Reversing a NSIS dropper using quick and dirty shellcode emulation

malcat.fr/blog/reversing-a-nsis-dropper-using-quick-and-dirty-shellcode-emulation/

Sample:

e850f3849ea82980cf23844ad3caadf73856b2d5b0c4179847d82ce4016e80ee (Bazaar, VT)

Infection chain:

Excel stylesheet -> Office equation -> Shellcode (downloader) -> NSIS installer -> Shellcode (stage 1) -> Shellcode (stage 2) -> Lokibot

Tools used:

Malcat, Speakeasy emulator

Difficulty:

Easy

The Excel document

The sample we are about to dissect today is an OpenXML Excel document which came as email attachment. The malicious document is very similar to the one we did analyze in our <u>previous blog post</u>: an encrypted OpenXML Excel document embedding an Equation object exploiting CVE-2018-0798. The same author is most likely behind this document as well, they just updated the bait picture:



Figure 1: Excel sheet baiting the user to deactivate safe mode

We won't go through the exploit shellcode extraction and decryption process again since the procedure is exactly the same (see <u>here</u>, shellcode offset is also 0×50). The exploit is again a downloader, downloading from the following url:

hxxp://103.153.79.104/windows10/csrss.exe

At the time of the analysis, the file is still online. But this time, we don't get a DBatLoader instance, but a NSIS installer instead. So let us fetch the file and have a look at the installer.

NSIS installer

The file csrss.exe is a 418KB PE file of sha256

291df8186e62df74b8fcf2c361c6913b9b73e3e864dde58eb63d5c3159a4c32d (Bazaar, <u>VT</u>). A NSIS installer is nothing more than a NSIS archive appended to the NSIS PE installer. The file format of the archive, while not very documented, is relatively simple as we will see.

NSIS archive

A NSIS archive is composed of a small NSIS header followed by the archive content. The header does not contain a lot of information:

FirstHeader:		
Flags:		<pre>// some installation</pre>
flags		
Signature:	0xdeadbeef	// NSIS archive start
magic		
Magic:	"NullsoftInst"	// also magic
InstallerSize:	0x6244	// unpacked size of
the setup script		
ArchiveSize:	0x5e12e	// size of the
archive		

Like you can see, it does not tell us a lot. Directly following the headers come the "files". I say "files" because they don't really have names, it is more like a list of data bytes or buffers. The files are compressed, and can be stored using two modes:

- the *solid* mode: archive content is a single compressed stream. The unpacked stream is a sequence of N buffers, where each buffer is prefixed by a DWORD telling the size of the buffer.
- the *non-solid* mode: archive content is a sequence of N compressed streams, one for each file. Each compressed stream is prefixed by a DWORD telling the size of the stream.

There is sadly no flag in the header telling us which mode is used, this information is hardcoded inside the NSIS installer executable. The only solution there is trial and error: if the start of the archive starts with a DWORD which could be a size, then it's most likely the *non-solid* mode. If it looks like a compression header, then it's most likely the solid mode. And regarding compression, NSIS supports three compression algorithms:

- LZMA (without headers)
- Zlib
- A custom bzip2 compression algorithm

Malcat supports NSIS files using both solid and non-solid mode for the Zlib and LZMA compression methods, but lacks support for bzip2, since the compression algorithm is custom. But since it's also the least used one, it's not really a big deal. The NSIS archive we are looking at is a solid LZMA archive, so unpacking it is no issue. Like for most archive formats, Malcat lists found files in the upper left corner of the screen, under the *Virtual File System* tree. Double-clicking on a file opens it inside Malcat.

files (4 + 0)		
Name	Size	
✓ ♣ Virtual File System		
SETUP	25156	
d54hdan9c9ebsx	216843	Figure 2: Content of the archive
🗋 lognp	5245	
FILE3-#0x36390	294400	
 Carved Files 		

The first file is always the installer setup script, followed by user-provided files and/or installer plugins. As you can see, Malcat did give name to some of the files (all but the last one) which somehow contradicts what I said before. But these names have been recovered by reversing the SETUP script, and there is no guarantee that it is the real name for these files. Even worse, a buffer in the archive can be extracted under different names on the local filesystem, so don't trust these names 100%.

The SETUP script

The first thing to look at when reversing a NSIS installer is the setup script. NSIS scripts are a bunch of sections and assembly code written for the NSIS virtual machine. The NSIS VM architecture is relatively simple:

- Every instruction is encoded on 7 DWORDs: first DWORD is for the opcode (about 70 different opcodes) and the other 6 DWORD encode arguments
- Depending on the opcode, arguments can be either:
 - a register (up to 31 registers): \$0 .. \$9 , \$R0 .. \$R9 or one of 11 specific registers like \$EXEPATH or \$CMDLINE (some are read-only, so more like constants)
 - a global variable: \$var0 .. \$varN
 - an integer, signed or unsigned. It can also be an offset into the code section for jump-like opcodes
 - a string, more precisely an index into the Strings section of the setup script
- Strings themselves can be somewhat complex to parse/interpret:
 - there are 3 NSIS versions: ansi, park (a mix between ansi and unicode) and unicode. Each version encodes strings differently. There is sadly no flag telling you which version is used.
 - strings can contains any of 4 special opcodes: skip , shell , var or lang
 - strings can include reference to system paths, variables or other strings, e.g.
 "open {\$INSTDIR}\rampage\goodie\noticeably.tif"

Luckily for us, the full edition of Malcat features a NSIS disassembler / decompiler, so let us jump directly to the entry point of the script (**Ctrl+E**) and have a look at the **OnInit** method:

	;======================================	ONINIT
	OnInit() {	
Entries 000003ea0:	05000002102000000000000000000	Call sub_4974() ↓1 ; →
Entries 000003ebc:	000000000000000000000000000000000000000	SetFlag 0xd, ""
Entries 000003ed8:	0B000000B10F0000010000000000	CreateDirectory "{\$INSTDIR}", 0x1
Entries 000003ef4:	140000009000005B50F0000000	ExtractFile 0x5000090, "d54hdan9c9ebsx", 0x0, 0xfb612500, 0x1d8068e, 0xffffffd9
Entries 000003f10:	140000009000005C40F00000F4F	ExtractFile 0x5000090, "lognp", 0x34f0f, 0xfb612500, 0x1d8068e, 0xffffffd9
Entries 000003f2c:	0500000210200000000000000000	Call sub_4974() ↓1 ; →
Entries 000003f48:	1400000091000005CA0F00009063	ExtractFile 0x5000091, "{\$PLUGINDIR}\dwksh.dll", 0x36390, 0xffffffff, 0xffffffff, 0xfffffffd9
Entries 000003f64:	OD000000000000000000000000000000000000	SetFlag 0xd, ""
Entries 000003f80:	2C00000CA0F0000D80F0000000	RegisterDLL "{\$PLUGINDIR}\dwksh.dll", "sdvffryp", "", 0x0
Entries 000003f9c:	0D0000002000007F040000000	SetFlag 0x2, 0x4f7b
Entries 000003fb8:	0D000000B000002A070000000	SetFlag 0xb, 0x5226
Entries 000003fd4:	1F000000C000001000000000	Pop \$R2
Entries 000003ff0:	0F0000009000000B000000000	GetFlag \$9, 0xb
Entries 00000400c:	1F000000DE0E0000000000000000	Push "{\$R3}"
Entries 000004028:	1F000000000000000000000000000000000000	Swap \$0
Entries 000004044:	1F000000D000001000000000	Pop \$R3
Entries 000004060:	2E000000E10F0000011000001007	CopyFiles "{\$INSTDIR}\doohickey\guardian\slut.bmp", "{\$INSTDIR}\divorces\extract.docx", 0x710
Entries 00000407c:	24000008E0600000000000000000	SetCtlColors "{\$1}", 0x0
Entries 000004098:	1F000000381000000000000000000	Push "{\$R5}"
Entries 0000040b4:	0B000003C1000000000000000000	CreateDirectory " {\$INSTDIR }\flannel\haggling\procrastinating", 0x0
Entries 0000040d0:	300000061100006F100000000	WriteINI "ghtcbslisjklc", "ydilvvilstoe", "", "{\$INSTDIR}\manic\tidings\dunk.zip"
Entries 0000040ec:	1F0000002000001000000000	Pop \$2
Entries 000004108:	0600000971000000000000000000	UpdateText "nnxkkxwxomg"
Entries 000004124:	19000000100000A310000000	Assign \$1, "gvrzioegyz", 0x0, 0x0
Entries 000004140:	2B000000E0000000000000AE10	GetDLLVersion \$R4, \$0, 0x10ae
Entries 00000415c:	3500000070000002000080CC10	RegEnumKey \$7, 0x80000002, "dytsrudxmci", "4432"
Entries 000004178:	0B000000D1000000000000000000	CreateDirectory " {\$INSTDIR}\marmalade\implausible\admonish ", 0x0
Entries 000004194:	020000000000000000000000000000000000000	Jmp loc_0

Figure 3: NSIS setup script entry point

We can see that the script does the following:

- extract the first buffer (offset header+0 in archive) to a file named d54hdan9c9ebsx
- extract the second buffer (offset header+0x34f0f in archive) to a file named lognp
- extract the third buffer (offset header+0x36390 in archive) to \${PLUGINDIR}\dwksh.dll, wherever that could be
- call dwksh.dll's exported method sdvffryp without any argument

The rest of the method seems like junk code, judging by the strings which are either random letters or picked out of dictionary. Quickly inspecting the first two files tells us that both are encrypted and/or compressed, so no quick-win there. We have to dig into the dll.

Stage 1: dwksh.dll

dwksh.dll is a 294KB 32-bits DLL of sha256

be00a655cdf4e4706bf1bbf3659d698f967cad66acdf7cda0d12f16dc0cfda3e (VT). It contains several obfuscated methods. But we reversed the setup script and know what to look for: the method sdvffryp. This methods starts by reading a local file named lognp :

<pre>sdvffryp() {</pre>			
55	push	ebp	
8BEC	mov	ebp, esp	
B8A4180000	mov	eax, 0x18A4	
E8266E0200	call	sub_10047e60() +1	; →
56	push	esi	
6A6C	push	0x6C = 'l'	
58	рор	eax	
6A6F	push	0x6F = 'o'	
668945F4	mov	[ebp-0x0C], ax	
58	рор	eax	
6A67	push	0x67 = 'g'	
668945F6	mov	[ebp-0x0A], ax	
58	рор	eax	
6A6E	push	0x6E = 'n'	
668945F8	mov	[ebp-0x08], ax	
58	рор	eax	
6A70	push	0x70 = 'p'	
668945FA	mov	[ebp-0x06], ax	
58	рор	eax	
668945FC	mov	[ebp-0x04], ax	
33C0	xor	eax, eax	
668945FE	mov	[ebp-0x02], ax	
8D85DCFBFFFF	lea	eax, [ebp-0x424]	
50	push	eax	
6803010000	push	0×103	
FF1518800410	call	[kernel32.GetTempPathW]	

Figure 4: lognp file is accessed

It then seems to decrypt it in memory into an executable buffer before jumping at the beginning of the buffer (see the call eax below?). The file lognp is relatively small (5KB), it definitely looks like a shellcode.



Figure 5: decryption loop in method sdvffryp

The decryption is pretty straightforward according to the sleigh decompiler. Every byte seem to be decrypted using the following formula:

```
byte[i] = ((((byte[i] - 3) ^ 0xf2) + 0x11) ^ 0x28) - 1
```

Decrypting the lognp file should be a piece of cake. Just open the file inside Malcat, select every byte (**Ctrl+A**) and open the transform dialog (**Ctrl+T**). There you can chose the **custom** arithmetic transform which allows you to transform sequence of bytes/words/dwords using a custom python expression. Just paste the equation above, replacing byte[i] by value and *voila*, you've just decrypted the second stage.

arithmetic ^	Infos					
SAC add	custom	operator defined using user python cod	e			
S _{AC} custom	works of	on bytes, word, dword or qword in lsb or i	msb mode			
SAC div	* index:	the 0-based index of the current byte/w	ord/dword/gword in the buffer			
S _{AC} mul						
SAC neg	(defined	d in C:\malcat\data\transforms\arithmetic	c.py)			
SAC SUD						
binary	Paramete	ers				
SAC add8		0.0			0x147D	•
S _{AC} and	address:		v	size:	01470	•
SAC OF						
s _{AC} subo	width:	byte				~
s _{AC} XOI						
	endiann	ness: Isb				~
⇒ACTOL						
		def operation(value:int, index:int):				~
		return ((((value - 3) ^ 0xf2) + 0x11) ^ 0x	(28) - 1			
Ac D22 compress						
Ac b22 decompress						
	code:					
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT						
Sc Izma compress						
Time compress Jac Izma decompress Jac Izma compress Jac Izma compress						~
Image: compress Image		<				>
Sc Izma compress Sc Izma decompress Sc Izo compress Sc Izo decompress Sc Office RLE → zith compress		<				>
Lizna compress Lizna compress Lizna decompress Lizna decompress Lo compress Coffice RLE Zib decompress Zib decompress	Preview	<				>
Science Science	Preview	< 46 15 00 ED ED A2 64	49 6C 49 AD 9C 9F B0 6C D	2 F∞ ÝÝódIlI	¦£∱ 1Ê	>
	Preview	< 46 15 00 ED ED A2 64 3D ED D8 80 2D CF D8	49 6C 49 AD 9C 9F B0 6C D DC B2 4D D8 B0 2D CF D8 D	2 F∞ ÝÝódIlI C =ÝÏڀ=¤Ï _ M	iff lÊ I¤ĭ	>
Izma compress Izma compress Izo decompress Izo decompress Izo decompress Izo decompress Ib compress Ib compress Ib decompress	Preview	46 15 00 ED ED A2 64 3D ED D8 80 2D CF D8 82 45 6C D2 35 ED 40	49 6C 49 AD 9C 9F B0 6C D DC B2 4D D8 B0 2D CF D8 D B2 39 05 ED ED ED 6C D2 4	2 F∞ ÝÝódIlI C =ÝI ¤I∎M 1 ∎ElÉSÝ@∎♦	¦£f_1Ê ΤÎ _ ÝYÝ1ÊA	>
c izma compress c izma compress c izo decompress c izo decompress c zlib compress c zlib compress c zlib decompress c zlib decompress c rypto (block) c aes decrypt c aes encrypt blowfish decrypt	Preview	46 15 00 ED ED A2 64 3D ED D8 80 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD C6 ED 41 C6 42 10 C6 42 10 C6	49 6C 49 AD 9C 9F BØ 6C D DC B2 4D D8 BØ 2D CF D8 D B2 39 05 ED ED ED 6C D2 4 6A B2 35 9D 45 3A ED ED E	2 F∞ ÝÝódIlI C =ÝI∭=¤I_¶ 1 ∰ElÊSÝ@∰9# D ÝÞ⊤ÚÞ⊤¦∋∰5 C = -	i£f 1Ê Î -¤ I ÝYÝ1ÊA ØE:ÝYÝ	>
Larna compress Larna compress Lac compress Lo decompress Lo decompress Lo decompress Lo decompress Lib decompress crybt (block) aes encrypt blowfish decrypt	Preview	<pre>46 15 00 ED ED A2 64 30 ED D8 B0 2D FF B8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8</pre>	49 6C 49 AD 9C 9F 80 6C D DC 82 4D 88 80 2D CF 08 D B2 39 05 ED ED ED 6C D2 4 A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 DD 6A 82 35 9	2 F∞ ÝÝódIlI C =ÝI =±i⊒M I ≣Elšý@9 D ÝP⊤UPT¦j≣5 D f≣∞fólP⊤ÚP	¦£f lê Î -≭I∎ ÝÝ¥1ÊA ØE:ÝÝÝ T¦j∎50	>
Larna compress Larna compress Larna decompress Lao decompress Lo decompress Lo decompress Lab decompress Lib decompress crypt (block) Lae sencrypt Lowfish decrypt Lowfish decrypt Lowfish decrypt	Preview	<pre>46 15 00 ED ED A2 64 30 ED D8 80 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8</pre>	49 6C 49 AD 9C 9F 80 6C D DC 82 4D D8 80 2D CF D8 D B2 39 05 ED ED ED 6C D2 4 6A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 DD 6A 82 35 9	2 F∞ ÝÝódIlI C =ÝĨ∷=≭Ĩ ⊒ M 1 ■ ElÊ\$Ý@94 D ÝÞ┬ÚÞ;¦3 ■ 5 D f ■ ∞fó↓Þ┬ÚÞ	i£ƒ lê ĭ -¤ĭ ýýýlê 80:ýýý T¦j∎50	>
Zima compress Zima compress Zizo decompress Zizo decompress Zizo decompress Zib compress Zib compress Zib decompress crypto (block) Aeas decrypt Solowfish decrypt Solowfish decrypt Solowfish encrypt Solowfish encrypt Solowfish encrypt Solowfish encrypt Solowfish encrypt Solowfish encrypt	Preview	<pre>46 15 00 ED ED A2 64 30 ED D8 80 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8</pre>	49 6C 49 AD 9C 9F 80 6C D DC 62 40 D8 80 2D CF D8 D B2 39 05 ED ED ED 6C D2 4 6A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 DD 6A 82 35 9	2 F∞ ÝÝódIlI C =ÝÏ≣-¤Ï.∎M 1 ■ElÊSý@9+ D ÝP⊤ÚP⊤¦J∎S D f∎∞fólP⊤ÚP	i£f lê ĭ -¤ĭ ýýýlêA ØE:Ýýý T¦j∎5Ø	>
Lizna compress Lizo compress Lizo compress Lizo compress Lizo compress Lizo compress Lib decompress crypto (block) Lea es encrypt blowfish decrypt Jobowfish decrypt Jobowfish encrypt Jobowfish Jo	Preview	<pre>46 15 00 ED ED A2 64 30 ED D8 80 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8 E9 D8 07 00 00 55 88</pre>	49 6C 49 AD 9C 9F B0 6C D C 82 4D D8 B0 2D CF D8 D B2 39 05 ED ED ED 6C D2 4 6A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 DD 6A 82 35 9 EC 83 EC 40 53 56 57 83 6	2 F∞ ÝÝódIlI C =ÝĨ∭-¤Ĩ∰M 1 ∰ElÊSÝ@∰9+ D ÝP⊤ÚP⊤¦3∰S D f∰∞fóiÞ⊤ÚP 5 ÚĨ• Uĭvâý	;£ƒ 1Ê Ĭ -¤Ĭ ÝÝÝ1ÊA ØE:ÝÝÝ T¦j∎50 ØSVWāe	>
Lizna compress Lizna compress Lizo compress Lizo decompress Lizo decompress Lizo decompress Lib	Preview	<pre>< 46 15 00 ED ED A2 64 3D ED D8 80 2D CF 08 82 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8 E9 D8 07 00 00 55 8B F0 00 0F 57 C0 66 0F</pre>	49 6C 49 AD 9C 9F 80 6C D C 82 4D D8 80 2D CF D8 D 82 39 05 ED ED ED 6C D2 4 6A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 DD 6A 82 35 9 EC 83 EC 40 53 56 57 83 6 13 45 E0 0F 57 C0 66 0F 1	2 F⇔ ÝÝódIll C =ÝÏ -¤I ∭ I ≣ElÉSÝ@ 9+ D ÝP⊤ÚP⊤¦J≣S D f∎∞főiÞ⊤ÚP 5 Úľ• Uïýâý 3 - owl+fo‼EÓ	i£f 1Ê Ĭ -¤Ĭ ÝÝÝ1ÊA ØE:ÝÝÝ T¦j≣5Ø @SVWâe ou L+Ge‼	~
Larma compress Larma compress Larma decompress La compress Lo decompress Lo decompress Lib decompress Lib decompress Lib decompress Lib decompress Lo decompress	Preview	<pre>< 46 15 00 ED ED A2 64 30 ED D8 80 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8 E9 D8 07 00 00 55 8B F0 00 0F 57 C0 66 0F 45 E8 83 65 F8 00 C7 </pre>	49 6C 49 AD 9C 9F 80 6C D DC 82 4D D8 80 2D CF D8 D 82 39 05 ED ED ED 6C D2 4 6A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 DD 6A 82 35 9	2 F∞ ÝÝódIll C = ÝI = ±i ∎M D ÝÞ⊤jðgo D ÝÞ⊤jðgo 5 ÚI = Uiýây 3 - ow ⊑oleo 4 Epãe° ÅE°(;£f 1Ê Î -≭I ÝÝÝ1ÊA DE:ÝÝÝ T¦j∎5Ø @SVWãe @SVWãe aeg	>
Sector Sector	Preview	<pre>46 15 00 ED ED A2 64 30 ED D8 60 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8 F0 00 0F 57 C0 66 0F 45 E8 83 65 F8 00 C7 00 FF 75 0C FF 75 10</pre>	49 6C 49 AD 9C 9F 80 6C D DC 82 4D 08 80 2D CF 08 0 82 39 05 ED ED ED 6C 02 4 6A 82 35 9D 45 3A ED ED E C2 E9 E8 C2 0D 6A 82 35 9	2 F∞ ÝÝódIlI C =ÝĨ = xĭ_=M1 1 ■ElÊsY@9 9 0 ÝÞ⊤ÚÞr¦3∎5 D f∎∞fóiÞ⊤ÚÞ 5 ÚĨ• Uĭyâý 3 - oWifolEô 4 Epàe° ĀE°(0 u♀u+IE°	iff lê Î -¤ I ©SVWâe ow Lfo‼ âe¶ PP ²	~
Sc. Erma compress Lzma decompress Lzo decompress Coffice RLE Cibi compress Lib decompress Lib decompress Lib decompress Lib decompress Labortic block) Labortic block) Labortic block Labo	Preview	<pre>46 15 00 ED ED A2 64 30 ED D8 80 2D CF D8 B2 45 6C D2 35 ED 40 ED E8 C2 E9 E8 C2 DD 66 B2 15 66 A2 19 E8 F0 00 0F 57 C0 66 0F 45 E8 83 65 F8 00 C7 00 FF 75 0C FF 75 10 89 45 D8 89 55 DC FF</pre>	49 6C 49 AD 9C 9F 80 6C D DC 82 4D D8 80 2D CF D8 D B2 39 05 ED ED ED 6C D2 4 6A 82 35 9D 45 3A ED ED ED C2 E9 E8 C2 DD 6A 82 35 9 EC 83 EC 40 53 56 57 83 6 13 45 E0 0F 57 C0 66 0F 1 45 FC 28 00 00 08 36 5F 80 45 F8 50 E8 FD 00 00 83 57 86 5F 85 0 85 45 78 5	2 F∞ ÝÝódIlI C =ŸĨ∷≭I_M 1 ElÊ\$Y@9=5 D ÝPŢÚPŢ13 5 ÚĨ• Uïýâý 3 - oklfo‼to 4 Epêe°Ã₽°(0 u♀ u+ĩ€° 0 ëEĨëU∎u♀	i£f lê ĭ -¤ĭ YýýlEA DE:ÝýÝ T¦j∎50 @SVWâe owlfo‼ ae¶ PP² u⊳iE°P	>

Figure 6: decrypting the shellcode

For the lazy readers, you can download the decrypted lognp file here (password: infected).

Stage2: obfuscated shellcode

Analyzing the shellcode

The lognp file, once decrypted, does not appear to be in any known file format. But the first byte is E9, which is a jump in the x86 architecture and is very typical for shellcode prologs. So before starting the analysis, we will have to tell Malcat two things:

- the architecture used: x86 in our case. This can be set using the dropdown menu in the status bar
- the entry point of the shellcode, which is at address 0 in our case. We just have to define a new function start at this address using the context menu in disassembly mode (F3)

After this, Malcat is smart enough to recover most of the shellcode's CFG using its usual set of analyses. Following the control flow, we quickly arrive in the function sub_7dd which contains interesting patterns:

- the string d54hdan9c9ebsx (one of the NSIS archive's file names) is pushed on the stack at address 0x8eb
- something like a CreateFileA API call is performed at address 0x989 (the constant 0x80000000 is most likely for GENERIC_READ). If we wanted to be sure, we would have to emulate the API lookup function at address 0x776, but it looks like safe assumption.
- soon after, the function sub_a01 gets called. Decompiling this functions reveals something similar to a decryption loop

The whole process can be retraced in the animated GIF below:

Walcat Professional	- D:\malware\demo\nsis\stage3	.shellcode.decrypted				- 0	×
File Edit Analysis View	Help					# 🔚 🛢 Aa 🛹	∎ ⊕
	► File infos	4					^
	File name: File size: Type: MD5: SHA1: SHA256: TLSH:	5 stage3.shellcode.decrypted 5245 bytes (5KB) ? 6575e68929f01725daf9d566823a 7154636af6c1cd4e93ab0ec75747 48d799fb67cbd2e6a9e0a4c7f808 05b1b92fec4552ef6353ac66e59b	1f94 8f3825a5f13b 5c6f5fea4081c05c829496de68c 2b38779fb71432195042fa38ac0	0dea907a6 577045ca6d124a3			
	<pre>- DVmakwaretdemokniskistage3.shellcode.decrypted Help tode.decrypted - File infos</pre>						
	► Signatures				► Check online	intelligence	
(5KB)	► Anomalies					► Run CAPA	
	► Report						
Legend :							
R Hdr	-						
RW Data	a						
RX Ind	ex						
🗮 RWX 🛛 Rsr	c						
Dbg							
Deleting analysis 3		□ _ℝ 0x00000000 (:0)	auto	∽ no CPU	→ 458 r	ns	✓

Figure 7: locating the decryption function inside the shellcode

The code of the decryption function is given below. It is obviously obfuscated, and sadly it would not be immediate to reimplement it in python in Malcat. So we will have to find an alternative. Since the decryption function prototype is very simple (it just needs a pointer to the buffer and the buffer size) and is without side effects, why not give emulation a go?

```
BYTE* sub_a01(BYTE* buffer, uint4 size) {
    uint1 uVar1;
    char cVar2;
    uint1 uVar3;
    uint4 i;
    i = sub_0;
    while (i < size) {</pre>
        uVar3 = i;
        uVar1 = -uVar3 - ((*(buffer + i) >> 1 | buffer[i] << 7) - 0x40 ^ 0xf2);
        uVar1 = -uVar3 - (uVar1 >> 7 | uVar1 * '\x02');
        uVar1 = -uVar3 - (((uVar1 >> 3 | uVar1 * ' ') ^ uVar3) - uVar3 ^ 0x9c) ^
0xd6;
        cVar2 = ~((uVar1 >> 7 | uVar1 << 1) + 0x34 ^ 0x87) - 0x10;
        uVar1 = ~(((-cVar2 >> sub_5 | cVar2 * -8) ^ 0x1d) + 0xac) ^ 0x5e;
        uVar1 = -(((0x99 - ((uVar1 >> 2 | uVar1 << 6) + 0x49) ^ 0xa0) + 0x30 ^ 0x34)
+ uVar3);
        uVar1 = (-uVar1 >> 6 | uVar1 * -4) - uVar3 ^ uVar3;
        uVar1 = (-uVar1 >> 2 | uVar1 * -0x40) + 0x93;
        uVar1 = (-((((uVar1 >> sub_5 | uVar1 * '\b') - 0x2e ^ 7) + 0xd ^ 0x96) +
0x31) ^ 0x73) + uVar3;
        uVar1 = -uVar3 - ((uVar1 >> 2 | uVar1 * '@') + 0x61) ^ uVar3;
        uVar1 = ~((uVar1 >> 3 | uVar1 << sub_5) ^ uVar3);
        uVar1 = (uVar1 >> 7 | uVar1 << 1) + uVar3 ^ 0x2e;
        uVar1 = ~(~((uVar3 - (~(~(-(0xbc - ((uVar1 >> 6 | uVar1 << 2) - uVar3) ^
0x1e) ^ 0xc5) ^ 0x46) ^ 0xc1) ^ 0x4c) +
                   uVar3) ^ 0x4d) + 0x4c ^ uVar3;
        uVar3 = 0x2d - (-((uVar1 >> 3 | uVar1 << sub_5) + uVar3) \land 0x43);
        buffer[i] = (uVar3 >> 7 | uVar3 * '\x02') + 0x15;
        i = i + 1;
    }
    return buffer;
}
```

Emulating the decryption function

To emulate shellcodes, Malcat comes bundled with a script named speakeay_shellcode.py which emulates shellcodes using the <u>Speakeasy emulator</u>. Note that Speakeasy is not bundled with Malcat, you will have to install the python package yourself (and if you are running Malcat under Windows, be sure to check Use system python interpreter in the options).

Patching lognp

Before emulating anything, we need to solve a problem: the data to decrypt (d54hdan9c9ebsx) is not embedded in the lognp shellcode, it is read from the filesystem using CreateFileA. So emulation is likely to fail. How are we going to solve this issue?

There is the clean way: we could hook the CreateFileA/ReadFile APIs in speakeasy and intercept the call to give back the content of the file d54hdan9c9ebsx.

But there is also the dirty way: we could patch the decrypted **lognp** shellcode in order to embed the content of d54hdan9c9ebsx in the shellcode space and patch the shellcode entry point to perform a call to the decryption function with the right parameters. Of course we will chose the *dirty* way. It is not only way faster, it is also more fun.

Here is how to proceed:

- First open a copy of the decrypted lognp shellcode in Malcat with extra space at the end of the file (File > Open Copy of File). The file d54hdan9c9ebsx is 216843 bytes big, we'll append 300KB just to be sure.
- 2. Copy the content of the file d54hdan9c9ebsx in the clipboard: in a second Malcat instance, open d54hdan9c9ebsx and then hit *Ctrl+A* followed by *Ctrl+C*
- Paste the copied data after the shellcode in the first Malcat instance, let's say at address 0x2000 to make it easy to remember
- 4. Enter disassembly view (F3) and go to the shellcode's entry point at address 0

Malcat does not (yet) support assembling your own instruction, so we will need to manually edit the machine code. Click on any hexa byte in disasm mode and enter edit mode (**Insert** key). We need to assemble the following code:

- Push the second parameter which is the size of the buffer to decrypt (216843 = 0x34F0B). push uint32 is assembled using 0x68 + LSB-encoded uint32 in x86:
 68 0B 4F 03 00
- 2. Push the first parameter which is the address of the buffer to decrypt (0x2000): 68 00 20 00 00
- 3. Call to the decryption function. The call opcode is 0xE8 + signed displacement starting from the end of the call opcode. The end of our call opcode is at address
 0x000F, we want to jump to 0x0A01, so 0x0A01 0x000F = 0x09f2. We need to assemble E8 F2 09 00 00.

You can use Malcat's calculator to perform quick computation while analysing a binary, just hit **Ctrl+Space**. Internally, it uses the python interpreter, so use python syntax.

At the end, the patched shellcode should look like in the picture below. For the lazy readers, you can download the patched **lognp** file <u>here</u> (password: infected).

Walcat Professional - D:\malware\demo\nsis\stage3.shellcode.decrypted.	patched						- 0	×
File Edit Analysis View Help								
E Stage3.shellcode.decrypted.patched							🔳 🌆 🗰 🗛 🛢 Aa 🗸	● 2 9
000001f80: 00 00 00 00 00 00 00 00 00 00 00 00 0	00 00 00 00-0	0 00 00 00 0	0 00 00 00	0 00 00 00 00 00	00 00 00			^
000001fa0: 00 00 00 00 00 00 00 00 00 00 00 00 0	00 00 00 00-0	0 00 00 00 0	00 00 00 00	00 00 00 00 00	00 00 00			
000001fc0: 00 00 00 00 00 00 00 00 00 00 00 00 0	00 00 00 00-0	0 00 00 00 0	00 00 00 00	0 00 00 00 00 00	00 00 00			
000001fe0: 00 00 00 00 00 00 00 00 00 00 00 00 0	00 00 00 00-0	0 00 00 00 0	00 00 00 00	0 00 00 00 00 00	00 00 00			
; v here we pasted the content of	the file d54h	idan9c9ebsx						
000002000: F5 0F 46 5B 00 40 09 5D 3D 3E 68 CB	42 5B 3F CD-0	4 AB F8 9E F	8 8E B1 0	F 33 8A EE 9D 8D	B4 DA 45	§∘F[_@○]=>h〒B[?=♦%°×°Ä o3è⁻Øì┤┌E	
000002020: EC D1 58 E2 B2 C7 13 52 C4 E4 EB DC	1D F7 71 28-2	2 C6 04 63 8	3D F7 A2 BI	F 7E EC 13 86 58	B8 10 6D	ýÐXÔ∎Ã‼R−õÙ _∎ ⇔,	q("ã♦cì óղ~ý‼åX©⊧n	1
000002040: 89 A7 19 D3 13 CB 73 91 C6 93 86 C7	31 45 57 34-4	4 18 76 74 4	14 E5 3E 2:	1 20 30 CB C4 13	D6 A3 2B	ëº↓Ë‼ ∏ sæãôåÃ1E	W4D↑vtDÖ>!_0╦-‼İú+	
000002060: CD FF 8A BE 1C 32 DC D4 D5 4D 52 B1	7A 24 AE BC-D	08 18 1E 54 6	52 E9 4E 1	7 C7 82 FA 6A E3	39 DØ D8	= è¥L2 _E E1MR z\$	«≟I↑▲TbÚN⊉Ãé∙j09ðI	
000002080: 45 10 4C F4 CB 05 D4 04 F3 5A 16 71	88 25 4C CB-2	D FE 8A 1A 8	39 AA 6F BI	E A1 14 66 CD 68	F5 B6 DD	E►L¶ T† E ♦ %Z = qê%	L〒-■è→ë¬o¥í*f=h§Ä¦	
0000020a0: E7 8F 82 97 DA E2 31 4A 01 C3 13 71	24 8B 95 E0-3	6 8A D6 84 E	9 AA 7F 44	A C1 D0 8A B3 3F	45 B2 20	þÁéù rÖ1J@ ¦‼q\$ï	òO6èIäU¬∩J [⊥] ðè ?E	
0000020c0: B3 C9 E1 3C 59 3C 36 1F 26 D5 3E DB	99 9C 15 16-9	1 66 B1 42 6	5F 27 D1 D2	2 49 0A 86 57 1D	BA CØ 68	_∏ ß <y<6▼&1>_O£</y<6▼&1>	∞ æf∭Bo'ÐEI∎åW⇔∥ Ч	
0000020e0: 33 9F F9 0C 4B 03 6C A7 F0 4F 80 B5	5B 20 C7 F5-3	B 1F 26 3A 0	C7 D8 CB 7	A 4F 5B 79 DB A2	4F 07 9E	3f"\$K♥1º-OÇA[A§;∙&:AI _T zO[y _ óO∎×	
000002100: 27 E4 CE 5B F6 40 B3 AF EE FC AC CB	17 13 9D CD-1	.9 7D 98 9E F	8 8E B1 0	F C7 17 AF 9D 8D	74 DA 45	`õ‡[÷@ »⁻³% <u>∓</u> ⊈‼	Ø=↓}ÿ×°A_⊙A⊈»Øìt _Γ E	
000002120: EC 75 F0 E2 B2 C7 CE 52 C4 45 EB DC	1D EC 71 28-B	1 C6 E4 63 8	3D F7 A2 BI	F 5B EC 54 86 71	B8 10 6D	ýu-O∎A‡R−EU∎⇔ý	q(≣ãõcì,ó┐[ýTắq©ኑn	1
000002140: AC A8 DE 4B 13 75 8E ED 68 15 8B DC	DF 3A AB C2-E	4 20 BF 49 0	5 32 A5 2	7 37 75 7D 17 F4	9C 94 5A	%;IK‼uAYh∞ï	% _T õ ןI+2N'7u}⊉¶£öZ	
000002160: 45 7E E6 59 01 CE 0F 62 E9 4E 6E C0	8D FF 7D 35-2	9 72 FA 4E 0	8 8A 03 A	C 38 82 FA 6A E3	39 DØ D8	E∼µY©╬obUNn ⊦i	}5)r•N⊑eV%8e•j09ð1	
000002180: 27 CD 01 EC 6A BD 53 03 67 80 00 6C	B5 02 DB 69-D	3 21 1A 6F 3	34 3D 74 C	7 12 5A 76 C8 ØE	46 85 75	'=©ýj¢S♥gÇ IA⊖	iE!→o4=tA‡Zv≞5Fau	~
10000007150 70 AA 18 68 8A 06 7A AG C3 78 17 87	0x00001f80 (·1	1 56 56 76 7		A AF AF 93 63 75	V86		665 ms	
		1100)	auto		100		0051113	• .4
Malcat Professional - D:\malware\demo\nsis\stage3.shellcode.decrypted.	patched						- 0	×
File Edit Analysis View Help								
Stage3.shellcode.decrypted.patched							🔳 🐻 🗰 品 🛢 Aa 🗸	
								^
			== SUB 0 ==					
→ 2 references sub 0() {			-					
000000000: 680B4F0300	push	0x34F0B			; size of	d54hdan9c9ebsx	in bytes	
00000005: 6800200000	, push	0×2000			; file's	content address		
0000000a: E8F2090000	call	sub_a01()	↓1		; calling	decryption fun	ction	
0000000f: C3	ret							
}								
→ 1 reference loc 10:								
	lock add	[edi]	c1					
000000013: 57	nush	edi						
0000000131 S/	shl	byte ntr [e	si+0x0Fl	0x13				
00000111	inc	ehn		0/120				
000000019: E00F	loopne	loc 2a						
0000001b: 57	nush	edi						
00000001b: 57 00000001c: 575	push sh1	edi	si+0x0Fl	0x13				
00000001b: 57 00000001c: C0660F13 00000002c: 45	push shl	edi byte ptr [e	esi+0x0F],	0×13				
00000011: 57 00000001c: C0660F13 000000020: 45 000000021: E83355500	push shl inc	edi byte ptr [e ebp	esi+0x0F],	0x13				~

Figure 8: patching the shellcode

Running speakeasy

Now the only thing we have to do is to let speakeasy do its magic:

- let us define the entry point: right-click at address 0 and chose Force function start in the context menu
- run the script speakeasy_shellcode.py (Ctrl+U to run user scripts)

... and voila, Malcat should open the result in a new file. A PE file has been detected by Malcat's file format parser at address 0x2000, perfect! Just double-click the PE file under "Carved Files" to open it.

👼 Malcat Professional - SC						- 0	×
File Edit Analysis View Help							
[] files (0 + 1)	-	stage3.shellco	de.decrypted.patched	► SC		🔳 🌆 🗰 🗛 🛢 Aa 🗸	
Name	Size	000002000:	4D 5A 90 00 03	00 00 00 04 00 00 00	FF FF 00 00	MZÉ♥ ♦	^
✓ ♣ Virtual File System		000002010:	B8 00 00 00 00	00 00 00 40 00 00 00	00 00 00 00	ھ ٥	
× Carved Files		000002020:	00 00 00 00 00	00 00 00 00 00 00 00	00 00 00 00		
× ₩ PF (1)		∥000002030:	00 00 00 00 00	00 00 00 00 00 00 00	F0 00 00 00	-	
* FE #2000	106406	000002040:	0E 1F BA 0E 00	B4 09 CD 21 B8 01 4C	CD 21 54 68	រា• រា o=!©©L=!Th	
QFFL.#2000	100490	∥000002050:	69 73 20 70 72	6F 67 72 61 6D 20 63	61 6E 6E 6F	is program canno	
		000002060:	74 20 62 65 20	72 75 6E 20 69 6E 20	44 4F 53 20	t be run in DOS	
		000002070:	6D 6F 64 65 2E	0D 0D 0A 24 00 00 00	00 00 00 00	mode.♪♪∎\$	
n data (0 + 3 + 0)		000002080:	CC CD 78 FE 88	AC 16 AD 88 AC 16 AD	88 AC 16 AD	_x ∎ê¼ - ;ê¼ - ;ê¼ - ;	
> < Structures		000002090:	81 D4 95 AD 89	AC 1 AD 48 A3 48 AD	8A AC 16 AD	üEò;ë‰ = ;KúK;è‰ = ;	
✓		0000020a0:	8D A0 19 AD 89	AC 10 AD 30 32 F3 AD	8B AC 16 AD	ìá↓;ë% - ;=2%;ï% - ;	
> Itoolset (1)		∥0000020b0:	88 AC 16 AD 8C	AC 16 AD 81 D4 83 AD	89 AC 16 AD	ê%=;î%=;üEâ;ê%=;	
\rightarrow a hash (1)		∥0000020c0 :	88 AC 17 AD C7	AC 16 AD 81 D4 85 AD	99 AC 16 AD	ë‰⊉;A‰=;üEå;O‰=;	
		∥0000020d0 :	3D 32 F7 AD F3	AC 16 AD 3D 32 C8 AD	89 AC 16 AD	=2, ;%%=;=2";ë%=;	
> guid (1)		∥0000020e0 :	52 69 63 68 88	AC 16 AD 00 00 00 00	00 00 00 00	Riche%=;	
Main Search result		∥0000020+0:	50 45 00 00 4C	01 04 00 85 08 6C 57	00 00 00 00	PE L⊕♦ å∘1W	
		∥000002100:	00 00 00 00 E0	00 03 01 0B 01 0C 00	00 38 01 00	0 ♥ @ð@¥ 8@	
f(x) code (0)		∥000002110:	00 A2 08 00 00	00 00 00 DE 39 01 00	00 10 00 00	ó□ I90 ►	
DE #0000	0	∥000002120:	00 50 01 00 00	00 40 00 00 10 00 00	00 02 00 00	P© @ ► ♥	~
		II 000002130:	05 00 01 00 00	00 00 00 05 00 01 00	00 00 00 00	♥ ♥ ♥ ♥	
Deleting analysis 4	W	0X00002000 (:2000)	auto	* no CPU		471 1115	V

Figure 9: decrypted d54hdan9c9ebsx

Stage 3: Lokibot and config extraction

The last (and final) stage we get is a PE file of sha

02dee91a81652e5234414e452622307fd61e7988e82bec43665f699f805c3151 (VT).

Judging by the low entropy and the visible strings, the file does not seem to be obfuscated, good news. So which kind of malware do we face? Malcat's Yara rules already spotted one of main malware intent: stealing credentials, as we can see in the screenshot below:

Walcat Professional - D:\malware\demo\nsis\stage4.stealer									-		×
File Edit Analysis View Help										An # 1	- -
		stage4.stea	ler					00		Ad 🔷 .	₽₩
★ data (15 + 3 + 0) ★ Structures ^ ₩Z		header	► Fi	le infos							Â
₩ MZ ■ Rich ■ PE ■ OptionalHeader ■ Sections > ■ ImportTable ✓ ■ Constants > ■ toolset (1)			Fi Fi MD SH TL	le name: le size: pe: 5: A1: A1: A256: SH:	stage4.steald 106496 bytes PE 2c12200e5fa2 dff0a56355fct 02dee91a81652 5ba32a42b2a50	er (104KB) 94577cde6b4 93a1f0feea4 2e5234414e4 c030f7b74db	6f49eba01 d155e937320cee15a 52622307fd61e7988 2bb73a5b7857e7c33	e82bec43665f69 2d22c44e935245	9f805c3151 9a18215e16	1 2b7ab1	.3
> hash (1)			► Met	adata							
f(x) code (455)	1	.text (78KB)									
PasswordStealer stealer	i 🔛		0	ipile date.	2010-00-25 10	5.04.21					
program is likely to steal passwords from local			► Si	gnatures				► Check onlin	e intelli	gence	
databases/files/registry					Suspect			Other			
author malcat				stealer			compiler				
= 22/31 patterns matching				PasswordSteal	er		MSVC 2013 linke	r			
* (2 occurences) wcx_ftp.ini				Ð							
#1658a sitemanager.xml				BrowserSteale	r		MSVC_2013_rich				
#14520 VaultGetItem											
#16244 ClassicFTP\FTPAccounts											
#16926 GoFTP\settings\Connections.txt		ndata									
#16efc Software\VanDyke\SecureFX		(16KB)									
#16aca oZone3D\MyFTP\myftp.ini		(1000)									
#171f2 GHISLER\wcx_ftp.ini											
#16876 FTPShell\ftpshell.fsi		.data									
#16b32 NetDrive2\drives.dat v											
Open rule		.x									~
Entropy: 129/255		[🚽 #0 (heade	r:0)	PE	~	x86	~ 49	14 ms		⊘

Figure 10: file summary, displaying matching Yara rules

If we want to be more precise, we can use Malcat's online intelligence view (**Ctrl+I**, only for paid versions). Normally I would avoid using Virustotal to identify a malware family (because of packer reuse among threat actors). But here we are dealing with the plain text final

malware, so we should get at least *some* valid labels. In our case, it seems to be Lokibot, a simple password stealer:

👹 Malcat Pro	fessional - D:\malware\demo\nsis\stage4.s	tealer					-		×
File Edit Analy	ysis View Help		E E					Aa 🦨	
Adata (15 +	3 + 0)								
× 🐟 Structur		0	Check hash of file against online threat intelligence servi	ices			Hide unknown	Re	fresh
■M7			Intelligence source		Level	Signature			~
Rich			✓						
			CrowdStrike Falcon Static Analysis (ML)		MALICIOUS	100% matching			
Optional	IHeader		Metadefender		MALICIOUS	96% matching			
Sections	s		VirusTotal		MALICIOUS	85% matching			
> ImportTa	able		Falcon Sandbox			(not submitted)			
× = Constan	ats		✓ X JoeSandbox [NOT FOUND]						
> = toolset (1)		MalwareBazaar [NOT FOUND]						
> = hash (1)			✓ ✓ VirusTotal						
> = quid (1)	/	~	ALYac		MALICIOUS	Trojan.PWS.ZKD			
f(x) code (455))		APEX		MALICIOUS	Malicious			
PasswordStea	aler	stealer	AVG		MALICIOUS	Win32:LokiBot-A [Trj]			
			Acronis		MALICIOUS	suspicious			
			Ad-Aware		MALICIOUS	Trojan.PWS.ZKD			
program is	s likely to steal passwords from local		AhnLab-V3	MALICIOUS	Trojan/Win32.Lokibot.R270	234			
ualabases	sillesitegistiy		Antiy-AVL	MALICIOUS	Trojan/Generic.ASMalwS.1E	6B4C6			
_			Arcabit	MALICIOUS	Trojan.PWS.ZKD				
author	malcat	^	Avast		MALICIOUS	Win32:LokiBot-A [Trj]			
= 22/31 patte	erns matching		Avira		MALICIOUS	TR/Crypt.XPACK.Gen			
(2 occurrence)	ces) wcx_ftp.ini		BitDefender		MALICIOUS	Trojan.PWS.ZKD			
#1658a	sitemanager.xml		BitDefenderTheta		MALICIOUS	AI:Packer.59A658E51E			
#14520	VaultGetitem		Bkav		MALICIOUS	W32.AIDetect.malware1			
#16244	ClassicFTP\FTPAccounts		CAT-QuickHeal		MALICIOUS	Trojan.Mauvaise.SL1			
#16926	GOF I P(settings)Connections.txt		ClamAV		MALICIOUS	Win.Trojan.Autoit-7057849-)		
#10efc	Sonware\vanDyke\SecureFX		Comodo		MALICIOUS	TrojWare.Win32.Fareit.LB@	7pzcfo		
#108C8	CHISI ED/way, ftp.ipi		CrowdStrike		MALICIOUS	win/malicious_confidence_1	00% (W)		
#17112		~	Cybereason	OUS malicious.e5fa29					
			Cylance	Unsafe					
	Open rule		Cynet MALICIOUS Malicious (score						
			Curen		MALICIOUS	M/32/S_f2ff7deQIEldorado			~
Analysis finishe	d in 2566 ms		🖳 #0 (header:0)	PE	~	x86 ~	2566 ms		0

Figure 11: querying online intelligence

Can we go further? The last section of the PE file is weirdly named ".x". It contains a single method at address $0 \times 4a0000$ and a few bytes of referenced data at address $0 \times 4a0074$. Looking at the function, it seems to decode the data using a XOR opcode, with the key $0 \times DDDDFFFF$. But actually, only the first byte of the key is used ($0 \times FF$), so it is strictly equivalent to performing a simple NOT on the data. Great, let us decrypt these few bytes using Malcat's transform:



Figure 12: decrypting the data buffer in the .x section Great, we got the address of the command and control server for this sample. This was a pretty easy catch ^^

Conclusion

NSIS installers have been abused by malware authors for some years now. While the NSIS VM instruction set is relatively limited, DLL plugins allow malicious actors to extend installer capabilities and obfuscate malware. In this example, two layers of shellcodes were used by the NSIS installer in order to deliver its final payload: a LokiBot password stealer.

Instead of running everything in a VM, we made great use of Malcat's NSIS disassembler, Malcat's transforms and speakeasy emulator in order to quickly unpack these two layers statically.

We hope you enjoyed this new quick-and-dirty malware unpacking sessions. Future blog posts will be more focused toward beginners as we will introduce a few of Malcat's features as in-depth tutorials.