



ADVANCED  
ANALYTICS

G DATA **ADVANCED ANALYTICS**

## **Analysis Results of Zeus.Variant.Panda**

Luca Ebach

Analysis Report. June 22, 2017

G DATA Advanced Analytics GmbH  
G DATA Campus · Königsallee 178  
D-44799 Bochum, Germany

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	General Information . . . . .	3
2.2	Execution Flow . . . . .	4
<b>3</b>	<b>Anti-Detection and Anti-Reverse-Engineering Techniques</b>	<b>6</b>
3.1	Malware Startup Checks . . . . .	6
3.1.1	Debug support . . . . .	6
3.1.2	Language checks . . . . .	6
3.1.3	Anti analysis check . . . . .	6
3.2	Windows API Imports . . . . .	10
3.3	Crypted Strings . . . . .	10
3.4	Cryptography . . . . .	11
3.4.1	Random Numbers . . . . .	11
3.4.2	Cryptography . . . . .	11
3.4.3	Hashing . . . . .	12
<b>4</b>	<b>Configuration</b>	<b>13</b>
4.1	Bot ID . . . . .	13
4.2	Configuration . . . . .	13
4.2.1	Base Config . . . . .	13
4.2.2	Local Config (PeSettings) . . . . .	14
4.2.3	Dynamic Config . . . . .	15
4.2.4	Local Settings . . . . .	17
4.3	Bot Update . . . . .	18
4.4	Configuration Update . . . . .	18
<b>5</b>	<b>Payload and Persistence</b>	<b>20</b>
5.1	Persistence . . . . .	20
5.2	HTTP Grabber and Injector . . . . .	20
5.3	Process Injection . . . . .	22
5.4	API Hooking Technique . . . . .	22
5.5	Hooks and Browser Manipulation . . . . .	22
5.5.1	Internet Explorer . . . . .	23
5.5.2	Mozilla Firefox . . . . .	25
5.5.3	Google Chrome . . . . .	25
5.5.4	User Functions . . . . .	26

<i>Contents</i>	1
5.6 Plug-in ability . . . . .	26
5.7 Webfilters . . . . .	27
5.8 Remote Script . . . . .	27
5.9 System Report . . . . .	29
<b>6 Conclusion</b>	<b>30</b>

# 1 Introduction

Aside from ransomware attacks, banking trojans are also a very dangerous type of malware. They do not have destructive behaviour in the first place, so their presence on a victim's system might not be detected for quite an amount of time if the victim has no proper antivirus product installed. Since Panda is possibly among the most dangerous families of banking trojans, we decided to do a comprehensive analysis of a recent sample of Panda.

In this paper we focus on the analysis of the binary part of a Zeus.Panda malware sample. For a detailed analysis of the actual webinject behaviour and the communication flow between infected machines and the automatic transfer system's server, please refer to our blogposts<sup>1 2</sup> by Manuel Körber-Bilgard and Karsten Tellmann.

---

<sup>1</sup><https://cyber.wtf/2017/02/03/zeus-panda-webinjects-a-case-study/>

<sup>2</sup><https://cyber.wtf/2017/03/13/zeus-panda-webinjects-dont-trust-your-eyes/>

## 2 Overview

### 2.1 General Information

The original Zeus banking trojan's source code was leaked in 2011 and since then several independent threat actors have used the source code as a basis for new variants of the malware. One of the most prolific and advanced of these variants is the Zeus.Panda banking trojan which we will analyse in this white paper. Zeus.Panda targets Windows operating systems from WinXP through Windows 10 and is typically spread through phishing mail campaigns, but proliferation through drive-by exploits has been seen.

The sample analyzed in this whitepaper is:

#### MD5

Packed: e005c4009c22e0f73fcdaeba99bd0075  
Unpacked: 655f65b1b08621dfcb2603b59fca05bc

#### SHA1

Packed: 6f5c186baa0d69799c250769052236b8bcfb13a1  
Unpacked: 88782d3b74067d405e56f0a5e9b92e3fdb77dcd8

#### SHA256

Packed: d037723b90acb9d5a283d54b833e171e913f6fa7f44dd6d996d0cecae9595d0b  
Unpacked: bd956b2e81731874995b9b92e20f75dbf67ac5f12f9daa194525e1b673c7f83c

#### Size

Packed: 252 KB  
Unpacked: 140 KB

#### Number of Functions

538

#### IOCs (Filesystem)

Panda tries to find a directory underneath `%APPDATA%\Roaming` that

- is empty,
- has a path that is at least 140 characters long,
- does not contain either of *microsoft* or *firefox*, and
- is as deep in the directory tree as possible

In our analysis environment, Panda ended up in `%APPDATA%\Roaming\Sun\Java`. In the directory, Panda creates four files with random file extensions. We discovered

`Desktop (create shortcut).exe` (malware executable), `Control Panel.cyd` (dynamic config file, section 4.2.3), `Desktop.y sq` (report file, section 5.9), and `Notepad.kix` (localconfig file, section 4.2.2).

### IOCs (Registry)

Aside from writing some files to disk, Panda also uses some registry keys to store data. All the registry keys used by Panda are located in the `HKCU\Software\Microsoft` key. The names of the keys are random and in our system we observed `Ivoc` (reg-DynamicConfig), `Kounhu` (regLocalConfig), and `Useglugy` (regLocalSettings). See section 4.2.2 for a more detailed description of the configuration. Additionally, Panda creates a new entry within the `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` registry key which is used to start the malware as soon as the infected user logs into its account.

### IOCs (other)

Internally, Panda uses several mutexes and events to synchronize between the controlling process and the client instances in the browsers. The names of these objects are fixed on the local system but are different for any other system. Although, the names are 32-character hexadecimal strings in either case. Example: `4A0000002571569EA477E09F768C1A07`

## 2.2 Execution Flow

Figure 2.1 gives an overview of the control flow of Zeus.Panda. Each step will be described in detail in the coming chapters.

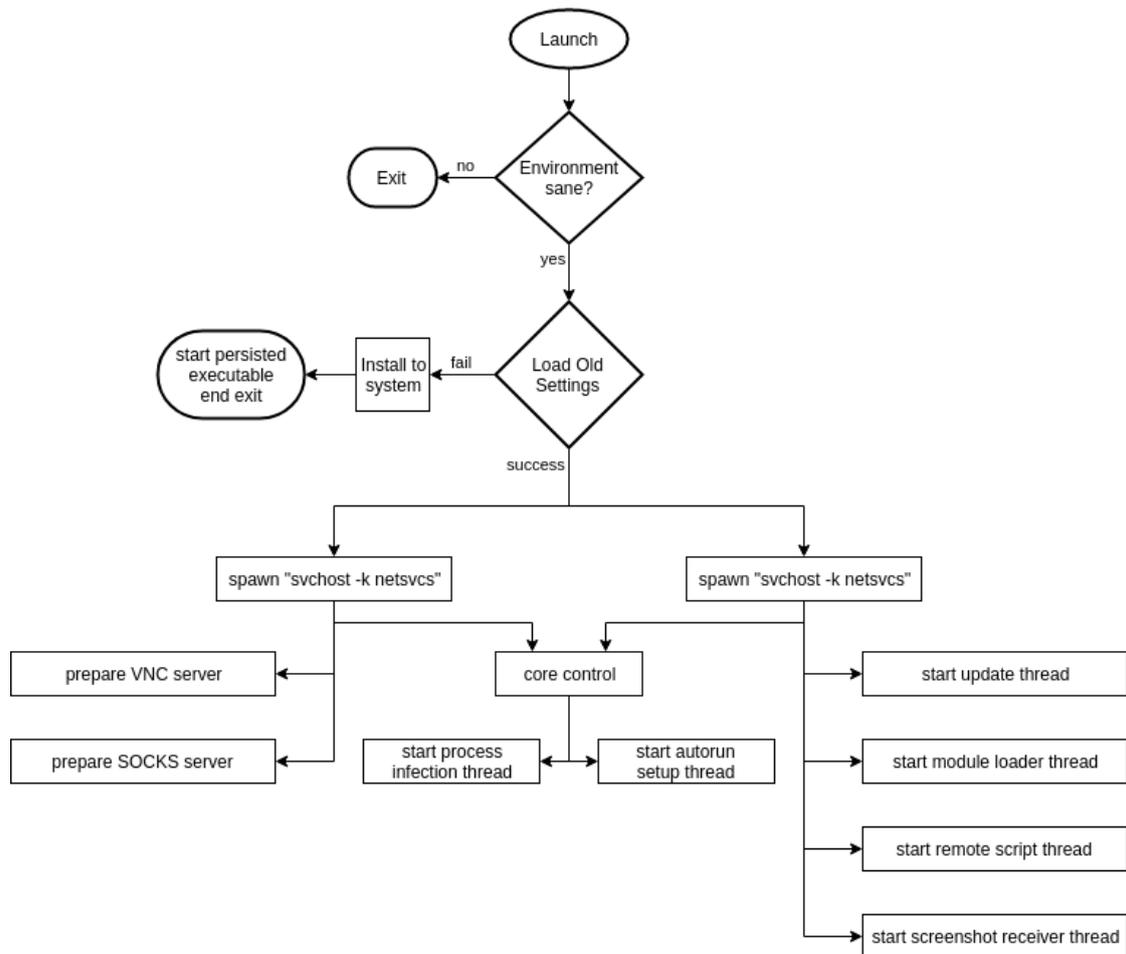


Figure 2.1: Control flow of the malware executable.

## 3 Anti-Detection and Anti-Reverse-Engineering Techniques

### 3.1 Malware Startup Checks

Before installing the malware executable in the victim's system, Panda performs several checks to verify that it runs in a sane environment.

#### 3.1.1 Debug support

The first check verifies the integrity of a `.dbg` file. If the file is present on the file system, it has the same name as the executable. The `.dbg` file contains encrypted JSON data 3.4 of the form

```
{
  "data": "[data]",
  "sign": "[signature]"
}
```

After reading the content of the file, Panda hashes the data part of the JSON object using SHA1 through the Windows Crypt API. Afterwards, it uses `CryptVerifySignature` to check the calculated hash against the content of the `sign` field using a static public key from the executable. If the signature is not valid, Panda removes itself from the system. If the signature check is passed, Panda will bypass the subsequent anti-analysis code.

#### 3.1.2 Language checks

Once the debug support check is passed, Panda checks the current keyboard layout against a predefined list of layouts. In the sample I analyzed, the list contained 0x419, 0x422, 0x423, 0x43f which stand for russian, ukrainian, belarusian, and kazakh, respectively. If either of those matches the current keyboard layout, Panda removes itself from the victim's PC.

#### 3.1.3 Anti analysis check

The last step of the pre-run checks is a rather long list of checks for debug and analysis tools. Some of these tools are antiquated such as SoftIce where support stopped long before Windows XP which is the least recent operating system supported by Panda. Other of the tools such as IDA Pro and Immunity Debugger remain popular tools with

malware analysts. If any of these tools are present Panda aborts execution and removes itself. To identify analysis tools Panda uses four different types of tests:

**file**

use `CreateFile` with `OPEN_EXISTING` flag to check if a file/device exists

**mutex**

use `OpenMutex` to try to open an existing mutex

**running process**

use `CreateToolhelp32Snapshot` to get the list of currently running processes and check if any of them contains a given string

**registry key**

use `RegOpenKey` to check if a registry key exists *or* check a registry key if it contains a given value

The full list contains checks for 23 tools and is shown in the table at the end of the section. If either of those tests fails, Panda stops to installing and removes itself from the system. Although, these checks can be skipped using `-f` as a command line parameter at the start of the malware.

**aut2exe**

process `aut2exe` running

**Bochs**

registry key `HKLM\HARDWARE\Description\System\SystemBiosVersion` contains *BOCHS*

**Execute**

file `C:\execute.exe` exists

**Frz**

mutex with name `Frz_State` exists

**IDA Pro**

process `idaq` running

**ImmunityDBG**

process `immunity` running

**Perl**

process `perl` running

**PopupKiller**

file `C:\popupkiller.exe` exists

**prl**

One of:

- file \\.\prl\_pv exists
- file \\.\prl\_tg exists
- file \\.\prl\_time exists

**ProcessExplorer**

process `procexp` running

**ProcessMonitor**

process `procmon` running

**ProcessHacker**

process `processhacker` running

**Python**

process `python` running

**Regshot**

process `regshot` running

**Sandboxie**

One of:

- `SbieDll.dll` can be loaded by `LoadLibraryA`
- mutex `Sandboxie_SingleInstanceMutex_Control` exists

**SoftICE**

One of:

- file \\.\SICE exists
- file \\.\SIWVID exists
- file \\.\SIWDEBUG exists
- file \\.\NTICE exists
- file \\.\REGVXG exists
- file \\.\FILEVXG exists
- file \\.\REGSYS exists
- file \\.\FILEM exists
- file \\.\TRW exists
- file \\.\ICEXT exists

**Stimulator**

file `C:\stimulator.exe` exists

**VirtualBox**

One of:

- file \\.\VBoxGuest exists
- file \\.\VBoxMouse exists
- file \\.\VBoxVideo exists
- file \\.\VBoxMiniRdrDN exists
- file \\.\VBoxMiniRdDN exists
- file \\.\VBoxTrayIPC exists
- registry key HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions exists
- registry key HKLM\HARDWARE\ACPI\DSDT\VBOX\_\_ exists

### VirtualIPC

One of:

- mutex MicrosoftVirtualPC7UserServiceMakeSureWe'reTheOnlyOneMutex exists
- file \\.\VirtualMachineServices exists

### VMware

One of:

- file \\.\HGFS exists
- file \\.\vmci exists
- registry key HKLM\SOFTWARE\VMware Inc.\VMware Tools exists

### Wine

One of:

- kernel32.dll contains "wine\_get\_unix\_file\_name" function
- registry key HKLM\Software\WINE exists
- registry key HKCU\Software\WINE exists

### Wireshark

One of:

- file \\.\NPF\_NdisWanIp exists
- process wireshark running

### Hypervisor

One of:

- check if hypervisor bit of CPU is set
- file \\.\VmGenerationCounter exists

```
Function Resolve(Module, FunctionID)
{
  For exportName in Module.Exports
  {
    If (CRC32(exportName) == FunctionID)
    {
      Return AddressOfFunction(exportName)
    }
  }
}

Function Import(ModuleID, FunctionID)
{
  If (FunctionID not in cache)
  {
    Module := DecryptName(ModuleID)
    If (Module is not loaded)
    {
      LoadLibrary(Module)
    }
    cache[functionID] := Resolve(Module, FunctionID)
  }

  Return cache[functionID]
}
```

Listing 3.1: Pseudocode describing the implementation of the Windows API import function.

## 3.2 Windows API Imports

To harden itself against static analysis, Panda avoids importing Windows API functions directly. Instead, it uses `LoadLibrary` and parses the export directory of libraries. It creates a CRC32 hash of each export name and compares it to a hardcoded CRC32 of the name of the desired import. If the two match, the function address from the export directory of the library is used. In case of forwarded exports Panda reverts to import the function by using the `GetProcAddress` API. A simplified pseudo code of the import function is shown in listing 3.1. The actual implementation is a bit more complicated, but this should give an overview of how it works.

There are exceptions however. It seems that some imports are, by accident, left in the binary. Fortunately, this includes functions like `LoadLibrary` and `GetProcAddress` which lowered the difficulty of the static analysis since we were able to determine the import function shortly after the start of the analysis. Also, calls to the `Heap*` functions (`Alloc`, `Free`, `ReAlloc`, `Create`, `Destroy`) and also a single call to `Sleep` are not imported using the custom import functions.

## 3.3 Crypted Strings

Most strings an analyst might come across during the analysis process are encrypted. This hinders an analyst from using strings to determine the purpose of some functions.

```
struct cryptEntry {
    char key;
    char unused;
    short length;
    const char* data;
}
```

Listing 3.2: The layout of an entry in the list of encrypted strings.

Panda decrypts the strings on the fly whenever a string is needed. The decryption routine for the  $i$ -th string is rather simple:

$$\text{output}[pos] = pos \oplus \text{encryptedStrings}[i].\text{data}[pos] \oplus \sim\text{encryptedStrings}[i].\text{key}$$

All encrypted strings are referenced in a large static array of structures in the read-only section of the binary. Each entry is a structure of type `cryptEntry` (see listing 3.2) which consists of the key character, the length of the encrypted string, and a pointer to the actual encrypted string. The decryption function then takes the index of the to-be-decrypted string in the array of structs, extracts the key, length, and string pointer from it and then decrypts the strings into a given buffer. Depending on how this function is used, it either decrypts the strings onto the stack (if the function is directly called) or the string is encrypted into the heap if any of the intermediate function is called. During the analysis we used the IDAPython plugin *idaemu* (frontend for UnicornEngine for use in IDA Pro) to emulate the encryption function for all possible string indexes and annotated the IDA database accordingly.

## 3.4 Cryptography

### 3.4.1 Random Numbers

Instead of using WinAPI functions to generate random numbers, Panda uses the Mersenne Twister MT 19937 to generate random numbers. Panda provides internal API functions to generate single numbers or buffers with support for upper and lower bounds for the numbers.

### 3.4.2 Cryptography

Additionally, Panda uses a set of cryptographic algorithms to encrypt and hash sensitive data to prevent analysis and manipulation of the data. For example, Panda encrypts almost all settings and configuration values in memory. The algorithms used are AES and RC4. Both of them are used either with a hardcoded or with a dynamic key (which is generated during the first run of the malware). Interestingly, both AES and RC4 share the same dynamic binary key material.

#### RC4 (static key)

- parts of the basic config that are double encrypted

- *PeSettings* in the *extended file attributes* of the malware executable (see section 4.2.2)
- object name generation (RC4 is used for scrambling there, no cryptographic purpose)
- encrypted data in dynamic config (e. g. backconnect IPs and ports for Vnc and Socks)

**RC4 (dynamic key)**

- local settings (see section 4.2.4)
- report data that is temporarily stored on disk until it is submitted to the command-and-control server

**AES (static key)**

- base config decryption (see section 4.2.1)
- internal public key decryption
- decryption of delay-loaded binary modules
- communication with command-and-control server

**AES (dynamic key)**

- registry data (dynamic config, local config; see section 4.2.3 and 4.2.2)

**3.4.3 Hashing**

Aside from encrypting data, Panda also uses some cryptographic hash functions.

**SHA256**

- DGA hostname generation (see section 4.4)
- bot ID (see section 4.1)
- object name generation
- integrity check of AES encrypted data sent by the command-and-control server

**SHA1**

- signature verification of the binary module data sent by the command-and-control server

## 4 Configuration

### 4.1 Bot ID

To be able to track and control each malware instance in the botnet, Panda generates a unique bot id. The bot id is a 32-byte hex string that can be described as

$$BotID \leftarrow HexString(SHA256(computerName||installDate||productId||versionInfo))$$

where

#### **computerName**

local computer name, fallback to "unknown" if error in `GetComputerNameW`

#### **installDate**

content of registry key `HKLM\Software\Microsoft\Windows NT\Current Version\InstallDate`

#### **productId**

CRC32 sum of the content of the registry key `HKLM\Software\Microsoft\Windows NT\Current Version\DigitalProductId`; fallback to 0 if failed getting key value

#### **versionInfo**

CRC32 sum of `OSVERSIONINFOEXW` where everything from (and including) `szCSDVersion` is zeroed out (`szCSDVersion`, `wServicePackMajor`, `wServicePackMinor`, `wSuiteMask`, `wProductType`, `wReserved`); fallback to CRC32 sum of `sizeof(OSVERSIONINFOEXW)` zeroes

Apart from identifying the bot, the bot id is also used as part of the algorithm that generates kernel object names (mutexes, window class names, event names, etc).

### 4.2 Configuration

Panda uses three different types of configurations: base, local, and dynamic. Each type of config has its own special purpose and is not available through static analysis – except for the base config.

#### 4.2.1 Base Config

For the initial configuration and the first connections to the command-and-control server, Panda contains a static base config with default settings for the most important configuration values. This includes the following values:

**dwDelayConfig**

delay in minutes how long to wait until malware starts to get the initial dynamic config

**dwRc4KeyLength**

length of the binary RC4 key

**szwDGAConfigUrls**

list of URLs suffixes for the DGA (see section 4.4)

**rc4Key**

binary RC4 key, used to encrypt the *PeSettings*

**dwDGAConfigUrlsLength**

length of szwDGAConfigUrls

**szwInitialCncHosts**

an encrypted, null-separated list of strings for initial command-and-control domains

**dwWaitAfterProcessInfection**

delay in minutes how long to wait for the core process to be initialized

**dwCncUrlCount**

number of command-and-control domains in szwInitialCncHosts

**dwCheckConfigDelay**

delay in minutes for next dynamic config check

### 4.2.2 Local Config (PeSettings)

The local config the data that is shared by all instances of the Panda malware on the local system and is generated only once at the first start of the malware and is then persisted in the malware executable using *Extended File Attributes*. The values of the *PeSettings* structure are as follows:

**dwStructSize**

the size of the structure

**szwBotId**

the ID of the bot that is used to identify the client against the backend server (see section 4.1)

**guid**

the GUID of the local system; if the malware is executed again after the first start, it recalculates the guid and checks if it matches the one from the *PeSettings*. If this is not the case, Panda aborts its execution. This can be used to check if the malware was moved to another PC after it was started once (e.g. copying a persisted sample

of the malware from a victim's computer to an analysis environment of a malware analyst)

**rc4BinKey**

this RC4 key is used to encrypt all data that goes to the registry keys (e.g. a backup of the currently used dynamic config)

**dwInfectionId**

a random number identifying the current infection

**szwCoreFile, szwReportFile, szwDynConfigFile, szwLocalConfigFile**

files on the local filesystem; szwCoreFile is the name of the malware executable; szwReportFile contains the path to the file where Panda temporarily stores the report data until they are sent to the server; szwDynConfigFile points to the file where the dynamic config is backed up on the filesystem; szwLocalConfigFile contains the file where the local config is stored

**regKey**

a random registry key name

**regDynamicConfig**

the name of the registry key that contains the backup of the current dynamic config

**regLocalConfig**

the name of the registry key containing a backup of the local *PeSettings*

**regLocalSettings**

the name of the registry key that is used to store the local settings into (e.g. IDs of socks and VNC modules)

### 4.2.3 Dynamic Config

The first thing Panda does after initializing and injecting into its run-time host process is to download a dynamic config from its command-and-control server. This configuration is created by the command-and-control server on demand and can change at any time. This allows the malware operator to maintain his control capability even in the event that the static configured command and control server is shut down. But especially the dynamic configuration is interesting for malware analysts because it contains the URLs and/or IP addresses of the ATS server(s).

Panda uses its built-in JSON parser to parse the dynamic configuration. The malware makes use of the following values:

**created**

the creation date of the config; used to check if the downloaded one is newer than the local one

**botnet**

the name of the botnet the client is part of

**check\_config**

time in seconds when to check for the next dynamic config

**send\_report**

time in seconds when to send the next system report

**check\_update**

time in seconds when to check for the next client update

**url\_config**

the url from where to download the next dynamic config

**url\_webinjects**

the url from where to download the webinjects

**url\_update**

the url for the bot update

**url\_plugin\_vnc32**

the url for the VNC32 module

**url\_plugin\_vnc64**

the url for the VNC64 module

**url\_plugin\_vnc\_backserver**

the URL/IP address where the VNC module should connect to

**url\_plugin\_grabber**

the url for the http grabber module

**url\_plugin\_backsocks**

the url for the backconnect socks proxy module

**url\_plugin\_backsocks\_backserver**

the URL/IP address where the socks backconnect proxy should connect to

**reserved**

encrypted data, from the context of the use of the data it seems that this is a list of fallback URLs for the download of the dynamic config (see section 4.4)

**grabber\_pause**

time in minutes how long to wait until starting the grabber module

There are some additional configuration values that can be provided which are not directly used by the sample, but probably used in one of the modules:

**grab\_softlist/grab\_pass/grab\_form/grab\_cert/grab\_cookie/grab\_del\_cookie/grab\_del\_cache**

flags denoting whether the grabber module should grab specific data or to delete some data (cookies, cache)

**dgaconfigs**

the url for the DGA config file; the DGA config file contains a list of URL suffixes which are appended to a generated string from where the bot will try to download the next dynamic configuration

**webfilters**

a list of URL masks where Panda can take special actions (see section 5.7)

**webinjects**

URLs, payloads, and location descriptions for the webinjects

#### 4.2.4 Local Settings

Additionally, Panda stores some run-time settings in a structure called *LocalSettings* by the malware authors. These settings are not meant to control the behaviour of the bot, it is more like a temporary data store of values that are client specific and need to be kept even after the malware is restarted (e.g. because of a system reboot). The structure contains the following values:

**dwModuleStartFlags**

bitmap denoting which of the modules has been started

**dwGrabberFlags**

bitmap denoting which of the http grabber features has been enabled

**dwPandaAntivirusFound**

set to 1 if Panda Antivirus was found, changes the behaviour of the bot update

**dwHashSet**

bitmap denoting which of the hashes has been set

**szConfigId, szWebinjectsId, szUpdateId, szGrabberId, szVnc32Id, szVnc64Id, szBacksocksId**

65-byte buffers to store the hashes of the respective files/modules

**dwCurrentUrlIdx**

the index of the currently used update URL in the list fallback URLs

**dwUrlRetryCount**

the retry count of the URL specified by *dwCurrentUrlIdx*; maximum value is set in the base config

**wBacksocksBackserverPort**

the port of the server of the backconnect socks proxy

**wVncBackserverPort**

the port of the server of the backconnect vnc module

## 4.3 Bot Update

Once persisted in the victim's system, Panda is able to update the malware executable by its own. In the usual case, Panda therefore downloads the new executable to a temporary file. The file is located in the directory returned by `GetTempPathW`. The name of the file is of the form `updXXXXXXXX.exe` where `XXXXXXXX` is the hexadecimal representation of a 4-byte random number. After writing the file and applying the *PeSettings* to the *Extended File Attributes*, the "update" is executed using `CreateProcessW` with `-f` as an argument flag. This triggers the "update" functionality of the bot so that all necessary settings are copied over to the new executable.

In the case of having Panda Antivirus present in the system, Panda overwrites the old malware executable in place and directly copies over the local settings instead of creating and executing a temporary file.

## 4.4 Configuration Update

One of the first things Panda does after initializing itself and persisting in the system is to download a dynamic configuration from the command-and-control server. To do so, Panda's base configuration (see section 4.2.1) contains a list of URLs from where to get the initial dynamic configuration. If the command-and-control server is already taken down at the time of checking, Panda cannot download a dynamic configuration and fails to exfiltrate any information. It still hooks all functions and gathers data (keystrokes, etc) but these information will never leave the system until the bot is able to download a (new) dynamic configuration.

The download routine for the dynamic configuration uses three different ways to get a dynamic configuration. First, it tries to get a dynamic configuration file from the URL provided in `url_config` in the old dynamic config. Of course, this only works if Panda already received a dynamic config once. If it did not receive a dynamic config at that point, it tries to get a configuration file from each of the command-and-control domains of the base config.

In case Panda is not able to download the dynamic config using the URL from the `url_config` field and the fallback command-and-control hosts (the malware allows for 5 failed retries for each of the domains), Panda takes the encrypted data from the reserved field, decrypts it, and tries to download a dynamic config from one of the URLs of that data.

If Panda is still not able to get a dynamic config at that point, it uses a domain generation algorithm to generate a possible hostname. Therefore, it takes the current system timestamp and modifies it a way that it stays the same for three days (set msec, sec, minute, hour to zero and subtract  $(dayOfMonth \bmod 3) * secsPerDay$  seconds from it). Then, Panda takes the built-in RC4 key to initialize a RC4 state and xors the timestamp onto it (first 8 bytes xor with plain timestamp, second 8 bytes with binary inverted timestamp) and calculates the SHA256 sum of the RC4 state. The result is then converted to a hex string and is used as the first part of the generated domain. The

second part of the domain is one of the domain suffixes from the base config and looks like "XX.tld/filename.ext" for the sample I analyzed. But the suffix can change and is not bound to any special requirements except for that it needs to make a valid domain from the generated name.

## 5 Payload and Persistence

### 5.1 Persistence

As part of the initialization procedure, Panda tries to persist in the following manner: First, it finds a suitable folder for the malware executable to reside in. In our case, it chose `%APPDATA%\Sun\Java`. It then moved the malware executable from the desktop to that folder and renamed it to `Desktop (Create Shortcut).exe`. Panda also creates three extra files with random file extensions which will be later used to temporarily store data. After moving the malware executable to the new folder, Panda adds a new value to the `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` registry key. This ensures that the malware is executed each time the infected user logs into the system. Additionally, it writes the initial `PeSettings` to `Desktop (Create Shortcut).exe` (see section 4.2.2).

### 5.2 HTTP Grabber and Injector

Since Panda is a banking trojan, its main purpose is to steal money from a victim's bank account and to grab login credentials for the bank accounts (and possibly other web services) wherever possible. A crucial part of its activity therefore is to intercept the web traffic of the victim's web browser(s) and to manipulate the content of the web page that is displayed in the browser. In order to achieve these goals Panda uses process injection (section 5.3) and API hooking (section 5.4). To know which web pages should be manipulated, Panda receives a list of URL masks and corresponding inject data. The inject data consist of the actual inject (script inclusion from attacker-controlled web server) and a description of the position where the inject has to be placed in the website. The included script is actually only a loader that loads the second stage of the inject which then communicates with the Panda web backend and does further modifications to the web page.

But there is a problem: today's web browser implement a feature called *content-security policy*. With (one of) the CSP header(s) sent by the web server, a website owner can tell the browser in detail, from where to load e.g. additional JavaScript code. Correctly configured, this hinders Panda to retrieve the second stage loader because it is loaded from a different web server. But since Panda is a man-in-the-browser malware, it can remove those headers from the server response and the browser will retrieve the loader.

Additionally, Panda removes the `TE` and `If-Modified-Since` headers from the request if the hijacked process is either Firefox or Chrome. This has two implications: web

servers will never send responses that have another transfer encoding than *chunked* (or no transfer encoding at all) and the server will always send a response that contains a HTTP response body. If Panda would not remove the **If-Modified-Since** header, a web server might send a response with a 304 status code and no response body content. Usually, this instructs the browser to use a cached version of the web page because the page content did not change since the last request (the time of the last request is specified in the **If-Modified-Since** header field). But since Panda intercepts web traffic between the raw socket and the handling of the browser, it cannot inject the malicious code into the response body because the web server never sent some. So, Panda must ensure that the web server sends a response body to be able to execute its injects. This can be achieved by removing the **If-Modified-Since** header and thereby simulating a fresh request to the web server.

Another thing Panda needs to take care of is *Accept-Encodings*. If the web server sends encoded data (e.g. gzip'ed), Panda will need to decode it to be able to analyze the response and maybe inject code. To avoid this, Panda simply changes (or adds) the **Accept-Encoding** request header to contain only *identity* which tells the web server to only send plain responses without any encoding at all.

Since Panda uses URL masks to detect which pages it should inject code into, it might happen that the masks match pages that do not contain valid HTML data (e.g. pictures, documents). In order to avoid those files, Panda checks the server response for specific Content-Types. Only if a valid content type is specified in the response header Panda tries to find injection points in the data. Valid content types are:

- text/
- application/x-javascript
- application/javascript
- application/xml
- application/xhtml+xml
- application/octet-stream
- application/json

Panda does not only inject data into web pages, it already grabs data at that point. If Panda finds any Authentication headers in the request, it checks for *basic authentication* and extracts username and password from it and adds it to the report. Additionally, Panda can extract all request data from GET and POST requests and reports them to the command-and-control server.

For a more detailed analysis on how the actual webinjects work and what the communication with the ATS looks like, please see our blogposts by Manuel Körber-Bilgard and Karsten Tellmann<sup>1 2</sup>

---

<sup>1</sup><https://cyber.wtf/2017/02/03/zeus-panda-webinjects-a-case-study/>

<sup>2</sup><https://cyber.wtf/2017/03/13/zeus-panda-webinjects-dont-trust-your-eyes/>

## 5.3 Process Injection

To apply its hooks, Panda needs to be part of each specific process space it wants to hook the functions in. In order to inject itself into the right process, Panda checks if the current targeted process fulfills some requirements:

- targeted process id  $\neq$  current process id ( $\rightarrow$  avoid injecting into its own process)
- targeted process owner = current process owner ( $\rightarrow$  avoid permission violation)
- the targeted process name must be one of: `firefox.exe`, `chrome.exe`, `iexplore.exe`, `panda.exe`, `MicrosoftEdge.exe`, or `MicrosoftEdgeCP.exe`

If all of those requirements are given, Panda injects itself into the process. This is done by allocating a virtual memory buffer of sufficient size in the target process using `VirtualAllocEx`. It then needs to relocate the copied binary because the old module base is most probably not the same it is in the remote one. If the relocation succeeded, Panda writes itself into that freshly allocated memory section. Afterwards, Panda copies over run-time data that has been modified by the infecting process during initialization and which is needed by the injected code.

After Panda successfully wrote all data into the address space of the targeted process, it creates a thread in this process. The thread continues to install the hooks and all execute all other necessary functions.

## 5.4 API Hooking Technique

As described in sections 5.5.1, 5.5.2, 5.5.3, and 5.5.4, Panda uses a hot-patch like function overriding method to hook its desired functions. Therefore, Panda overwrites the first 5 bytes of the function to contain a jump to its hook function. Because Panda needs to call the original function after doing its work in the hook function, it saves the overwritten instructions in a temporary buffer. For this purpose Panda has a built-in instruction length decoder. It then redirects the internal function resolver cache to point to that area (a so-called trampoline). Probably Panda does this to prevent an infinite recursion when the hook calls the hooked function. Interestingly, Panda searches its own IAT for hooked functions. However, as Panda has replaced importing through the IAT with the import resolver function (for most functions including all hooked functions) this has no purpose.

## 5.5 Hooks and Browser Manipulation

After Panda successfully injected into its target processes (see section 5.3), it starts hooking all necessary functions to provide banking trojan capabilities. The detailed technique is described in section 5.4 so this section focuses on the individual browser and how Panda implements its malicious activities.

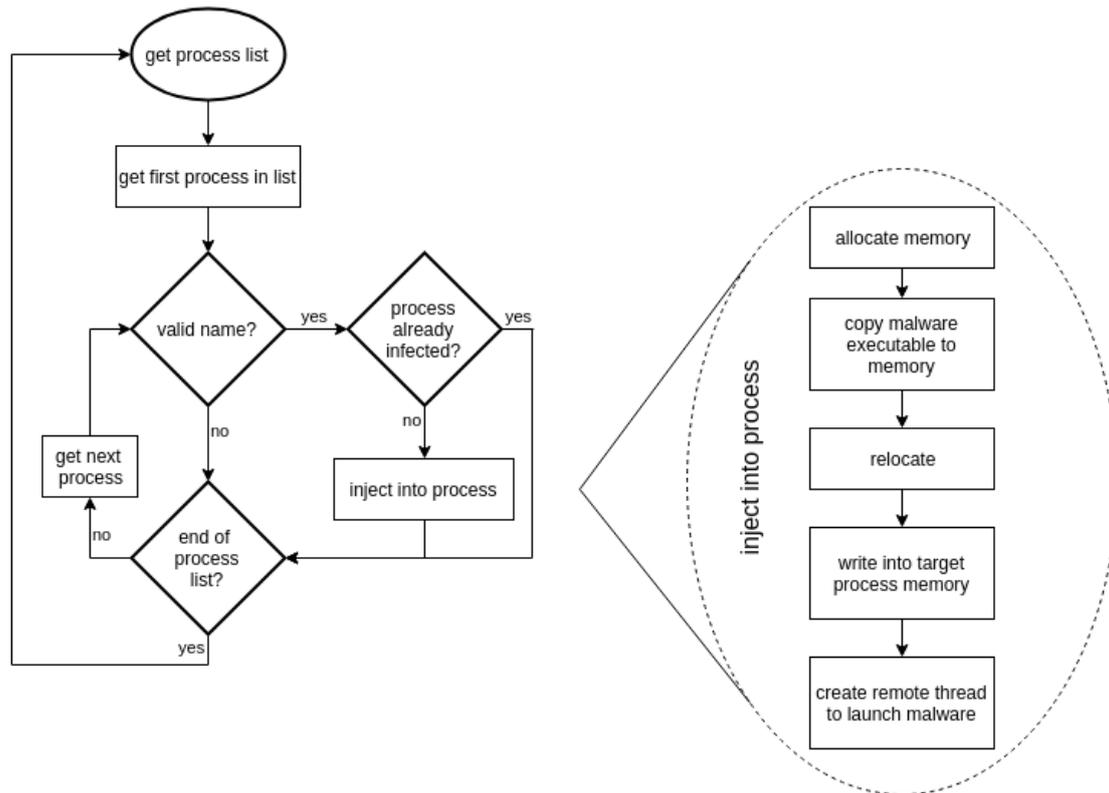


Figure 5.1: Flowgraph of the process infection thread.

### 5.5.1 Internet Explorer

Since Internet Explorer is a browser made by Microsoft, it vastly depends on functions from the Windows API and has no dependencies on third-party DLLs that need to be considered when hooking Internet Explorer. The actual hooks are done by overwriting some bytes in the function prologue (see section 5.4). The list of functions hooked by Panda is as follows:

- wininet!HttpSendRequestW
- wininet!HttpSendRequestA
- wininet!HttpSendRequestExW
- wininet!HttpSendRequestExA
- wininet!InternetReadFile
- wininet!InternetReadFileExW
- wininet!InternetReadFileExA

- wininet!InternetQueryDataAvailabe
- wininet!InternetCloseHandle
- wininet!HttpOpenRequestW
- wininet!HttpOpenRequestA
- wininet!HttpQueryInfoA
- wininet!InternetConnectW
- wininet!InternetConnectA
- wininet!InternetWriteFile

Additionally, Panda disables the phishing filter to avoid triggering it with the web injects, through modifying the following registry keys:

- HKCU\Software\Microsoft\Internet Explorer\PhishingFilter\Enabled
- HKCU\Software\Microsoft\Internet Explorer\PhishingFilter\EnabledV8
- HKCU\Software\Microsoft\Internet Explorer\PhishingFilter\EnabledV9

And it sets several internet zone policies to **allow** in order to get access to cookies and enable cross site script includes:

- URLACTION\_CROSS\_DOMAIN\_DATA
- URLACTION\_HTML\_MIXED\_CONTENT
- URLACTION\_COOKIES
- URLACTION\_COOKIES\_ENABLED
- URLACTION\_COOKIES\_SESSION
- URLACTION\_COOKIES\_THIRD\_PARTY
- URLACTION\_COOKIES\_SESSION\_THIRD\_PARTY

And finally it disables the “bad certificate” warning by modifying the registry key  
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\WarnonBadCertRecving

### 5.5.2 Mozilla Firefox

As described in section 5.5.3, Firefox uses a dynamically linked `NSPR4.dll`. This lowers the bounds for the malware to hook all necessary functions. Panda hooks the functions `PR_Close`, `PR_Read`, `PR_Write`, and `PR_Poll` by overwriting some bytes in the function prologue like it does for all Windows API hooks (see section 5.4).

Similarly to Internet Explorer, Panda modifies the user preferences the better fit the needs of the malware. In the case of Firefox, it walks through the profiles directory of Firefox's settings directory (`%APPDATA%\Mozilla\Firefox`) and sets the following user preferences to false:

- `privacy.clearOnShutdown.cookies`
- `security.warn_viewing_mixed`
- `security.warn_viewing_mixed.show_once`
- `security.warn_submit_insecure`
- `security.warn_submit_insecure.show_once`
- `security.warn_entering_secure`
- `security.warn_entering_weak`
- `security.warn_leaving_secure`
- `network.http.spdy.enabled`
- `network.http.spdy.enabled.v2`
- `network.http.spdy.enabled.v3`

### 5.5.3 Google Chrome

Hooking Google's Chrome browser is different compared to Firefox or Internet Explorer, because Chrome uses functions from both the Windows API and Mozilla's `NSPR4` library. The Windows API functions are as described in section 5.4. The difference between hooking Firefox and Chrome is that Chrome has a statically linked `nspr4.dll` instead of a dynamically linked one like Firefox has. Unfortunately, this has the consequence that one is not able to use `GetProcAddress` to get the address of the function and to overwrite some bytes at that address. However, Chrome internally uses a global struct of function pointers pointing to the actual functions. A pointer to this struct is shipped with each connection that is made by the browser. Panda tries to find the global struct and overwrites the function pointers in that specific struct to hook Chrome's `NSPR4` functions. The list of hooked functions (including Window API function) is as follows:

- `PR_Write` (`NSPR4` overwrite)

- PR\_Read (NSPR4 overwrite)
- PR\_Close (NSPR4 overwrite)
- closesocket (WinAPI-Hook)
- WSARcv (WinAPI-Hook)
- WSASend (WinAPI-Hook)
- recv (WinAPI-Hook)

#### 5.5.4 User Functions

In addition to the MITB hooks, Panda can also take screenshots, logs keyboard input, and watches for clipboard pastes.

To be able to log keyboard input, Panda hooks `TranslateMessage` for each process it is injected into. It then checks each windows message for `WM_KEYDOWN` and logs the (unicode) character representation of the pressed key. Additionally, Panda listens for `WM_MOUSEBUTTONDOWN` events and triggers a screenshot for each of the next 100 mouse clicks if a corresponding webfilter was triggered previously (see section 5.7 for a description of the webfilters).

Additionally, Panda hooks `GetClipboardData`. Hooking this specific function allows the malware authors to capture passwords that are not typed by the user but instead are pasted into the form fields in the browser (e.g. because the passwords are saved in a file on disk or because the user uses a password manager).

## 5.6 Plug-in ability

The Panda malware has the ability to dynamically load malware modules from web resources and to execute them in-place. This makes Panda a very flexible malware that can be retrofitted for other purposes.

Technically, they re-implemented `LoadLibrary` without the need of having the actual library on disk. First, the malware allocates enough space for the loaded DLL in the virtual memory of its process using `VirtualAlloc`. Afterwards, Panda section-wise copies the DLL into the previously allocated block of memory. Because DLLs are position independent, the third step is to relocate the sections. To achieve that, Panda walks through the relocation table (`.reloc` section) and resolves the required relocations by applying the base of the corresponding section to it. Panda also needs to resolve the imports of the module. The list of imports can be shortly described as a "what-where" list. For each of the entries in the list, Panda uses `LoadLibrary` and `GetProcAddress` to resolve the address of the imported function and writes it to the corresponding entry in the list. Finally, it calls the `DllMain` function of the loaded library to hand over control to the initialization function of the DLL.

Panda uses this technique to dynamically load its `HttpGrabber`, `Socks proxy`, and `VNC server` modules into the current process space.

## 5.7 Webfilters

Panda implements a feature that is called “webfilters” by the malware authors. Although, “filters” is not the correct term from my point of view. Consider `!http://*microsoft.com*` as an example for such a webfilter. The first character obviously does not belong to the actual URL although it should be clear that the exclamation mark stands for something like “not”. The position of the exclamation mark can be called “action” and is followed by the actual URL which can contain asterisks as placeholders for “any characters”. The full list of actions is as follows:

- P**  
report request content if request type is POST
- ^**  
block access to website and report the request content
- | (pipe symbol)**  
during my analysis I was not yet able to determine what this is used for
- @**  
takes a screenshot (500x500 pixels) on each of the next 100 mouse clicks (at max)
- !**  
don't write a report or analyze the data
- #**  
takes a screenshot (fullscreen) on each of the next 100 mouse clicks (at max)
- %**  
trigger the start of the VNC module (if not already started)
- &**  
trigger the start of the socks proxy module (if not already started)

## 5.8 Remote Script

In addition to the automatic information gathering, Panda provides a script-like interface where it can take several commands and performs actions on the victim's PC accordingly. Unfortunately, the script commands are hashed using CRC32 before comparing to the list of handlers so that we were not able to tell the names of the commands. But nevertheless we were able to determine the purpose of the commands by looking at their respective handlers. The possible actions the remote script can trigger, are:

- set shutdown flag**  
shutdown PC after the script finished
- set maintenance shutdown flag**  
shutdown PC in “minor maintenance” mode

**uninstall**

removes the bot from the PC

**update bot**

(force) updates the binary executable of the bot

**update config**

(force) updates the bot's dynamic configuration

**block or unblock webinjects**

allows for disabling or enabling certain webinjects

**list files matching a given path pattern**

searches the local file system for all files matching the pattern and adds the list to the report

**read files matching a given path pattern**

searches the local file system for all files matching the pattern and adds the content of the files to the report

**remove a local file**

deletes a file from the local file system

**execute remote file**

downloads and executes an arbitrary file

**block or unblock a given URL**

allows for blocking or unblocking a given URL so that the user can (or cannot) open the page in the browser

**enable HttpGrabber features**

grab passwords, forms, certificates, cookies (1+2), delete cookies (1+2), softlist, delete cache

**start VNC module**

(force) starts the VNC module

**start VNC module and set a flag in the local settings**

(force) start the VNC module and sets the appropriate flag in the local settings

**start socks module**

(force) starts the Socks proxy module

**start socks module and set a flag in the local settings**

(force) starts the Socks proxy module and sets the appropriate flag in the local settings

## 5.9 System Report

Each time Panda communicates with the command-and-control server, it sends status information about the bot back to the command-and-control server. The exact information depend on the type of the message sent to the server. But there are five groups of information that can be sent:

### **SYSINFO\_TIME**

- current system time (UTC)

### **SYSINFO\_USER**

- the name of the process executable where the control process resides in
- the current system user

### **SYSINFO\_BOTVERSION**

- bot ID
- the botnet the client is part of
- the version of the bot

### **SYSINFO\_OS**

- system version (e. g. 6.1 for Windows 7)
- service pack number
- build id
- architecture (32/64 bit)
- server edition?
- default ui language

### **SYSINFO\_MISC**

- network latency
- localized time
- computer name
- installed antivirus, antispysware, and firewall products

## 6 Conclusion

Panda must be considered to be among the more advanced types of malware. The code basis is large and sports a number of features not found in less sophisticated malware. These features include extensive anti-analysis code and an advanced hooking framework in which Panda brings, among other things, its own instruction length decoder. The code seems to be mature and the quality of the code appears to be above the average for malware.

The main purpose of Panda is to serve as a banking trojan. Therefore its author equipped the malware with sophisticated capabilities and supports all major browsers in the Windows ecosystem. However, Panda shows significant flexibility allowing it to be used for other malicious purposes. For example, Panda implements a modifiable configuration that can be changed at any time by the attacker. Additionally, Panda is able to spy on user activity, provides a remotely accessible scripting language, and has the ability to load a VNC server and a SOCKS proxy module to provide additional remote access features to the attacker.

Thus, the Panda trojan family remains a considerable threat even six years after the Zeus source was made public.