithm.

011100110110110001

Copy the 18-bit number into a temporary location. Shift the number two bits to the right. This will produce a 16-bit number:

0111001101101100

The upper-most 8 bits have already been decoded into an ASCII character. But the lower 8 bits of the number have not been decoded into an ASCII value. The lower 8 bits are the second letter of the decoded command.

01101100 -> 108 -> 'l'

Step 4: Lookup the numeric value for 'l' and convert it into a 6-bit binary value:

'l' -> 37 -> 100101

Append the 6-bit number to the 18-bit number from step 3. This will produce a 24-bit number.

011100110110110001100101

Since the first 16 bits have already been decoded, only the lower 8 bits must be decoded. Note that for this step, no right shift operation is necessary. The third letter of the command has been decoded.

01100101 -> 101 -> 'e'

The malware uses the following registers to perform the algorithm:

1. The EAX register stores the ASCII code of an encoded letter
2. The ECX register stores the numeric value of the character in the EAX register.
3. The ESI register is used to store the concatenated 6-bit binary values.
4. The EBX/BL register is used to decode an 8-bit number into an ASCII character.
5. Before the contents of the EBX register can be decoded to an ASCII character, the contents must be shifted 4, 2, or 0 bits to the right depending on which step in the algorithm is being executed. The CL register is used to store the number of bits to shift the contents of the EBX register.

Figure 11 shows the malware's implementation of the algorithm. The disassembly shown in the figure was produced by IDA Pro.

```
.text:00401051 loc_401051:                             ; CODE XREF: sub_401000+B2↓j
.text:00401051                 mov     eax, [esp+114h+arg_0]
.text:00401058                 add     edx, 6
.text:0040105B                 shl     esi, 6
.text:0040105E                 movsx   ecx, byte ptr [ebx+eax]
.text:00401062                 xor     eax, eax
.text:00401064                 mov     al, [esp+ecx+114h+var_100]
.text:00401068                 mov     ecx, eax
.text:0040106A                 xor     esi, ecx
.text:0040106C                 cmp     edx, 7
.text:0040106F                 jle     short loc_40109A
.text:00401071                 mov     eax, edx
.text:00401073                 lea     ecx, [edx-8]
.text:00401076                 shr     eax, 3
.text:00401079                 mov     edi, eax
.text:0040107B                 neg     edi
.text:0040107D                 lea     edx, [edx+edi*8]
.text:00401080
.text:00401080 loc_401080:                             ; CODE XREF: sub_401000+94↓j
.text:00401080                 mov     edi, [esp+114h+arg_4]
.text:00401087                 mov     ebx, esi
.text:00401089                 sar     ebx, cl
.text:0040108B                 sub     ecx, 8
.text:0040108E                 inc     ebp
.text:0040108F                 dec     eax
.text:00401090                 mov     [edi+ebp-1], bl
.text:00401094                 jnz     short loc_401080
.text:00401096                 mov     ebx, [esp+114h+var_104]
.text:0040109A
```

Figure 11: Malware Base64 decode implementation

The instruction at offset 00401051 reads a letter from the encoded message, and moves the ASCII code for the letter into the EAX register. The EBP register is used as an index into the character array that is used to store the encoded message. The instruction at offset 0040105E performs the lookup in the scrambled chart to determine the numeric value corresponding to the letter. The numeric value is stored in the ECX register.

The ESI register is used to concatenate the 6-bit numeric values as the letters are decoded from the scrambled chart. The instruction at offset 0040105B shifts the contents of the ESI register to the left by 6 bits, so the lower 6 bits of the register will be zeroed out. The instruction at offset 0040106A loads the contents of the ECX register into the vacated lower bits of the ESI register via an XOR operation. In this way, a 6-bit numeric value is concatenated to the previous value stored in the ESI register.

The instruction at offset 00401087 copies the contents of the ESI register into the EBX register so that the value in the ESI register is not modified. The instruction at offset 00401089 performs the shift right operation. The contents of the EBX register are shifted to the right based on a value in the CL register. This allows the EBX register to be shifted a different number of bits depending on which 6-bit value has been concatenated. The instruction at offset 00401090 copies the contents of the lowest 8 bits of the EBX register into a character array that will store the decoded command.

IDS Detection

Snort is an open source Intrusion Detection System that can be used to monitor network traffic based on a set of "signatures". If a network packet matching a signature is detected, Snort will generate an alert so the host responsible for generating the network traffic can be inspected (Scott 2004). Since the encoding/decoding algorithm for this malware has been determined, a Snort signature can be written to detect specific C&C commands.

For example, suppose "sleep 30" is the C&C command that instructs the infected host to wait 30 minutes before requesting another command. Since the number of minutes for the "sleep" command may vary, only the first six characters of the command are constant ("sleep" plus a space character). Figure 12 shows the encoded character string for "sleep " based on the scrambled chart. The encoded character string is "2upczxAX". Snort's "content" keyword can be used to look for this string within TCP packets (Scott 2004). An example Snort signature is shown below:

```
alert tcp $HOME_NET 1024: -> any 80 (content:"2upczxAX";)
```

ASCII characters: "sleep "	
Step 1	's' = 115, 'l' = 108, 'e' = 101, 'e' = 101, 'p' = 112, ' ' = 32
Step 2	01110011 01101100 01100101 01100101 01110000 00100000
Step 3	011100 110110 110001 100101 011001 010111 000000 1000000
Step 4	28 54 49 37 25 23 0 32
Step 5	28 = '2', 54 = 'u', 49 = 'p', 37 = 'c', 25 = 'z', 23 = 'x', 0 = 'A', 32 = 'X'
Encoded characters: "2upczxAX"	

Figure 12: Encoding "sleep " with scrambled Base64 chart

Signatures for other commands can be created in a similar way if the command set for this HTTP Bot is known.

Other Possible Detection Methods

In the previous section, a Snort signature was created based on the scrambled chart in the malware. What if the malware author modifies the code so that a different scrambled chart is used? The encoded command would probably be different, and a signature would need to be created based on the new scrambled chart. Is there another way to detect network activity generated by variants of this malware? For the sake of argument, suppose the C&C website embeds the encoded command within a set of comment characters at the very beginning of the web page. The command would look like:

<!--*encoded_command* -->

One possible detection method involves a brute force approach. In this method, automation would look for HTTP traffic from a web server that contains a comment at the beginning of the web page. The embedded comment would then be decoded using every possible Base64 chart to see if the text within the comments decodes into a command used by the malware. However, each "key" for this algorithm is a rearrangement of 64 characters. So, there are 64! (about 10^{89}) different keys available to the malware author for a given 64 character set. Since the key space is so large, using brute force methods to decode commands may not be practical.

The encoded "sleep " command from the previous section has an interesting property. In Figure 12, the 6-bit numeric values in step 4 are all different. A consequence of this property is that for any sequence of eight unique characters, there exists a "scrambled" chart that will decode to "sleep ". For example, suppose the command from the C&C website is "abcdefgh". The corresponding Base64 chart will need to have the mappings shown in step 5 of Figure 13. Figure 14 shows an example Base64 chart with the required mappings. Note that 56 of the numeric values have no corresponding characters because any arrangement of the remaining 56 characters can be used. If a brute force search had been performed, a chart such as the one shown in Figure 14 would eventually have been found. Constructing a scrambled chart is much faster than performing a brute force search, so the use of brute force methods is not necessary.

Step 1	‘s’ = 115, ‘l’ = 108, ‘e’ = 101, ‘e’ = 101, ‘p’ = 112, ‘ ‘ = 32
Step 2	01110011 01101100 01100101 01100101 01110000 00100000
Step 3	011100 110110 110001 100101 011001 010111 000000 1000000
Step 4	28 54 49 37 25 23 0 32
Step 5	28 = ‘a’, 54 = ‘b’, 49 = ‘c’, 37 = ‘d’, 25 = ‘e’, 23 = ‘f’, 0 = ‘g’, 32 = ‘h’

Encoded characters: “abcdefgh”

Figure 13: Encoding "sleep " with scrambled Base64 chart

Value	Char	Value	Char	Value	Char	Value	Char
0	g	16		32	h	48	
1		17		33		49	c
2		18		34		50	
3		19		35		51	
4		20		36		52	
5		21		37	d	53	
6		22		38		54	b
7		23	f	39		55	
8		24		40		56	
9		25	e	41		57	
10		26		42		58	
11		27		43		59	
12		28	a	44		60	
13		29		45		61	
14		30		46		62	
15		31		47		63	

Figure 14: An incomplete scrambled chart

Suppose the Base64 algorithm is used to encode the command "telnet". Figure 15 shows the command "telnet" encoded into the string "abcdebcf" using the scrambled chart shown in Figure 16. For this command, the 2nd and 6th characters are the same (letter 'b') and the 3rd and 7th characters are the same (letter 'c'). Notice that the letter pattern for this encoded command is different from the letter pattern for the encoded "sleep " command. The encoded "sleep " command did not have any repeated letters, but the letters 'b' and 'c' each appear twice in the encoded command for "telnet".

ASCII characters: "telnet"

Step 1 't' = 116, 'e' = 101, 'l' = 108, 'n' = 110, 'e' = 101, 't' = 116

Step 2 01110010 01100101 01101100 01101110 01100101 01110010

Step 3 011100 100110 010101 101100 011011 100110 010101 110010

Step 4 28 38 21 44 27 38 21 50

Step 5 28 = 'a', 38 = 'b', 21 = 'c', 44 = 'd', 27 = 'e', 38 = 'b', 21 = 'c', 50 = 'f'

Encoded characters: "abcdebcf"

Figure 15: Encoding "telnet" with scrambled Base64 chart

Value	Char	Value	Char	Value	Char	Value	Char
0		16		32		48	
1		17		33		49	
2		18		34		50	f
3		19		35		51	
4		20		36		52	
5		21	c	37		53	
6		22		38	b	54	
7		23		39		55	
8		24		40		56	
9		25		41		57	
10		26		42		58	
11		27	e	43		59	
12		28	a	44	d	60	
13		29		45		61	
14		30		46		62	
15		31		47		63	

Figure 16: Partial scrambled Base64 chart used to encode "telnet"

Is there a chart that will decode the command “abcdebcf” to “sleep ”? Figure 17 shows the character mappings that would be required by the scrambled chart. Step 5 of the figure shows that the character ‘b’ must map to two different numeric values, 23 and 54. The character ‘c’ must also map to two different numeric values, 0 and 49. Since each letter in a Base64 chart can map to only one numeric value, the mappings in step 5 of Figure 17 will not create a valid Base64 chart. Therefore, there is no chart that will decode “abcdebcf” to “sleep ”. This shows that an encoded command matching a specific letter pattern may or may not decode to a specific command recognized by the infected host.

ASCII characters: “sleep ”

Step 1	‘s’ = 115, ‘l’ = 108, ‘e’ = 101, ‘e’ = 101, ‘p’ = 112, ‘ ‘ = 32
Step 2	01110011 01101100 01100101 01100101 01110000 00100000
Step 3	011100 110110 110001 100101 011001 010111 000000 1000000
Step 4	28 54 49 37 25 23 0 32
Step 5	28 = ‘a’, 54 = ‘b’, 49 = ‘c’, 37 = ‘d’, 25 = ‘e’, 23 = ‘b’, 0 = ‘c’, 32 = ‘f’

Encoded characters: “abcdefgh”

Figure 17: Encoding "sleep " with scrambled Base64 chart

If the set of commands for the malware is known, a set of letter patterns can be constructed for each of the commands. The set of patterns can be used to inspect network traffic for possible C&C commands within the comments of a web page. For example, suppose the malware uses only two commands: “sleep …” and “telnet …”. Automation may be able to detect variants of the malware that use different scrambled charts by first looking for web traffic from the server with a comment at the beginning of the web page. The automation can then inspect the first eight characters of the (potentially) encoded command. If all eight letters are distinct, the website may be sending a “sleep …” command to the infected host. If the 2nd and 6th characters match, the 3rd and 7th characters match, and the remaining characters are distinct, the website may be sending the “telnet …” command to the client. But, suppose the first seven characters are distinct and the 8th character is the same as the 1st character. This pattern does not match the pattern for “sleep…” or “telnet…” commands. So, no scrambled chart exists that will decode the eight letters to either of the commands used by the malware. In this way, the pattern of the first eight characters within the comment can be used to determine if the network activity is possibly related to this malware.

It is important to note that a pattern match may be found for traffic that is not related to a variant of the malware. For example, the character string “flood ” has the same characteristics as the string “sleep ”. It will be encoded into a string of eight distinct characters, as shown in Figure 18. Since “flood “ is not one of the commands that the malware responds to, it’s possible for a string of eight distinct letters to decode to a string of six letters that is not related to C&C activity for this malware.

ASCII characters: “flood ”

Step 1	‘f’ = 102, ‘l’ = 108, ‘o’ = 111, ‘o’ = 111, ‘d’ = 100, ‘ ‘ = 32
Step 2	01100110 01101100 01101111 01101111 01100100 00100000
Step 3	011001 100110 110001 101111 011011 110110 010000 1000000
Step 4	25 38 49 47 27 54 16 32
Step 5	25 = ‘a’, 38 = ‘b’, 49 = ‘c’, 47 = ‘d’, 27 = ‘e’, 54 = ‘f’, 16 = ‘g’, 32 = ‘h’

Encoded characters: “abcdefgh”

Figure 18: Encoding "flood " with scrambled Base64 chart

Conclusions

A cryptographic algorithm that uses an identical key for both encryption and decryption is classified as a symmetric algorithm (Schneier 1996). If the Base64 algorithm is viewed as an encryption algorithm, and the Base64 chart is viewed as the encryption key, the Base64 algorithm can be classified as a symmetric-key encryption algorithm. The reason Base64 encoded data is trivial to decode is because a well-known key is used. However, if the letters in the chart are rearranged, a nonstandard key would be used to encode the data. The chart with the rearranged letters would then be needed to decode the encoded data. The malware author uses this technique to obfuscate the commands generated by the C&C website.

The previous section showed that a Base64 chart can be constructed from a sample of plain text and cipher text. Therefore, the Base64 algorithm is susceptible to a known-plaintext attack (Schneier 1996). Certain algorithms (such as DES) were designed so that brute force analysis is required to deduce the key even when a known-plaintext attack is possible (2003 Mao).

Since the "key" for this type of encryption is susceptible to a known-plaintext attack, the malware author is probably using this algorithm primarily for obfuscation. It would be impossible to write IDS signatures for every possible key because the key space is so large. Therefore, the malware author can easily evade IDS signatures by creating variants that make use of different keys.

The algorithm is also simple to implement and requires no linking to external libraries. If the malware author chose to use an encryption algorithm such as AES, he would be required to link to an external library. If the code is dynamically linked, and the malware infected a computer that did not have a copy of the library, the malware would not work properly. If the code is statically linked, the size of the malware will increase because the malware will include copies of any subroutines in the library used by the malware. By using the Base64 algorithm with a scrambled chart, the size of the malware can be kept small, while ensuring that the malware executes reliably on an infected host.

References

1. (2006 June 14). Base64 Encoding – uCertify Articles. Retrieved August 31, 2010 from uCertify Web site: http://www.ucertify.com/article/base64-encoding.html
2. (2006 October). The Base16, Base32, and Base64 Data Encodings. Retrieved September 20, 2010 from the Internet Engineering Task Force Web site: http://tools.ietf.org/html/rfc4648
3. Schneier, B (1996). Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons Inc.
4. Scott, C, Wolfe, P, & Hayes, B (2004). Snort for Dummies. Hoboken: Wiley Publishing Inc.
5. Mao, W (2003 July 25). Modern Cryptography: Theory and Practice. Upper Saddle River: Prentice Hall.
6. Tschabitscher, H (2010) Base64 Encoding Table. Retrieved September 15, 2010, from About.com Web site: http://email.about.com/od/emailbehindthescenes/l/blbase64enctabl.htm

Appendix 1: Array with Standard Base64 Chart

chart[0]		chart[32]	62	chart[64]		chart[96]	
chart[1]		chart[33]		chart[65]	0	chart[97]	26
chart[2]		chart[34]		chart[66]	1	chart[98]	27
chart[3]		chart[35]		chart[67]	2	chart[99]	28
chart[4]		chart[36]		chart[68]	3	chart[100]	29
chart[5]		chart[37]		chart[69]	4	chart[101]	30
chart[6]		chart[38]		chart[70]	5	chart[102]	31
chart[7]		chart[39]		chart[71]	6	chart[103]	32
chart[8]		chart[40]		chart[72]	7	chart[104]	33
chart[9]		chart[41]		chart[73]	8	chart[105]	34
chart[10]		chart[42]		chart[74]	9	chart[106]	35
chart[11]		chart[43]		chart[75]	10	chart[107]	36
chart[12]		chart[44]		chart[76]	11	chart[108]	37
chart[13]		chart[45]		chart[77]	12	chart[109]	38
chart[14]		chart[46]		chart[78]	13	chart[110]	39
chart[15]		chart[47]	63	chart[79]	14	chart[111]	40
chart[16]		chart[48]	52	chart[80]	15	chart[112]	41
chart[17]		chart[49]	53	chart[81]	16	chart[113]	42
chart[18]		chart[50]	54	chart[82]	17	chart[114]	43
chart[19]		chart[51]	55	chart[83]	18	chart[115]	44
chart[20]		chart[52]	56	chart[84]	19	chart[116]	45
chart[21]		chart[53]	57	chart[85]	20	chart[117]	46
chart[22]		chart[54]	58	chart[86]	21	chart[118]	47
chart[23]		chart[55]	59	chart[87]	22	chart[119]	48
chart[24]		chart[56]	60	chart[88]	23	chart[120]	49
chart[25]		chart[57]	61	chart[89]	24	chart[121]	50
chart[26]		chart[58]		chart[90]	25	chart[122]	51
chart[27]		chart[59]		chart[91]		chart[123]	
chart[28]		chart[60]		chart[92]		chart[124]	
chart[29]		chart[61]		chart[93]		chart[125]	
chart[30]		chart[62]		chart[94]		chart[126]	
chart[31]		chart[63]		chart[95]		chart[127]	

Appendix 2: Array with Scrambled Base64 Chart

chart[0]		chart[32]		chart[64]		chart[96]	
chart[1]		chart[33]		chart[65]	0	chart[97]	35
chart[2]		chart[34]		chart[66]	1	chart[98]	36
chart[3]		chart[35]		chart[67]	2	chart[99]	37
chart[4]		chart[36]		chart[68]	44	chart[100]	8
chart[5]		chart[37]		chart[69]	45	chart[101]	9
chart[6]		chart[38]		chart[70]	3	chart[102]	10
chart[7]		chart[39]		chart[71]	4	chart[103]	11
chart[8]		chart[40]		chart[72]	5	chart[104]	12
chart[9]		chart[41]		chart[73]	6	chart[105]	13
chart[10]		chart[42]		chart[74]	7	chart[106]	14
chart[11]		chart[43]	62	chart[75]	16	chart[107]	15
chart[12]		chart[44]		chart[76]	17	chart[108]	38
chart[13]		chart[45]		chart[77]	18	chart[109]	39
chart[14]		chart[46]		chart[78]	19	chart[110]	47
chart[15]		chart[47]		chart[79]	20	chart[111]	48
chart[16]		chart[48]	26	chart[80]	21	chart[112]	49
chart[17]		chart[49]	27	chart[81]	40	chart[113]	50
chart[18]		chart[50]	28	chart[82]	41	chart[114]	51
chart[19]		chart[51]	29	chart[83]	42	chart[115]	52
chart[20]		chart[52]	30	chart[84]	43	chart[116]	53
chart[21]		chart[53]	57	chart[85]	46	chart[117]	54
chart[22]		chart[54]	58	chart[86]	22	chart[118]	55
chart[23]		chart[55]	59	chart[87]	31	chart[119]	56
chart[24]		chart[56]	60	chart[88]	32	chart[120]	23
chart[25]		chart[57]	61	chart[89]	33	chart[121]	24
chart[26]		chart[58]		chart[90]	34	chart[122]	25
chart[27]		chart[59]		chart[91]		chart[123]	
chart[28]		chart[60]		chart[92]		chart[124]	
chart[29]		chart[61]	63	chart[93]		chart[125]	
chart[30]		chart[62]		chart[94]		chart[126]	
chart[31]		chart[63]		chart[95]		chart[127]	