White Paper: PlugX - Payload Extraction

Kevin O'Reilly

plugx@contextis.com

March 2013

Contents

Introduction

The remote access Trojan malware strain known as PlugX has attracted a certain amount of attention in the security world during the last few months. PlugX is a relatively new backdoor implant, implicated in security problems experienced by a number of different organisations. It provides backdoor or remote access functionality, allowing an attacker to obtain information about infected systems and to egress data from the target. This white paper outlines analysis conducted by Context of PlugX in action within a client network. Below you will find details of the intelligence gathered during this process, including a description of how PlugX hides itself on disk using custom encryption. We also release source code for the command line tool that accompanies this paper, designed to recognise if a given file is in fact a PlugX payload file, and extract the executable and data contents ready for further analysis.

As a specialist provider of incident response services and one of the companies selected by GCHQ and CPNI to be part of the Cyber Incident Response scheme, Context has excellent visibility of the malware being used on a daily basis to compromise companies and government organisations. We are particularly knowledgeable about how malware is used to facilitate targeted, state sponsored cyber attacks. Recently, a client network which we have been monitoring through our managed service for several years and which is targeted regularly by state actors became infected with multiple instances of PlugX. Thanks to the client's appetite for risk and our ability to detect malware, we were able to conduct a live analysis of the attackers' activities and their use of PlugX.

Our client has been under sustained attack by several discrete Chinese threat actors of varying abilities for some time. Most of them appear[1] to have some connection to the Chinese state. The company operates in the extraction sector and has been targeted repeatedly by attackers seeking to access commercial and strategic information and to conduct mass email harvesting of personnel in key positions.

During our analysis we found an apparent propensity for the attacker to deploy the malware by leveraging access gained through previous compromise. PlugX was often deployed alongside other more common attacker tools such as Poison Ivy. In all we identified about eight samples on the same corporate network, distributed across North America, Africa, Asia and Australasia.

Initially PlugX was seen deployed on hosts by the attacker alongside pre-existing implants. Use of this strategy, in combination with other factors such as the creation of large debug logs, strings analysis indicating paths to symbol files, and a perceived instability in the malware, led us to believe that these early deployments were development versions of the malware, with the client network being used by the attacker as a kind of live test-bed. Over time there was some progression in the malware itself, and the attacker moved to deploying it in standalone form on hosts not previously compromised, with varying degrees of success.

PlugX is notable for a number of novel design decisions made by the author(s) of the malware. One interesting aspect, certainly in the more recent variants, is the hiding of the

[1] In terms of the company being attacked, the data being taken, the malware and infrastructure used in the attack (and observed being used in other attacks against other clients), and the actors with a requirement for such data, China is the only candidate. We have thoroughly explored and discounted all other possibilities.

malicious functionality inside a payload file. These files usually appear with innocuous file names, helping to conceal their true nature. In fact, they consist of an assembly code stub and an executable file in compressed and encrypted form. As we shall see below, the compression used is the LZ algorithm, provided by the Windows API function *RtlCompressBuffer.* The encryption process uses a custom algorithm written by the malware author(s).

Encryption Algorithm

The encryption algorithm may be described as the result of performing a bytewise XOR operation on the plaintext data (which here consists of the compressed executable image) with the output of a simple *Linear Congruential Generator* or *LCG* which has been written by the author(s). Often in the field of cryptography a *pseudo-random number* generator or *PRNG* is desirable, and in this respect *LCGs* represent one of the oldest and best-known families of algorithm for this purpose. Despite this, they are often regarded as a poor source of pseudo-random numbers, as the *nth* number r_n and that which follows it, r_{n+1}, are not truly independent of each other as random numbers would be, and anyone who knows r_n can predict r_{n+1} , therefore an LCG is not cryptographically secure.

However, in the application of malware there are a few advantages which are likely the motivating factors in using such an algorithm in the ways PlugX does. In this family, the simple LCG algorithm is used to both encrypt and compress the executable payload on disk, as well as the traffic sent to and from the remote command-and-control server. Since the remote server is a web server of some form, and the malware is written in C or IA32 assembly, it is desirable that the encryption algorithm be easily implemented in different programming languages, which is the case here. Such an algorithm also offers speed which is important, particularly on the server side where the language used may be interpreted rather than compiled. Finally, this method allows the algorithm to easily reproduce the same sequence of numbers, provided the initial seed is the same. In the case of the PlugX implementation, this seed may otherwise be considered as the encryption 'key', and since it is actually given along with the encrypted payload files and communications traffic, allows us to easily decrypt the encrypted data in each of these cases. However, as we will see later, the presence of this 'key' or seed is not actually necessary for successful decryption to take place.

Structure of Payload Files

The payload files seen by Context in examples of infected client machines all have a similar structure. These files are saved to disk with a variety of filenames which are intended to distract from their true nature. Examples seen include *xxx.xxx*, *Nv.mp3*, and *QQBrowser.pak*, but of course the filenames are not important, it is their content that is significant. To understand their contents, we should first begin by describing the way the malware is packaged, and the chain of execution that results upon launch. In the examples Context has analysed, as well as others which have been described in the media by other security researchers, the malware is packaged in three separate files. The execution begins with a legitimate third-party EXE, or Windows executable, which can be directly launched by the operating system. This EXE may not be something that was present on the target system before infection, but it is presumably hoped that it will pass unnoticed when run, particularly as it has a legitimate digital signature as well as a recognisable vendor and name.

This EXE in turn loads a DLL file which it depends upon, and whose name and imported functions are hard-coded within the file. The DLL file has, however, been modified by the malware authors to load the third and final file, which we call the payload file. The malware presumably depended upon the fact that the modifications made to the DLL file to load, decrypt and decompress the contents of the payload file had not been seen by Anti-Virus firms at the time of the malware being first used, and hence had not been blacklisted for detection by Anti-Virus products.

This set up does mean that once the modified DLL file has been seen by Anti-Virus vendors, and a signature to detect it made, the authors can then switch to a new and unseen loading mechanism whilst leaving this concern separate from the payload file which actually contains the malicious backdoor functionality.

Finally we come to the payload file itself. This begins with x86 assembly code, which is intended to be executed when the files are first loaded by the modified DLL. This code consists initially of just a few instructions which serve to identify the code's location in memory when it is executed.

```
call    $+5     ; call 5 bytes beyond current instruction (i.e. next instruction)
pop     eax     ; take return address for call from the stack
sub     eax, 5  ; subtract 5 to get address of call in eax
```

This function is commonly seen in code that is intended to be relocatable, as its instructions serve the purpose of ultimately setting the *EAX* register with the address in memory at which the *call* instruction is located. The reason for this requirement within the setup of this malware can be understood by analysing the code of the modified DLL file which is responsible for loading the payload file. Within this code we find the following call to the Windows API function *VirtualAlloc*[1], which is responsible for allocating the memory buffer which the payload file will be loaded into:

```
PayloadBuffer = VirtualAlloc(NULL, 0x10000h, MEM_COMMIT,  PAGE_EXECUTE_READWRITE);
```

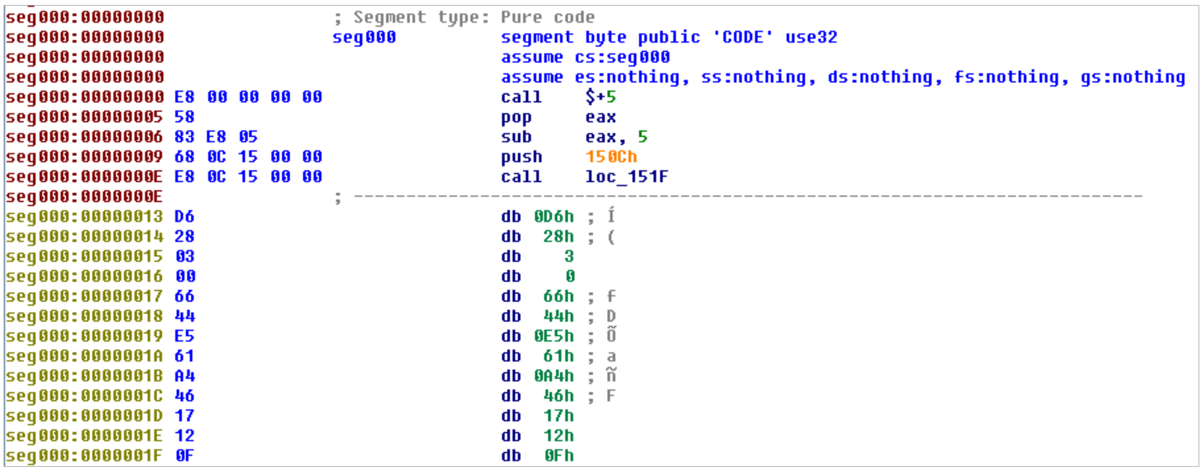
[1] http://msdn.microsoft.com/en-gb/library/windows/desktop/aa36688 7%28v=vs.85% 29.aspx

The documentation tells us that when the first parameter, *lpAddress, is set to NULL, it is left to the system to determine the location of the allocated buffer. Therefore this address may vary from one version of Windows to another, or from one execution to another.*

This self-locating code is immediately followed by two further instructions:

```
push    <size of encrypted block>        ; store block size to stack
call    <beyond encrypted block>         ; continue execution after block
```

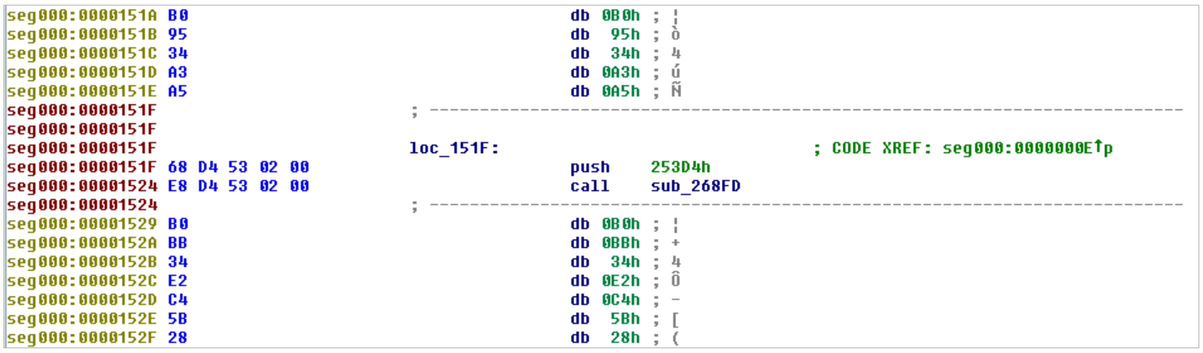
These instructions are immediately followed by a section or block of encrypted data, and can be interpreted as an instruction, the *push*, which copies the size of the ensuing encrypted block to the top of the stack, and then a *call* which serves to jump beyond the encrypted block to continue execution.



```
seg000:00000000                          ; Segment type: Pure code
seg000:00000000                          seg000          segment byte public 'CODE' use32
seg000:00000000                                          assume cs:seg000
seg000:00000000                                          assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000 E8 00 00 00 00                           call    $+5
seg000:00000005 58                                       pop     eax
seg000:00000006 83 E8 05                                 sub     eax, 5
seg000:00000009 68 0C 15 00 00                           push    150Ch
seg000:0000000E E8 0C 15 00 00                           call    loc_151F
seg000:0000000E                          ; ---------------------------------------------------------------------------
seg000:00000013 D6                                       db 0D6h ; Í
seg000:00000014 28                                       db  28h ; (
seg000:00000015 03                                       db    3
seg000:00000016 00                                       db    0
seg000:00000017 66                                       db  66h ; f
seg000:00000018 44                                       db  44h ; D
seg000:00000019 E5                                       db 0E5h ; Õ
seg000:0000001A 61                                       db  61h ; a
seg000:0000001B A4                                       db 0A4h ; ñ
seg000:0000001C 46                                       db  46h ; F
seg000:0000001D 17                                       db  17h
seg000:0000001E 12                                       db  12h
seg000:0000001F 0F                                       db  0Fh
```

This first block turns out to be a section of data which has been encrypted and compressed using the same algorithms as used throughout the operation of this malware, and contains information which is to be used by the malware payload, such as the domain name or IP address of the command and control server, as well as the disk location for the service the malware will use when it installs itself, its name and its description.

Once beyond this data block, the code continues its execution with just two more instructions, again a *push* and a *call*, this time defining the size of a second encrypted and compressed region which turns out to be the most significant; this is the executable payload in DLL form. Again the code continues by jumping beyond this block.



```
seg000:0000151A B0                                    db 0B0h ; ¦
seg000:0000151B 95                                    db  95h ; ò
seg000:0000151C 34                                    db  34h ; 4
seg000:0000151D A3                                    db 0A3h ; ú
seg000:0000151E A5                                    db 0A5h ; Ñ
seg000:0000151F                       ; ---------------------------------------------------------------------------
seg000:0000151F
seg000:0000151F                       loc_151F:                               ; CODE XREF: seg000:0000000E↑p
seg000:0000151F 68 D4 53 02 00                        push    253D4h
seg000:00001524 E8 D4 53 02 00                        call    sub_268FD
seg000:00001524                       ; ---------------------------------------------------------------------------
seg000:00001529 B0                                    db 0B0h ; ¦
seg000:0000152A BB                                    db 0BBh ; +
seg000:0000152B 34                                    db  34h ; 4
seg000:0000152C E2                                    db 0E2h ; Ô
seg000:0000152D C4                                    db 0C4h ; -
seg000:0000152E 5B                                    db  5Bh ; [
seg000:0000152F 28                                    db  28h ; (
```

Structure of Encrypted Blocks

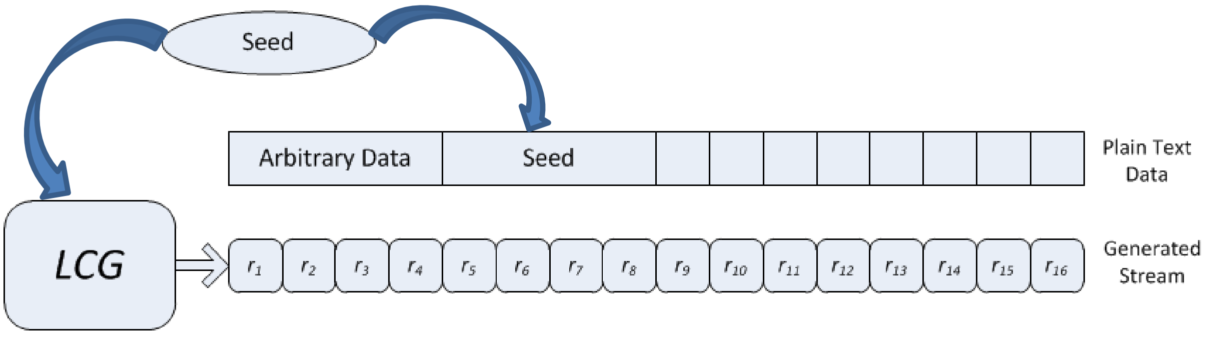
One of the features that the malware relies upon in all of its encryption in the samples seen, whether this be files on disk or internet communications with the command-and-control server, is that the seed (or 'key') for the encryption is written in the first four bytes of the (otherwise) encrypted block. This of course compromises the security of the encryption, and allows us an easy path to decrypting both encrypted files and internet communications traffic.

But the malware goes a step further, and includes the seed within the data to be encrypted. This allows a check to be performed once decryption of this data has occurred, to ensure the seed used for decryption matches the decrypted seed stored in the cipher text.

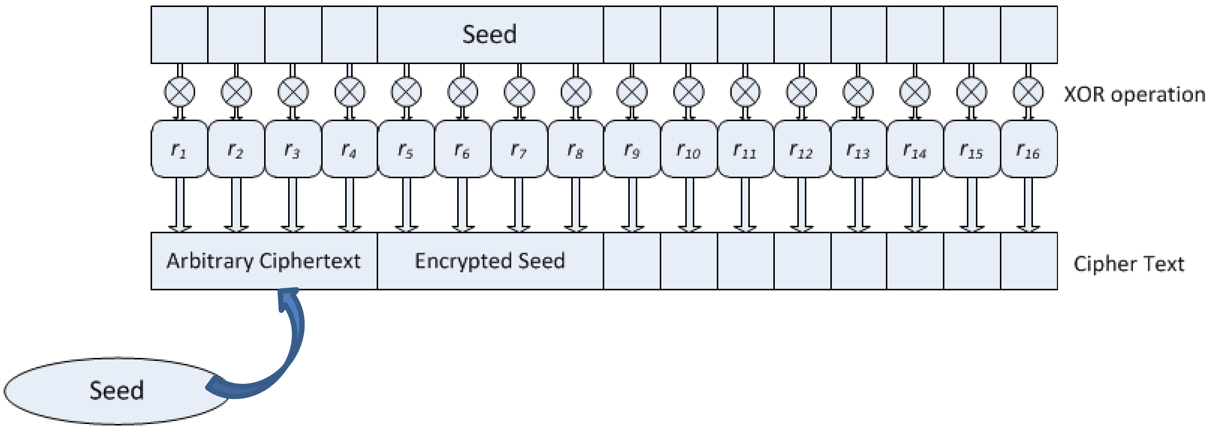
Encryption Process

When the encrypted block is first created, the first four bytes are in fact arbitrary and contain inconsequential data, as this four-byte space will later be used to store the seed in plain text, which will overwrite whatever ends up there.

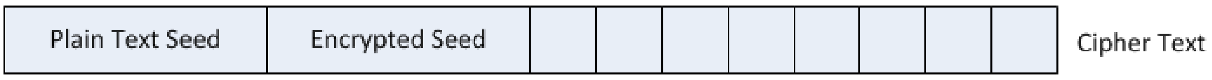
The seed is generated to feed to the LCG algorithm, but this four-byte value is also written into the second four bytes of the plain text data block to be encrypted.


Seed
Arbitrary Data
Seed
Plain Text Data
LCG
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15} r_{16}
Generated Stream

Then the plain text data is encrypted using the stream generated from the LCG algorithm. The arbitrary data, then the seed, then the remaining data are all XORed against the stream from the LCG producing the cipher text.

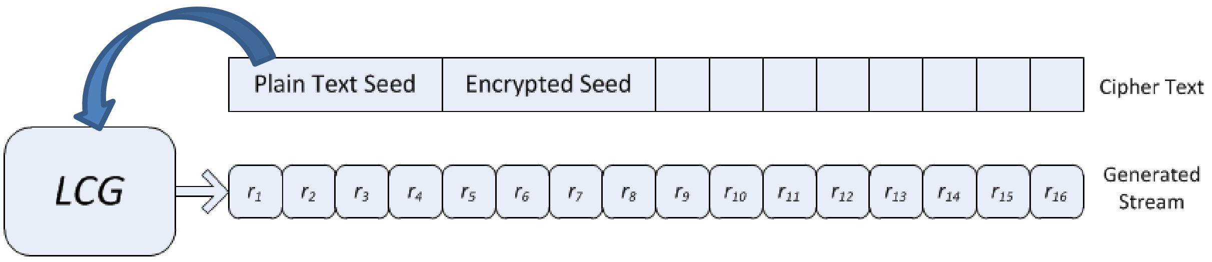

Seed
XOR operation
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15} r_{16}
Arbitrary Ciphertext
Encrypted Seed
Cipher Text
Seed

Finally the first four bytes of the cipher text are overwritten with the plain text seed so that the result is the plain text seed, followed by the encrypted seed, followed by the rest of the encrypted data.

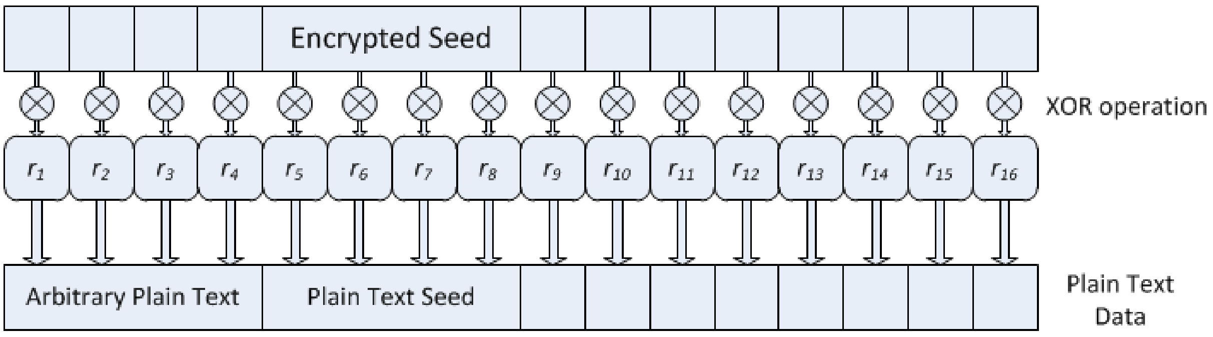


Decryption Process

When the decryption process commences, the seed for the LCG algorithm is taken from the first four bytes of the (otherwise) encrypted block:


Plain Text Seed
Encrypted Seed
Cipher Text
LCG
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15} r_{16}
Generated Stream

This allows the stream to be generated which is then used to decrypt the cipher text by performing a bytewise XOR operation:


Encrypted Seed
XOR operation
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15} r_{16}
Arbitrary Plain Text
Plain Text Seed
Plain Text Data

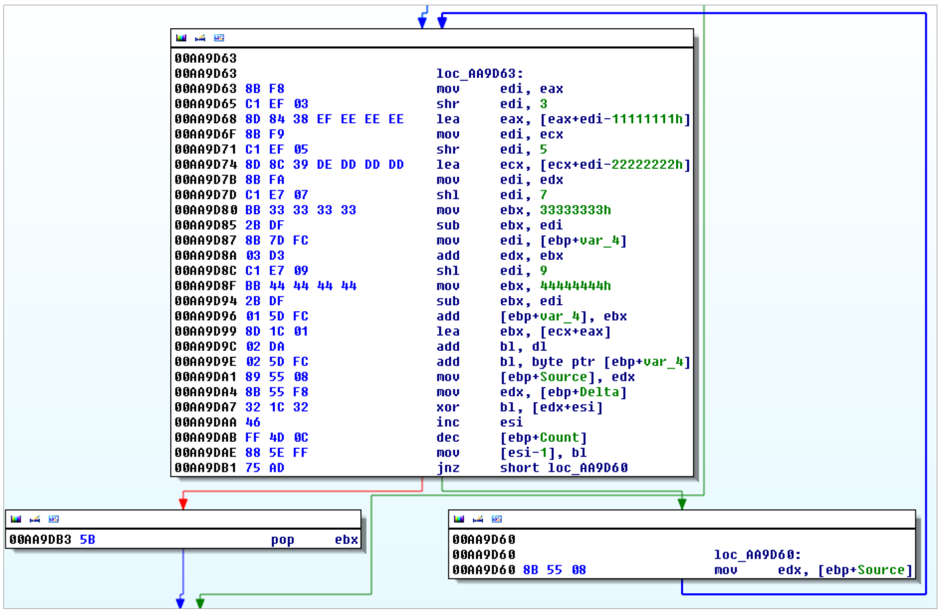
The result is that the first four bytes contain arbitrary data once more (this is actually the seed encrypted with the first four bytes of the LCG stream – but this is of no consequence). But now the second four bytes contain the decrypted seed. This permits the malware to check that the data it is performing the decryption operation upon is indeed PlugX encrypted data.

But this fact is even more valuable to us. It not only provides us with the same mechanism to check that we are decrypting a PlugX payload, but because the process of taking the seed, generating the LCG stream and decrypting data is quick to do computationally, we can put this to even greater use as we will see later on.

Decryption of Suspected Payload Files

Having studied the structure of the payload files which we have seen, we may write code to first re-implement the encryption algorithm, and then go to the relevant offsets where the encrypted cipher text is located in the payload files, retrieve the seed for the LCG, and decrypt the encrypted block. We can then make use of the malware's own checking mechanism by ensuring that the second set of four bytes which we have decrypted is actually the encryption seed. With this check in place we can be confident we have definitely correctly located an encrypted block, and so we write the result of decrypting the entire block out to a file.

Our first task in creating a decryption tool is to re-implement the malware's algorithm using its LCG. It is very straightforward to achieve this when creating our tool in C (or C++), as once we have located the LCG algorithm within the malware disassembly, we can lift it straight from the disassembler and integrate it into our tool source as an assembly source file with just a few additions. Here is the main part of the LCG algorithm in our disassembler, in this case IDA Pro:



```
00AA9D63
00AA9D63                              loc_AA9D63:
00AA9D63 8B F8                        mov     edi, eax
00AA9D65 C1 EF 03                     shr     edi, 3
00AA9D68 8D 84 38 EF EE EE EE         lea     eax, [eax+edi-11111111h]
00AA9D6F 8B F9                        mov     edi, ecx
00AA9D71 C1 EF 05                     shr     edi, 5
00AA9D74 8D 8C 39 DE DD DD DD         lea     ecx, [ecx+edi-22222222h]
00AA9D7B 8B FA                        mov     edi, edx
00AA9D7D C1 E7 07                     shl     edi, 7
00AA9D80 BB 33 33 33 33               mov     ebx, 33333333h
00AA9D85 2B DF                        sub     ebx, edi
00AA9D87 8B 7D FC                     mov     edi, [ebp+var_4]
00AA9D8A 03 D3                        add     edx, ebx
00AA9D8C C1 E7 09                     shl     edi, 9
00AA9D8F BB 44 44 44 44               mov     ebx, 44444444h
00AA9D94 2B DF                        sub     ebx, edi
00AA9D96 01 5D FC                     add     [ebp+var_4], ebx
00AA9D99 8D 1C 01                     lea     ebx, [ecx+eax]
00AA9D9C 02 DA                        add     bl, dl
00AA9D9E 02 5D FC                     add     bl, byte ptr [ebp+var_4]
00AA9DA1 89 55 08                     mov     [ebp+Source], edx
00AA9DA4 8B 55 F8                     mov     edx, [ebp+Delta]
00AA9DA7 32 1C 32                     xor     bl, [edx+esi]
00AA9DAA 46                           inc     esi
00AA9DAB FF 4D 0C                     dec     [ebp+Count]
00AA9DAE 88 5E FF                     mov     [esi-1], bl
00AA9DB1 75 AD                        jnz     short loc_AA9D60

00AA9DB3 5B                           pop     ebx

00AA9D60
00AA9D60                              loc_AA9D60:
00AA9D60 8B 55 08                     mov     edx, [ebp+Source]
```

Here is the same code reformatted as assembly which we can add to our sources to make our tool:

```
title  "PlugX Crypto"

.586p
.MODEL FLAT, stdcall

.DATA
.CODE

PlugXCrypt PROC C Seed:DWORD, Destination:DWORD, Source:DWORD, Count:DWORD

LOCAL    Delta    : DWORD
LOCAL    Var      : DWORD
```

```
        cmp     Count, 0
        mov     eax, Seed
        mov     ecx, Destination
        mov     edi, eax
        mov     esi, ecx
        mov     ecx, edi
        mov     edx, edi
        mov     Var, edi
        jle     finish
        mov     edi, Source
        sub     edi, esi
        mov     Delta, edi
        mov     edi, Count
        push    ebx
        mov     Count, edi
        jmp     do_loop
continue:
        mov     edx, Source
do_loop:
        mov     edi, eax
        shr     edi, 3
        lea     eax, [eax+edi-11111111h]
        mov     edi, ecx
        shr     edi, 5
        lea     ecx, [ecx+edi-22222222h]
        mov     edi, edx
        shl     edi, 7
        mov     ebx, 33333333h
        sub     ebx, edi
        mov     edi, Var
        add     edx, ebx
        shl     edi, 9
        mov     ebx, 44444444h
        sub     ebx, edi
        add     Var, ebx
        lea     ebx, [ecx+eax]
        add     bl, dl
        add     bl, byte ptr Var
        mov     Source, edx
        mov     edx, Delta
        xor     bl, [edx+esi]

        inc     esi
        dec     Count
        mov     [esi-1], bl
        jnz     continue
        pop     ebx
finish:
        xor     eax, eax
        ret

PlugXCrypt       ENDP
end
```

In the above assembly code I have made use of human-readable labels to symbolise the respective variables with their meanings, such as *Source, Destination, Seed* and *Count*.

With this integrated into our source, we proceed to code functionality to open the suspected payload files, then move to the location of the encrypted blocks and call our decryption algorithm upon them, decrypting the entire block and writing the result to file. However, in the examples Context has seen, the payload files contain different sizes of encrypted block, although they are all found at the same offsets within the files. But it is quite possible that different samples exist where the offsets of the encrypted blocks may differ, as well as their sizes. We would like to produce a tool which stands a fair chance of being able to decrypt the payload with potentially variable sizes and offsets from existing as well as new, previously unseen payload files.

A first step to a more general approach to decryption of the encrypted blocks within a payload file is to note that the assembly instructions we have seen may be found immediately prior to an encrypted block, and the instruction opcode bytes themselves contain the size of the blocks. So we can write code to recognise these instructions in the hexadecimal form within these files, with a simple pattern match, and take the sizes directly from the instruction operands.

In this way we are able to decrypt the contents of unseen payload files, assuming they continue to use this characteristic pattern of assembly instructions with the size of an encrypted block encoded in a *push* instruction immediately followed by a *call* to beyond the block to the next *push* and *call* for the next block.

Generalising Our Decryption Approach

Although we have found a method to reliably decrypt the contents of payload files we have seen, and have generalised in terms of the sizes and offsets of encrypted blocks that may be found in payload files we haven't seen, we are still reliant on several assumptions. These are that not only will the encryption algorithm not change in future examples, but that the assembly code will have the same structure marking the beginning of encrypted blocks and encoding their size. But we should attempt to minimise as much as possible the assumptions we are reliant upon, considering that there are ultimately many factors that the author(s) might change that could thwart our attempts for a yet-to-be-seen payload file. So to try and make our decryption tool resistant to changes in the layout of the payload files or the assembly code, we will take a different approach and remove our reliance on the assembly code layout and values. To do this, we will use the malware authors' own checking mechanism against them, harnessing it in a 'brute force' method.

To achieve this in our decryption tool we will simply open a candidate payload file, take the first four bytes as seed, and begin to generate the pseudo-random number stream from the LCG, taking the second four bytes of the stream to XOR with the second four bytes of the file, just as the malware's checking mechanism would. If the resulting four bytes are the same as the four bytes taken as seed, we know we have an encrypted block. If not, we simply move one byte further into the file, and repeat the process. If, by the time we have reached the end of the file and no match has resulted from our attempts at decrypting and finding the seed, we can rule out the file. But if an encrypted block (or more) is present, our check will succeed and we will know we have found it.

The resulting code does away with any pattern matching to recognise assembly code, or values for size contained within it, so we now have a tool that will find and decrypt any block that has been encrypted using the PlugX LCG algorithm within a file, as long as the encrypted block still uses the checking mechanism of containing an encrypted copy of the seed just after the plaintext seed.

We may at this point consider that we have done all we can in terms of reducing our assumptions and generalising our tool, but there is still one more step we can take. Of our two remaining assumptions, that the algorithm will remain the same, and that the encrypted seed check will persist, we can go further and strip the latter. But to do so relies on a quirk of the dictionary compression method used to compress the executable payload content just prior to decrypting it.

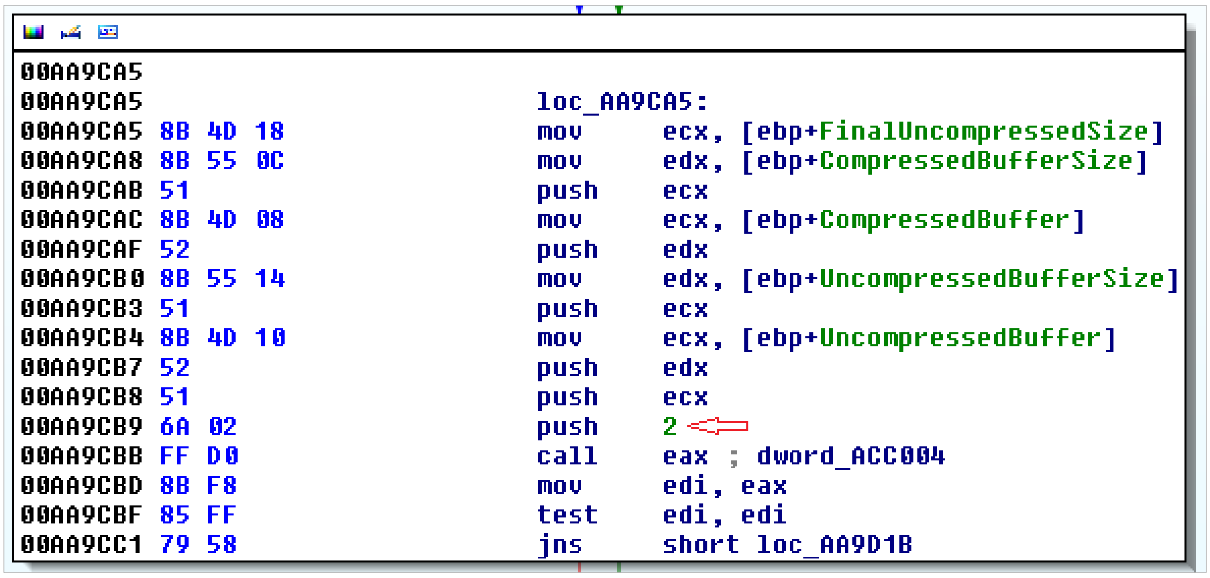
Compression Algorithm

The Windows API function that is used to compress the DLL payload (as well as the command-and-control traffic) prior to its encryption is *RtlCompressBuffer* (http://msdn.microsoft.com/en-gb/library/windows/hardware/ff552127%28v=vs.85%29.aspx).

The function prototype for this function is as follows:

```
NTSTATUS RtlCompressBuffer(
  _In_   USHORT CompressionFormatAndEngine,
  _In_   PUCHAR UncompressedBuffer,
  _In_   ULONG UncompressedBufferSize,
  _Out_  PUCHAR CompressedBuffer,
  _In_   ULONG CompressedBufferSize,
  _In_   ULONG UncompressedChunkSize,
  _Out_  PULONG FinalCompressedSize,
  _In_   PVOID WorkSpace
);
```

We see from the disassembled code from our PlugX samples that the *CompressionFormatAndEngine* parameter passed to the function has the value of 2.



```
00AA9CA5
00AA9CA5                         loc_AA9CA5:
00AA9CA5 8B 4D 18                mov     ecx, [ebp+FinalUncompressedSize]
00AA9CA8 8B 55 0C                mov     edx, [ebp+CompressedBufferSize]
00AA9CAB 51                      push    ecx
00AA9CAC 8B 4D 08                mov     ecx, [ebp+CompressedBuffer]
00AA9CAF 52                      push    edx
00AA9CB0 8B 55 14                mov     edx, [ebp+UncompressedBufferSize]
00AA9CB3 51                      push    ecx
00AA9CB4 8B 4D 10                mov     ecx, [ebp+UncompressedBuffer]
00AA9CB7 52                      push    edx
00AA9CB8 51                      push    ecx
00AA9CB9 6A 02                   push    2
00AA9CBB FF D0                   call    eax ; dword_ACC004
00AA9CBD 8B F8                   mov     edi, eax
00AA9CBF 85 FF                   test    edi, edi
00AA9CC1 79 58                   jns     short loc_AA9D1B
```

We see from the *Ntifs.h* header file that this parameter corresponds to *COMPRESSION_FORMAT_LZNT1*:

```
//  Compression package types and procedures.
//

#define COMPRESSION_FORMAT_NONE          (0x0000)   // winnt
#define COMPRESSION_FORMAT_DEFAULT       (0x0001)   // winnt
#define COMPRESSION_FORMAT_LZNT1         (0x0002)   // winnt

#define COMPRESSION_ENGINE_STANDARD      (0x0000)   // winnt
#define COMPRESSION_ENGINE_MAXIMUM       (0x0100)   // winnt
#define COMPRESSION_ENGINE_HIBER         (0x0200)   // winnt
```

This algorithm is is a *lossless compression algorithm* which was invened by Abraham Lempel and Jacob Ziv in 1977, and is a *dictionary coder*. This means that the mechanism by which it achieves lossless compression is by searching for matches between the data to be compressed and a *dictionary*. In the case of LZ compression, the dictionary is built during the compression of the data, such that patterns which are present in the data end up being stored in the dictionary during the encoding process.

This fact has a useful quirk in the case of LZ compression of a Windows executable, or *Portable Executable* (PE) file. One of the defining features of a PE file is the *DOS Signature* which consists of the first two bytes of every PE file. This is defined by Microsoft in the *WinNt.h* header file as follows:

```
#define IMAGE_DOS_SIGNATURE                 0x5A4D      // MZ
```

Now, if we write some code which implements the LZ algorithm using the *RtlCompressBuffer* function to compress a PE file, we immediately notice something convenient about its output. For example, look at the first few bytes resulting from compressing some randomly chosen Windows DLL files:

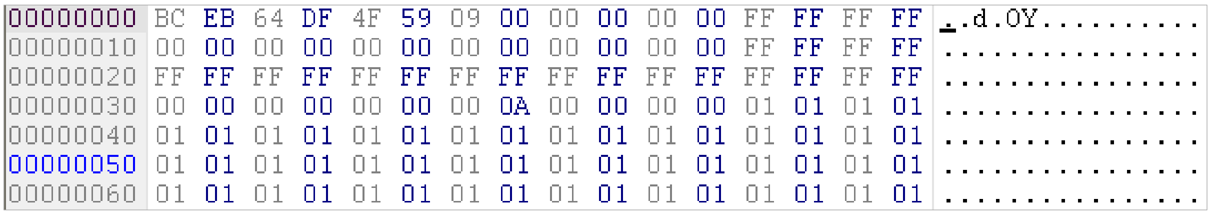
```
4F B9 00 4D  5A 90 00 03  O..MZ...    (ntdll.dll)
A9 BD 00 4D  5A 90 00 03  ...MZ...    (advapi32.dll)
72 BC 00 4D  5A 90 00 03  r..MZ...    (kernel32.dll)
55 BA 00 4D  5A 90 00 03  U..MZ...    (user32.dll)
55 BA 00 4D  5A 90 00 03  U..MZ...    (dbghelp.dll)
```

We see that the LZ algorithm has ended up storing the DOS signature (MZ) as the fourth and fifth bytes consistenly in all these files. We can use this fact to further our decryption tool by removing the reliance on the presence of the encrypted seed immediately following the plaintext seed, as now we know we can make a check for the MZ signature or 'magic number' at this location in our decrypted stream. If we find this, we will know that what we have found and can decrypt is in fact a PlugX payload in the form of a Portable Executable file.

The *RtlDecompressBuffer* API function[2] is quite forgiving in its *CompressedBufferSize* and *UncompressedBufferSize* input parameters, and so we can attempt decompression on the stream of bytes beginning three bytes before the MZ signature, with a *CompressedBufferSize* of the remainder of the file. With a large starting value, iterating downwards with the *UncompressedBufferSize* parameter will eventually result in success if the file is compressed, even if the input buffer size is actually larger than the compressed stream. Since the decompression algorithm works on a block-by-block basis, it will ignore any invalid compression blocks after the end of the compressed stream, and output the full decompressed file and no more.

Although in the samples we have seen this executable has been compressed, we can cater for the possibility that it might be uncompressed in other samples. In the case where the DOS and PE signatures are valid, as well as the *e_lfanew* pointer[3], we assume the file is not compressed, and just write the decrypted stream to file beginning at the MZ header.

The same principles can be applied to the encrypted data block. Since the decrypted data is structured with predictable byte patterns at the beginning, the same method of testing for these bytes upon decryption can be applied. The presence of a long stream of *01* bytes beginning part-way into the data is an obvious signature to look for.



```
00000000 BC EB 64 DF 4F 59 09 00 00 00 00 00 FF FF FF FF  ..d.OY..........
00000010 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF  ................
00000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00000030 00 00 00 00 00 00 00 0A 00 00 00 00 01 01 01 01  ................
00000040 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  ................
00000050 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  ................
00000060 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  ................
```

If they are not present upon decryption alone, the decompression phase is attempted, and the output again tested. In this way we can find, decrypt and optionally decompress the data segment too.

[2] *http://msdn.microsoft.com/en-gb/library/windows/hardware/ff552191%28v=vs.85%29.aspx*
[3] http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

Conclusion

With the PlugX LCG algorithm included as an assembly source file, we can create a simple tool which will iterate along an input file, testing each offset for the possibility of it being an encrypted block. The resulting tool can then search for the decrypted MZ header from a payload binary within, or the tell-tale signature of a data block. Both compressed and uncompressed scenarios can be catered for, and the tool can then go on to decrypt and optionally decompress the streams and write them to disk for further analysis.

The information and the attached source code will be useful to those of you who are dealing with a suspected PlugX infection, or require a command line tool to decrypt and decompress payload files automatically. Please download a copy of the source code for this tool from our website at the following address: *http://www.contextis.co.uk/research/white-papers/plugx-payload-extraction/*

The attackers will almost certainly change their tactics wherever discovered – the attackers in our case study have not used PlugX malware on the same client again since we chose (with the client's agreement) to close the infections down, having learned all there was to learn about their intentions and not wishing to further risk data. Where the attackers are still having success it is likely that they will continue to use the same malware. Where new versions of PlugX are used to reacquire targets, it may be the case that the encryption remains the same and our tool will continue to have use.

If you are dealing with an infection and have success in decrypting traffic, Context would obviously be very interested in any details you are able to share – in particular details of infrastructure used in the attack, types of data being stolen, the type of organisation being targeted and its geographic location, and of course, whether your analysis suggests that the attack is also the work of the Chinese state. Feedback can be sent to plugx@contextis.co.uk.

About Context Response

Context has worked for an extraordinary range of clients, including some of the most high profile financial companies in the world and government organisations, to identify and nullify security vulnerabilities and compromises.

Our monitoring and investigative services are tailored to meet the needs of each client and designed to help them understand the security risks facing the organisation and the potential implications of those risks. Services can be offered on a one-off or managed service basis, with forensic, analytical and reverse engineering techniques complemented by network monitoring and attack detection services.

Many clients ask Context to help them re-examine the security of their technology infrastructures: performing risk, impact and gap analysis exercises, studying network design and data handling and storage practices. We can help clients to establish which of their data are of most value to them and of most interest to an attacker; and the potential impacts of that data being lost.

We can also advise on changes that might be made to security policies and can help clients to embed security awareness and best practice more effectively within the culture of their organisations.

About Context

Context Information Security is an independent security consultancy specialising in both technical security and information assurance services.

The company was founded in 1998. Its client base has grown steadily over the years, thanks in large part to personal recommendations from existing clients who value us as business partners. We believe our success is based on the value our clients place on our product-agnostic, holistic approach; the way we work closely with them to develop a tailored service; and to the independence, integrity and technical skills of our consultants.

The company's client base now includes some of the most prestigious blue chip companies in the world, as well as government organisations.

The best security experts need to bring a broad portfolio of skills to the job, so Context has always sought to recruit staff with extensive business experience as well as technical expertise. Our aim is to provide effective and practical solutions, advice and support: when we report back to clients we always communicate our findings and recommendations in plain terms at a business level as well as in the form of an in-depth technical report.

For more news on Context follow us on Twitter @CTXIS or LinkedIn

context
INFORMATION SECURITY