Vawtrak Banking Trojan Technical Report

Raashid Bhat
BlueLiv Labs
Vawtrak Trojan has been an enduring banking trojan for a long time. It is one of the most prevalent banking Trojan in the wild today. It also known as neverquest or snifula.

Default packer embeds a resource section which consists an encoded and compressed data buffer. Which consists a static configuration buffer and main binaries (32 bit and 64 bit).

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3970</td>
<td>0xF82</td>
<td>Microsoft portable executable</td>
</tr>
<tr>
<td>71554</td>
<td>0x11702</td>
<td>Microsoft portable executable</td>
</tr>
</tbody>
</table>

This data is compressed using LZMAT compression (previously was using APLIB) and is LCG encoded.

A static configuration buffer consists of important parameters for the main bot to function and an executable code to load the main binary in memory.

This code implements a small loader which relocates one of the main binaries on an allocated executable region and jumps to entrypoint.
Base of this static configuration buffer is passed in as a second parameter to decoded main binary.

Following is a tentative list of items which are present in static configuration

1: Command and control servers
2: Botnet Configuration eg ProjectID, UpdateVersion, TorAddress, etc etc
3: URL resources
4: Keys to verify module integrity

A simple python script can be used to extract the second stage buffer from the resource object dump

```python
import struct
import os
from ctypes import *

_LZMATPATH = os.path.join(os.path.dirname(__file__), './lzmat.so')
print(_LZMATPATH)
lib = cdll.LoadLibrary(_LZMATPATH)

print(_LZMATPATH)
def encode(data):
    size = len(data)
```
```python
outlen = (size)+(size+7) >> 3)+0x21
out = create_string_buffer(outlen+sizeof(c_int()))
outlen = c_int(outlen)
ret = lib.lzmat_encode(byref(out), byref(outlen), data, len(data))
print str(bytearray(out)[:outlen.value]).encode('hex'), outlen
if ret == 0:
    return out[:outlen.value]
else:
    raise Exception('Return error: %d' % ret)

def decode(data, size):
    if size:
        outlen = size
    else:
        outlen = len(data)*10000
out = create_string_buffer(outlen)
outlen = c_int(outlen)
ret = lib.lzmat_decode(byref(out), byref(outlen), str(data), len(data))
if ret == 0:
    return out[:outlen.value]
else:
    raise Exception('Return error: %d' % ret)

def LCG(seed):
    seed = ((seed * 0x41C64E6D) + 0x3039 ) & 0xFFFFFFFF
    return seed

def LCGDecode(buff, seed, packLen):
    dst = ""
    for p in range(0, packLen):
        seed = LCG(seed)
        dst = dst + chr( ( ord(buff[p]) - (seed & 0xff) ) & 0xff )
    return dst

fp = open("data.mem", "rb")
data = fp.read()

seed = struct.unpack("<L", data[0:4])[0]
print "["seed = %x" % seed
data = LCGDecode(data[4:], seed, len(data[4:]))
DecompressLen = struct.unpack("<L", data[0:4])[0]

open("final.bin", "wb").write(decode(data[4:], DecompressLen))
```
Main binary

Vawtrak begins by generating an event name for each running process which is of the following format
{\%0.8X-%0.4X-%0.4X-%0.4X-%0.4X%0.8X} in which each of the format specifier are generated by successive pseudo random values generated out from an initial seed. Initial seed in this case is the process ID. This event is used to prevent code injection twice in a same process.

Strings are encoded using the LCG algorithm and can be effectively decoded using an IDA script

```c
unsigned int StringDecode(const char *a1, ...
{
    unsigned int v2; // [sp+0h] [bp-Ch]@1
    int v3; // [sp+4h] [bp-8h]@0
    unsigned int i; // [sp+0h] [bp-4h]@0
    const char *v5; // [sp+14h] [bp+8h]@0
    va_list va; // [sp+18h] [bp+Ch]@0

    va_start(va, a1);
    v3 = *((DWORD *)a1);
    v2 = (unsigned int)(*((DWORD *)a1) ^ *((DWORD *)a1 + 1)) >> 16;
    v5 = a1 + 8;
    for ( i = 0; i < v2; ++i )
    {
        v3 = 1103515245 * v3 + 12345;  
        *(_BYTE *)(i + *(int *)va) = v5[i] - v3;
    }
    return v2;
}
```

Depending upon the stage of binary (injected or standalone) a main function of the binary would be either executed as a thread or directly called.

Upon the execution of main thread it again create an event which has a format OLE%0.8X%0.2X%0.2X%0.8X%0.8X with initial seed as 13. This event will be used to trigger down a shutdown event call. Which basically unmaps the code from an executing image.
In the first run it will try to locate and inject into explorer.exe. A byte at an offset 0x0C in static configuration buffer is used as a marker to keep track of the injection process.

Meanwhile if executed inside explorer.exe it starts to inject inside all processes except a blacklist.

Vawtrak generates namespaces for multiple objects like filepaths, registry storage and other UID’s which are used for specific purposes thought out the run time of the malware (explained later).

All these name spaces are generated using an initial number as a seed to pseudo number generator. Following is a tentative list of name spaces and their corresponding numbers (init Seed)

**UID namespace**

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Registry Run persistencekey</td>
</tr>
<tr>
<td>#3</td>
<td>DebugLog FileMapping</td>
</tr>
<tr>
<td>#5</td>
<td>Update Internal Static Config</td>
</tr>
<tr>
<td>#6</td>
<td>CommStructEvent</td>
</tr>
<tr>
<td>#8</td>
<td>IPC InternalStruct Eventname</td>
</tr>
<tr>
<td>#9</td>
<td>IPC InternalStruct FileMapping Name</td>
</tr>
<tr>
<td>#10</td>
<td>IPC InternalStruct Mutex Name</td>
</tr>
<tr>
<td>#13</td>
<td>OLEID for Shutdown bot event listener</td>
</tr>
<tr>
<td>#16</td>
<td>OLEID for installer subroutine event</td>
</tr>
</tbody>
</table>

**Registry Path Namespace**

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Saved Webinject Configuration</td>
</tr>
<tr>
<td>#2</td>
<td>Set At restartPC event</td>
</tr>
<tr>
<td>#4</td>
<td>BotnetID</td>
</tr>
<tr>
<td>#5</td>
<td>Updated StaticConf Reg value</td>
</tr>
<tr>
<td>#6</td>
<td>Random Hashed Value</td>
</tr>
<tr>
<td>#7</td>
<td>1 (const) Process list delivered</td>
</tr>
</tbody>
</table>
After this step binary starts its initialization phase. It sets up necessary privileges necessary for the binary and retrieves the VolumeSerialDrive number (which is used to generate folderpath and registry path). Depending upon the configuration in static buffer it might also setup a UnhandledExceptionFilter to log crash dumps

certains parameter which are required globally for the identification of bot are retrieved/generated some of which include
Bot ID is generated from a combination of data from GetAdaptersInfo() and VolumeDriveSerial and this value is immediately stored in in register storage number #3.

If Registry configuration number #5 is if, which consists of an updated configuration buffer retrieved from a tor connected web address (explained later) this pointer is rather pointed to that buffer.

A type of process variable is set which basically corresponds to shell if explorer is the running process or 'browser' if it is running in one of the target browsers.

Vawtrak also logs its event in a shared filemapping. Most of the important events are logged thought a Shared File Mapping. The name of this file map is generated using namespace number #3. It has been intentionally put in by the author to assist him/her during the developmental stage of this trojan. It provides a function to print a formatted debug log.
These debug logs can be captured using the following C Program

```c
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

void dump(char buf[]) {
    int i = 0;

    for (i = 0; i < 0x10000; i++)
        printf("%c", buf[i]);
}

int main (int argc, char **argv)
{
    HANDLE hMapFile;
    LPCTSTR pBuf;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,FALSE,argv[1]);
    if (hMapFile == NULL)
    {
        printf("Error opening FileMap");
        return -1;
    }

    pBuf = (LPTSTR) MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0,0,0x10000);

    va_start(va, a2);
    if ( DebugFilemapingGUID )
    {
        v9 = (LPSTR)HeapAllocate(0x1000u);
        if ( v9 )
        {
            if ( printf? )
            {
                GetLocalTime(&SystemTime);
                v2 = SystemTime.wSecond;
                v3 = SystemTime.wMinute;
                v4 = SystemTime.wHour;
                v5 = GetCurrentProcessId();
                wprintfA(v9, "PID: %u [%0.2u:%0.2u:%0.2u ] ", v5, v4, v3, v2);
                v8 = sub_1000F6EF((int)DebugFilemapingGUID, v9);

```
dump(pBuf);
UnMapViewOfFile(pBuf);

CloseHandle(hMapFile);

return 0;
}

Output shows a debug log consisting of some internal calls and their return status

Based on the embedded static configuration, Vawtrak also sets up a ring-3 rootkit

It basically sets up a hook at CreateProcessInternalW to inject inside any new process launched in the current running executable
Most of the major tasks and communication with the command and control server is done through the code injected inside a browser. Some inbuilt commands or the ones received from c2 are supposed to be executed globally by all injected processes.

For this purpose vawtrak sets up a IPC communication mechanism again using a memory mapped File object. It sets up this IPC mechanism. It is internally represented using a structure

```c
struct CommStruct
{
    HANDLE CLSIDEvent; // CLSID ID 6
    void *Filemap; // 0x04 : SEE FileMap Struct
    void *PFilemap_0x16; // : Filemap + 16
    void *PFilemap_0x12_Sz1_; // 0x0c: FILEMAP + 0x_Sz1_400h + 0x10
    DWORD FileMapLowSize; // : Sz1_400h
    DWORD FileMapHighSize; // :Sz2_5000h (if map size is small, data buffer is written to a file)
    HANDLE FnStubThreadhandle; // :Handle for Stub Function
    void *StubFnPtr; // : Stub Pointer
    DWORD Const1; // : Function Argument
    DWORD waitTimeoutSeed; // : Seed to be fed to waitForSingleObject Timeout() 0
    INFINITE
    DWORD Const3;
};
```

A memory mapped file object is used to pass on the input data though an IPC call and an event which is used to trigger an IPC call

IPC call handler has the following definition

```c
int __stdcall IPCHandler(unsigned __int16 ProcID, int Const, const void *buff, DWORD bufflen)
```
ProcID: Process ID of caller
Const: Represents type of call to be performed
buff: Input Buffer
bufflen: input length

for example when a bot receives a PluginUnmap command from c2, it has to unmap from all injected processes. Though an IPC call it is instructed to perform so.

def VawtrakCrc32(buff):
    initMask = 0xffffffff
    for i in range(0, len(buff)):
        bt = ord(buff[i])
        for j in range(0, 8):
            if (bt ^ initMask) & 1:
                initMask = ((initMask >> 1) & 0xffffffff) ^ 0xEDB88320
            else:
\[
\text{initMask} = (\text{initMask} >> 1) & 0xffffffff
\]
\[
bt = (bt >> 1) & 0xff
\]
return initMask

Saved Web injects are encoded and compressed using the save LCG encoding and LZMAT compression.

First 4 bytes are used a seed and next 4 bytes represent actual length of the compressed web inject file.

Another namespace of value of #13 is generated to set up a shutdown event. If this event is signaled vawtrak unmaps the binary from the target image.
If the executing process is explorer.exe then an installer thread is created which copies the file to specified path and sets up the registry run keys. And if the executing process is a browser then it calls a subroutine which is responsible for communicating with the command and control center. This function looks for IBM rapport which is used for protection against MITM attacks in browsers and if found it disables the functionality of it by suspending the thread of rapport module.

\textbf{image > rapport}

\textbf{vawtrak} implements a structure for heap management and certain functions are provided for manipulating it.

Structure is represented as

\begin{verbatim}
struct MemStruct {
    void *Memory;
    CRITIALSECTION Crti;
    DWORD roundedtotalsize;
    DWORD datasize;
    DWORD Const; // (block section?)
    DWORD pagesize; // = 0x1000
}
\end{verbatim}
Following structure is prepared which is later fed to another subroutine and some parameters are generated from it

```c
struct InfoEntry
{
    BYTE TypeID;
    WORD Len;
    BYTE Data[Len];
};

struct
{
    struct botInfo
    {
        DWORD botID;
        DWORD projectID;
        WORD  updateVersion;
        WORD  buildVersion;
        WORD  Const0;
        BYTE  Const0;
        BYTE  isInstalled;
    }
    struct InfoEntry Injectcrc32_rand = {0, 8, [injecthash + randomDword]}
    struct InfoEntry ProxyServer = {1, strlen(proxyserver), proxyserver}
    struct InfoEntry CompName = {2, len(CompInfo), CompInfo}
    struct InfoEntry LangGroup = {3, len(LangGroup), LangGroup}
    struct InfoEntry VersoinInfo = {4, len(VersoinInfo), VersoinInfo }
    struct InfoEntry InstalledPlugins = {5, len(plugins), [WORD Plugins[i]]}
};
```

botInfo is used to generate a 32byte session ID, which is used as a cookie phpsessionid rest of the bytes in the struct are encrypted by rc4 using 4 byte botID.

```python
def GenerateSessionID(data):
    chars = "0123456789ABCDEF"
    output = ""
    InitSeed = struct.unpack("<B", data[0:1])[0]
    output = output + chr(InitSeed & 0xff)
    for i in range(1, len(data)):
        output = output + chr(ord(data[i]) ^ (InitSeed & 0xff))
        InitSeed = InitSeed + i
    PhpSess = ""
    #print output.encode("hex")
    for i in range(0, 16):
        PhpSess = PhpSess + chars[ord(output[i]) >> 4]
```
\textit{PhpSess} = \textit{PhpSess} + \textit{chars[ord(output[i]) & 0xf]} \\
\text{print "Session ID generated \[] = \%s" % PhpSess} \\
\text{return PhpSess}

This data is sent to command and control server and based on the configuration setting communication is either http or https.

List of c2's are present in encoded form inside static configuration buffer and and retrieved using the following subroutine

```c
*a3 = UrlIndex; 
*a2 = v5; 
v6 = (*DWORD *)4+v3[0x44 * v5 + 0x206]; 
v7 = 0; 
v10 = v6; 
do 
{ 
    (*BYTE *)(v7 + Dest) = *((_BYTE *)C2EncodedBuffer + 0x44 * v5 + v7 + 0x20A) ^ generateRandom(&v10); 
    v7++; 
} 
while (v7 < 0x40); 
}
```

before sending this data to c2 registry configuration key #7 is read and is not found then list of running processes are retrieved in a buffer in format of "%u(%u)t%s\n" representing processID, parent ProcessID and ImageName. This call is made though the opcode 14 provided by the main module. Vawtrak is a modular trojan it implements a plugin based architecture in which each module is a segregated component of the system including the main module. Each of these modules expose a method to call any API (opcodes) provided by that particular module. That is why we saw in the debug call log some of the function called initially

| PID: 1812 | [01:45:15] CALL TO=0 FROM=0 CMD=14 PCount=0 |
| PID: 1812 | [01:45:15] CALL Status=1 Size=579 |
| PID: 1812 | [01:45:15] CALL TO=0 FROM=0 CMD=7 PCount=2 |
| PID: 1812 | [01:45:15] CALL Status=1 Size=0 |

are two function to retrieve processlist and to send raw data to c2

those two functions CMD 14 and CMD 7
Main base module implements around **41 Command or API's**. These function can be called from any plugin or even by the request of c2.

Following table list some of the main functions provided:

- NullSubroutine
- GetsavedConfSectionData
- DecodeC2DatapacketWebinject
- DownloadAndLoadModules
- DeletePluginOp
- RetriveModNVerList
- InjectBotInProcess
- sendData2c2
- SendandRecvC2
- SetMainC2Eventhandle
- RetriveRegconfvalue
- SetConfigurationkeyValue
- DelConfigurationkeyValue
- RemoveAllConfigurationKeys
RetriveProcessList
ExecCommand
DownloadAndExecuteExeFile
RetriveDebugLog
OutpuDebugLog
ResetDebugLog
RestartPC
RestartPCSetConf2key
ShellExecuteExFile
ClearHostsFile
WriteHostsFileUnicode
AppendHostsFile
uploadDnsHostsFilesBuffer
DownloadUpdateAndShutdown
OverRideUserAgent
RetriveFolderPath
RetriveFilePath
RetriveRegPath
RetriveOLECLSID
RetriveRandom16
RetriveCLSID
RetriveShellCodeBase
SetC2TimeWait
send0x4000009Const

Any of these APIs can be called though a Dispatch Handler subroutine which acts as a focal point when there is an call to one of these API's . API's can be called by their position in the API array and module index.

During the initialization phase dispatch function and plugin structures are initialized for the main modules and as well as the saved Plugin. A list of modindex's and modversion's array is sent to c2 during initial packet for the c2 to determine which plug-ins are installed on the system.
struct basePacket
{
    DWORD Size;
    BYTE numsections;
    struct SectionData
    {
        BYTE moduleId;  // moduleIndex?
        BYTE callType;  // callType?
        BYTE NumSec;   // NumSec?
        struct _Data { // next section
            DWORD lenSection;
            BYTE Buffer[lenSection]
        } Data[NumSec];
    } SectionData[numsections];
}

It contains a header and multiple sections which have own headers specifying to which API and plug-in that particular section is designated for.

Data present in each SectionData entry has its own format, For example when a call opcode 3 == DownloadAndLoadPlugins() is received it has the following hex data

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF00Ch</td>
<td>52 01 00 0C 01 00 03 05 3E 00 0C 00 60 7A 7A 7A</td>
</tr>
<tr>
<td>FF01Ch</td>
<td>3A 2F 39 35 32 31 33 32 3E 2E 39 35 32 31 33</td>
</tr>
<tr>
<td>FF02Ch</td>
<td>3E 2F 6D 6E 64 75 6C 65 2F 32 37 32 61 35 61 64</td>
</tr>
<tr>
<td>FF03Ch</td>
<td>34 61 31 62 39 37 61 32 61 63 38 37 34 9A 36 64</td>
</tr>
<tr>
<td>FF04Ch</td>
<td>33 65 35 66 56 66 30 31 64 00 3E 00 00 00 68 7A</td>
</tr>
<tr>
<td>FF05Ch</td>
<td>74 70 3A 2F 2F 39 35 32 3E 32 31 33 2E 31 33 39 3E</td>
</tr>
<tr>
<td>FF06Ch</td>
<td>31 31 36 2F 6D 6F 64 75 6C 65 2F 64 31 31 36 37</td>
</tr>
<tr>
<td>FF07Ch</td>
<td>63 39 39 63 30 63 37 65 39 62 34 36 38 66 62 65</td>
</tr>
<tr>
<td>FF08Ch</td>
<td>30 38 65 35 39 65 34 31 66 66 65 00 3E 00 00 00</td>
</tr>
<tr>
<td>FF09Ch</td>
<td>68 74 74 70 3A 2F 2F 39 35 32 3E 32 31 33 3E 31</td>
</tr>
<tr>
<td>FF0ACh</td>
<td>39 2E 31 31 36 2F 6D 6F 64 75 6C 65 2F 39 36 36</td>
</tr>
<tr>
<td>FF0BCh</td>
<td>66 31 63 36 34 63 37 65 62 31 33 65 38 38 30 65</td>
</tr>
<tr>
<td>FF0CCh</td>
<td>33 39 39 66 39 36 32 37 65 30 64 62 30 00 3E 00</td>
</tr>
<tr>
<td>FF0DCh</td>
<td>00 00 60 7A 7A 7A 7A 7A 7A 7A 7A 7A 7A 7A 7A</td>
</tr>
<tr>
<td>FF0ECh</td>
<td>52 01 00 0C 01 00 03 05 3E 00 0C 00 60 7A 7A 7A</td>
</tr>
<tr>
<td>FF0FCh</td>
<td>3A 2F 39 35 32 31 33 32 3E 2E 39 35 32 31 33</td>
</tr>
<tr>
<td>FF10Ch</td>
<td>3E 2F 6D 6E 64 75 6C 65 2F 32 37 32 61 35 61 64</td>
</tr>
<tr>
<td>FF11Ch</td>
<td>34 61 31 62 39 37 61 32 61 63 38 37 34 9A 36 64</td>
</tr>
<tr>
<td>FF12Ch</td>
<td>33 65 35 66 56 66 30 31 64 00 3E 00 00 00 68 7A</td>
</tr>
<tr>
<td>FF13Ch</td>
<td>74 70 3A 2F 2F 2F 39 35 32 3E 32 31 33 2E 31 33</td>
</tr>
<tr>
<td>FF14Ch</td>
<td>31 31 36 2F 6D 6F 64 75 6C 65 2F 64 31 31 36 37</td>
</tr>
<tr>
<td>FF15Ch</td>
<td>63 39 39 63 30 63 37 65 39 62 34 36 38 66 62 65</td>
</tr>
</tbody>
</table>

first four bytes represent the total size  next bytes represents subsection and call type and mod index

In this case it is

MODINDEX = 0 (main module)
API CALL = 3 (DownloadAndLoadPlugins)

These plugins are encoded and compressed. Plugins are again encoded using LCG with the first 4 bytes of the file as seed. Other information like MODID and BUILDVERSION of plugin is provided for the loader. Plugins are saved locally and path is generated using #MODID namespace.

Following is a tentative list of plugins which are usually downloaded:

- **Web inject Plugin**
- **A stealer plugin (based on modified version of pony)**
- **Back Connect Module to Tunnel traffic though victims module**
- **Keylogger**
- **Certificate, history Stealer and A fileManger**

Input to any plugin takes the following parameters:

```cpp
struct Arg0
{
    BYTE modID;
    const BYTE CallingMod;
    WORD CallOpcode;
    BYTE numsection;
    struct SecinfoArr[numsection]
    {
        DWORD buffertype;  // {1 == integer, 2 == pointer, 0 MOD Number call}
        DWORD SectionLength;  // = 0x24
        DWORD InfostructPtr;
        BYTE compressed?;  //
    }
};
```

```cpp
struct InfostructPtr{
    DWORD SubCallID
    DWORD BotID;
    DWORD ProjectID;
    DWORD VolumeSerialNumber;
    DWORD CallDispatcherfnt;
    DWORD TLSIndex;
    DWORD ProcessTypeOPT;
    DWORD Const1;
    DWORD ProcessType;
};
```

Usually a webinject section of call type 2 is also supplied. Webinject file of Vawtrak is comprehensive and is similarity to UrlZone inject file. It further consists of encoded subsection which are numerically named which further consist of other sections.
For example web inject section #3 which corresponds to list of URL which are subjected to POST, HTTP AUTH and cookie theft has the following structure

```c
struct WebConfigSection
{
    DWORD TotalLen; // including header
    WORD NumOfSubSections;
    DWORD SboxSeed;

    struct SubSection [NumOfSubSection]
    {
        DWORD SectionSize; // including header
        DWORD SboxSeed; // including header
        WORD SectionIdentifier;
        WORD NumOfEntries;
        struct EntryCharacteristics
        {
            DWORD TotalEntrySize;
            BYTE TypeOfEntry; // Regex, Plain
            BYTE SupportedURLTypes; // Mask {1,2,3}
            BYTE Size; // Excluding Header
        }
        BYTE unknown;
        BYTE BrowserType;
        BYTE UNKNOWN;
        BYTE UNKNOWN;

        struct Sectionvalue{BYTE Size, BYTE Data[Size]};
    };
};
```