Clash of the Titans: ZeuS v SpyEye

Harshit Nayyar
Clash of the Titans: ZeuS v SpyEye

GIAC (GREM) Gold Certification

Author: Harshit Nayyar, harshit.nayyar@telus.com
Advisor: Pedro Bueno

Accepted: 2010

Abstract
In this paper, we discuss details of two recent bots - ZeuS and SpyEye, and study their interaction. We describe how to reverse engineer the two binaries and compare the obfuscation and anti-debugging techniques used by them. Since there is already much literature describing their individual malicious capabilities, this paper does not focus on those aspects. Instead, the focus is primarily on the inter-process communication between the two bots - which is a relatively rare phenomenon in the world of malware.
Introduction

The stage, it seems, is set for an epic battle between two of the most dangerous fighters in the nefarious world of malware. In one corner: ZeuS, undoubtedly the reigning champion of Banking Trojans, so much so, that the distinction of “king” has often been used to describe it (Falliere & Chien, 2009). In the other corner: SpyEye, a relatively new, but at the same time worthy, challenger posing to dethrone ZeuS. This paper documents a part of this budding and dynamic battle as it unfolds – so dynamic in fact, that within the time it took to write this paper, both crimeware kits had already moved on to their next releases, implementing some serious licensing and anti-reversing measures (Krebs, 2010).

The skinny from reports is that SpyEye is challenging the supremacy of ZeuS by threatening to remove ZBot from infected systems.\(^1\) It has received much attention (Krebs, 2010; Coogan, 2010; Mieres, 2010; McMillan, 2010; Kelchner, 2010; Paz, 2010) recently due to its ability to “Kill ZeuS”. Has SpyEye really discovered the Achilles’ heel of ZeuS? And is all this media hype justified? After all, there have been others such as Opachki (Stewart, 2010), which also claim to be able to remove ZeuS from an infected host. Why then, should SpyEye receive such attention? Well, in this case, as is often seen, the devil lies in the details. This paper shows that SpyEye uses a far superior technique for removing ZeuS than other malware. Whether SpyEye is actually successful at removing the large diversity of ZeuS variants that have appeared over the years, remains to be seen. However, the technique in itself is rather interesting and has been discussed in-depth in Section of this paper. But first, let us get to know more about the stage and the contestants.

The Stage: Reversing Tools and Environments

During the course of this research we have used many tools – most of which are mentioned in this section. Even though, output from all of these might not appear in text of the paper - this was done to conserve space and not to devalue the importance of the

\(^1\) Zbot is the name given to bot binaries belonging to the ZeuS botnet(s). However, in this paper, we use the term Zbot and ZeuS interchangeably.

Harshit Nayyar, harshit.nayyar@telus.com
Virtual Machines

Virtual machines provide a safe sandbox for executing and analyzing malware. In addition, they also provide the ability to take snapshots and to revert to an existing snapshot at a later stage. This makes both static and dynamic analysis easier. For this research, we used VMWare ESXi Server version 4.0 in order to host our Virtual Machines. We used two virtual machines:

1. The Infected box, which is a Windows XP SP2 version 5.1.2800 virtual machine with single 500MHz processor with 512 MB Ram and 4 GB disk space. The host file (c:\Windows\System32\drivers\etc\host) was modified to redirect traffic from the bots to the second VM described below. (Figure 1)

2. The C&C box, which is a Windows 2000 Server SP4 version 5.0.2195 virtual machine. We are running a fake server script written in python. This script is given in the section entitled “Fake Server Script”. (Figure 2)
Figure 1: Victim/Infected VM running Zeus And SpyEye
Dynamic Analysis Tools

Regshot: Shows Registry and File changes between two of its snapshots (TianWei, 2008).


Process Explorer: Tool to display process information (Russinovich M., 1998).


Code Analysis Tools

OllyDbg: Debugger for Win32 binaries (Yuschuk, 2000).


IDA Pro: Disassembler and debugger with graphing and visualization capabilities (Datarescue sa/nv, 2006).

Harshit Nayyar, harshit.nayyar@telus.com
**Miscellaneous Tools**

- **LordPE**: PE dumping and editing utility (mmi, 2009).
- **PEiD**: PE File identification tool to identify file type, packers, compilers etc. (Jibz, snaker, & Qwerton, 2005).
- **Python**: General Programming Language (Rossum, 2000).
- **VirusTotal**: Online Scanning of Binaries (Hispacec, 2008).


ZeuS : The Reigning Champion

The sample used to analyze ZeuS botnet was found in the wild from a public source. Here are some details about the sample:

Identifiers:
- **MD5**: 3cfc97f88e7b24d3ceced4ba7054e138
- **SHA1**: b7a242bf3138a090ab977982b65850d54b540893
- **AV Names**: Trojan.Zbot [Symantec], TROJ_SBOT.SVR [TrendMicro], Trojan-Spy.Win32.Zbot.aeth [Kaspersky]

The output is shown in Figure 3 although PEiD is not able to identify the packer, some of its heuristics indicate that the binary is packed.

![Figure 3: PEiD Output For ZBot](image)

---

2 A more comprehensive list of AV names is given in Appendix

Harshit Nayyar, harshit.nayyar@telus.com
Inspecting the file in OllyDbg shows the common dialog box shown in Figure 4.

![Compressed code?]

Figure 4: Common OllyDbg Dialog.

Moreover, the absence of meaningful referenced strings in code confirms the packing (Figure 5).

![Text strings referenced in exe]

Figure 5: All Referenced Text Strings

Since the binary is packed, the packer is not known and no automated unpacking is possible, we unpacked the binary manually. During the process, we found the following anti-debugging technique:

```
... 00403A2A .>PUSH DWORD PTR SS:[EBP-8] ; /hObject <- Push invalid handle
00403A30 .>CALL <JMP.&KERNEL32.CloseHandle> ; \CloseHandle
```

This technique is well documented (Falliere, 2007) and involves trying to close an invalid handle, which, in presence of a debugger, gives STATUS_INVALID_HANDLE (0xC0000008) exception. To bypass it, we patched the binary by replacing these instructions with NoOps (0x90). Once this check is bypassed, the binary does not give much more trouble. After a series of cross section hops, each section, contributing to unpacking, the binary is finally unpacked. However, even after unpacking, no meaningful referenced text strings are seen in the code. This is because strings are obfuscated. They are present in the binary in a form where first byte gives the string size and the remaining bytes are encoded. The following figure shows the code used to de-obfuscate strings on

Harshit Nayyar, harshit.nayyar@telus.com
the fly (Figure 6):

```
0bfuscated_Str= dword ptr 8
Destination= dword ptr 4Ch

push esi
mov esi, [esp+0bfuscated_Str]
xor ecx, ecx
xor al, al
cmp [esi], cl
push edi
mov edi, [esp+Destination]
je short loc_4099A7
```

After unpacking, the Trojan creates the import table by loading the necessary DLLs using Kernel32!LoadLibrary and getting the address of needed procedures using Kernel32!GetProcAddress.

ZBot is known to create Mutexes to mark different stages of execution. At this initial stage, our sample creates the following Mutex as shown by the code in Figure 7:

```
loc_40998E:
    mov bl, [ecx+esi+1]
    add bl, dl
    inc al
    mov [ecx+edi], bl
    movsx ebx, byte ptr [esi]
    movzx ecx, al
    add dl, 2
    cmp ecx, ebx
    jl short loc_40998E
```

Figure 6: String Decoding Loop as shown by IDAPro

Figure 7: Main Dropper’s Mutex Creation
Then it checks if another Mutex is present as shown in Figure 7. This Mutex is created at a later stage in execution. This check ensures that the host is not already infected (Figure 8).

![Figure 8: Check for Existing Mutex](image)

The Trojan then checks for the presence of the processes associated with firewall software as shown by the code in Figure 9:

![Figure 9: Outpost and ZoneAlarm Check](image)

Next, it creates a copy of itself in the following location:

```
%system%\ntos.exe
```
It sets the attributes of the file created to `SYSTEM\ARCHIVE` and then adds a random amount of random data at the end of the file as shown by the code Figure 10. This ensures that the dropped file has a different MD5/SHA1 hash each time it is created.

![OllyDbg - zeus.exe - [CPU - main thread, module zeus]](image)

**Figure 10: Append Random Data**

It then appends the name of the newly created file to the end of the key:

`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\userinit`

This ensures that the malware executes upon windows start-up as shown in the Figure 11.
Now that the binary is installed on the system, it starts the infection process by injecting code into the WinLogon.exe process (Figure 12, Figure 13):
Once the code injection is complete, the main process goes into an infinite loop waiting for the creation of a certain Mutex, which will be created at a later stage in the infection process. This loop along with the name of the Mutex is shown in the Figure 14.

Now let’s inspect the injected code. The injected code creates a new directory as
shown in Figure 15.

![Figure 15: Create wsmopem Directory](image1.png)

And then creates a file named “video.dll”. This file is later used to save the downloaded configuration file (Figure 16):

![Figure 16: Create video.dll file](image2.png)

It also creates a file named “audio.dll” which is used to store the stolen data to be sent to the server (Figure 17).
Next, it creates a Named pipe, which is later used for communication of this component with other ZBot components (Figure 18).

Then it creates a thread, which is responsible for Inter-Process Communication between various ZBot components. This thread first creates a Mutex (Figure 19). Note that the presence of this Mutex was checked earlier to ensure that the system is not already infected. Also note that the name of the Mutex is same as the name of the Named pipe created earlier. The importance of this fact will be discussed when in Section covering the interaction of SpyEye with ZeuS. This thread then connects to the Named Pipe and waits for a client to connect to it.
Meanwhile, the main thread, running the code injected into winlogon.exe, checks for the presence of a Mutex. Note that this Mutex is same as the one for which the dropper process is waiting. If this Mutex does not exist, the code in winlogon.exe, does another code injection – this time into svchost.exe using the same techniques described before.

The code injected into svchost.exe creates another Named pipe (Figure 20) and then creates a thread, which is responsible for Inter-Process Communication via Mutexes and Named Pipes. This thread creates another Mutex with the same name as the pipe. Note that this is the same Mutex for which the original dropper process is waiting, as discussed earlier. After that, the thread connects to the Named Pipe and waits for a client to connect to it and then send data.
The injected code in svchost.exe is responsible for network communication. It sets up a connection with the remote server and downloads an encrypted configuration file. A lot has already been written about network communication of ZeuS (Shevchenko, 2009). Therefore, in order to save space, we have included the configuration file format and encryption scheme in Appendix.

As mentioned previously, the main binary is waiting in an infinite loop for creation of the Mutex, which is finally created by the code injected to svchost.exe. Once this Mutex has been created, it breaks out of that loop. It steals cookies and other information from Macromedia Flash *.so files, saving the stolen data into the file audio.dll. It also connects to named pipes created by other components and communicates with them as discussed later. Finally it terminates leaving other components injected into processes to steal data, and communicate with each other via their individual pipes.

Since we have Zbot installed and running, happily stealing information and passing it on to its server, we can now discuss a little bit about the named pipe based communication technique of Zbot. The function responsible for this communication takes a single parameter as input. This parameter is a pointer to a structure which, based on code analysis, seems to have the following format:
<table>
<thead>
<tr>
<th>Offset</th>
<th>Size(Bytes)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>4</td>
<td>hPipe : Named Pipe Handle</td>
</tr>
<tr>
<td>0x000</td>
<td>4</td>
<td>hEvent1 : Handle to an Event</td>
</tr>
<tr>
<td>0x000</td>
<td>4</td>
<td>hEvent2 : Handle to another Event</td>
</tr>
<tr>
<td>0x000</td>
<td>4</td>
<td>fPtr: Function pointer of Command Processor function</td>
</tr>
<tr>
<td>0x001</td>
<td>4</td>
<td>szMutexName: Pointer to Mutex-Name String</td>
</tr>
</tbody>
</table>

This function does the following:

1. Create a Mutex with the name szMutexName (Figure 22)

2. Set hEvent2 (Figure 22)

3. Wait for hEvent1 to be set (Figure 22)

4. While hEvent1 is set loop:

```
; Attributes: bp-based frame

CommunicateWithNamedPipe proc near

hmMutex= dword ptr -10h
var_C= dword ptr -0Ch
Buffer= dword ptr -8
var_A= dword ptr -4
arg_0= dword ptr 8

push ebp
mov ebp, esp
sub esp, 10h
push esi
mov esi, [ebp+arg_0]
push edi

push dw [esi+1ah]; <- szMutexName
add edi, edi
push edi

push offset MutexAttributes
call ds:CreateMutexV

push dw [esi+8]; <- hEvent2
mov [ebp+hmMutex], eax
call ds:SetEvent

push dw [esi]
call ds:DisconnectNamedPipe

push dw [esi+4]; <- hEvent1
call ds:WaitForSingleObject

test eax, eax
jz loc_409578

Figure 22: Zeus IPC Steps 1-3
```
a. Connect to the Named Pipe – This is a blocking call, which waits till a client connects. (Figure 23)

b. Once a client is connected, read the command code from the named pipe. (Figure 23)

c. Read the size of parameters of the command. (Figure 23)

d. If the parameter size is higher than 0x10 (16):
   i. Allocate memory for the parameter.
   ii. Read the parameter into the allocated buffer.
e. Call the command processor function with the command code, the parameter and the parameter size. (Figure 25)

f. Write the response to the Named Pipe. (Figure 25)
g. Disconnect from the Named Pipe. (Figure 26)
5. Close Mutex Handle. (Figure 26)

6. Set hEvent2. (Figure 26)

7. Return. (Figure 26)

During code analysis we found two kinds of command processors:

1. Command Processor to handle commands between 0x01 and 0x0E
2. Command Processor to handle commands 0x03, 0x0F through 0x11

These commands can be used to gain information, such as the current path of Zbot binary (Command Code: 0x0B) or to Request for some action such as close handle of the Zbot config file to allow another component to write to it (Command Code: 0x06). The code in Figure 27 shows processing of Command Code: 0x06, which causes the closure of the handle to the config file (“Video.dll”).
At this point, our introduction to ZeuS is more or less complete. However, the above description of ZeuS must follow a “Mea culpa” of sorts, since it does not come close to describing the complete functionality of Zbot – unfortunately that would take considerably more time and space than what the scope of this paper allows. That said, it is now time to move on to the other side of our metaphorical battlefield and get introduced to SpyEye.

**SpyEye: The challenger**

The SpyEye sample was also found in the wild.

Here are some details about the sample:

<table>
<thead>
<tr>
<th>Identifiers:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5:</td>
<td>84714c100d2dfe88629531f6456b8276</td>
</tr>
<tr>
<td>SHA1:</td>
<td>eece2684f143b02fe187a4a6af22f1e9e6c2c6f</td>
</tr>
</tbody>
</table>

PEiD output for the sample is shown in the Figure 28. It appears that the sample is

---

3 A more comprehensive list of AV names is given in Appendix

Harshit Nayyar, harshit.nayyar@telus.com
not packed. Figure 29 shows the list of inter-modular calls and text strings found in OllyDbg. While the text strings show many DLLs, the inter-modular calls do not show any method being called from them. This seems quaint.

Figure 28: PEiD Output For SpyEye Sample

Harshit Nayyar, harshit.nayyar@telus.com
The reason becomes clear as further code is inspected. The first function to be called returns the OS version. To understand how SpyEye does this, let us consider the code shown in Figure 30. The returned value is the OS version. So somehow, the 4-byte value passed (0x9C480E24) is being converted into a function that returns the OS version and that function is being called. Clearly some hashing scheme is used wherein the key 0x9C480E24 is converted into a function that can return the OS Version. Such obfuscation techniques are discussed in some books and papers (Eagle, 2008; Skape, 2004).

Upon further inspection, we found the code snippet shown in Figure 31.

![Figure 30: SpyEye Win32API Calls via Hashes](image)
The following python code explains the algorithm:

```
#!/usr/bin/python
import sys

def nameToHash(name):
    j = 0
    for i in range(0,len(name)):
        left = (j << 0x07) & 0xFFFFFFFF
        right = (j >> 0x19)
        j = left | right
        j = j ^ ord(name[i])
    return j
```

Hence, we are now able to explain how the function name is converted to hash. To find the appropriate function, SpyEye converts the name of each exported function to its hash and matches this value with the hash passed as the function argument. If the value matches, the address of the exported function is returned. To make further analysis easier,
we wrote a python script given in Appendix, which can return the name of the function given the hash value. Using this script, the function name corresponding to 0x9C480E24 was found to be Kernel32!GetVersionExA. This explains the mystery of how an API function is identified based on its hash.

So now let’s see how the exported function gets called. We assumed that if we put a breakpoint on this function i.e. Kernel32!GetVersionExA, it would be hit and we could see the stack-trace and find where the call is made – but to our surprise we found that the breakpoint is never hit. Looks like SpyEye goes to great lengths to avoid such analysis in a debugger. Upon further static analysis we found that it uses a novel technique that has not been observed in common malware. Almost all calls to common Windows DLLs have been replaced with a special code that maps a DLL into a new section in the memory and find the address of the function to be called within that section and then calls that function. The general technique used to accomplish this is as follows:

1. Select the library implementing the function.
2. Find the base of Ntdll.dll.
3. Find address of the procedure LdrLoadDll within Ntdll.dll.
4. Load the library, whose name was found in Step 1, using LdrLoadDll.
5. Find the function name from the hash value as discussed before and shown in Figure 31.
6. Find the offset of the function inside the loaded DLL.
7. Pass the DLL name and the function offset to a function that does the following:
   a. Open the DLL file using ntdll!NtOpenFile
   b. Create a new section using ntdll!NtCreateSection
   c. Map the file into the newly created section using ntdll!MapViewOfSection. Hence the DLL is mapped into a new section in memory.
   d. Find the function address inside the new section from the offset passed. Return this address.
8. Now function at address returned by step 7d is called using the appropriate arguments.
9. After the function call, the DLL is unmapped.

Harshit Nayyar, harshit.nayyar@telus.com
The code showing this implementation is given in Appendix. Astute readers will quickly realize that the malware must need a different technique than the one described above to call the methods, such as ntdll!NtOpenFile used to determine the function addresses. This is absolutely correct. The Trojan has functions that use a slightly different approach to finding these native API functions. For these functions, the code finds the method name from hash and then calls them without mapping ntdll into a new section – since they are themselves needed for the process of mapping.

Once this mechanism was found, we searched the code for all calls, which were made in this manner. Using the script given in Appendix, we added labels to code where we found such calls being made. These labels took the form of “[API FunctionName]Caller”, for example, “CreateDirectoryACaller” as shown in Figure 32. Over all, close to 50 such labels were added.

![Figure 32: Some Labels Added in IDAPro](image)

Stepping further into code we also encountered the code snippet shown in Figure 33.
The function takes a 4-byte value and returns a string (in this case explorer.exe). Hence, the call is somehow converting the 4-byte hash value into a string. Following the code we found that the hash value is used as an index into a table. The table points to strings of bytes in memory. This is actually an ASCII string in its encoded form. The decoding algorithm is shown in Figure 34.

Simply stated, it consists of starting from the end of the string and subtracting each character from its previous one, while all changes are made in place. Here is an encoded string:

```
00405AE0 65 DD 4D B9 28 9A FF 71 9F 04 7C E1
```

Using the algorithm discussed above, we can see that the last character should be

```
0xE1 - 0x7C = 0x65 ('e')
```

Decoding whole string in this manner, we find that the string is “explorer.exe”:

```
00405AE0 65 78 70 6C 6F 72 65 72 2E 65 78 65
```

Having understood both the code and string obfuscation mechanisms, the code can now be analyzed in detail. The main task being performed can be described in the following Steps as shown in flowchart of Figure 35:

1. Try to Create Mutex named “__CLEANSWEEP__”

Harshit Nayyar, harshit.nayyar@telus.com
2. If Mutex already exists, it means the system is already infected, go to step . Else go to Step.

3. Try to inject code into “explorer.exe”. If injection succeeds, go to step . Else go to Step.

4. Try to inject code into any process. If injection succeeds, go to step . Else go to Step.

5. All injection attempts have failed, so just call the injected code in this current process itself. Then go to Step.

6. Call ExitProcess
Since following code injection into explorer.exe is tedious, we patched the code in...
OllyDbg to directly go to code that should have been injected into Explorer. This happens to be the dropper code that creates a directory with the name `c:\cleansweep.exe` and inside the directory creates a file `cleansweep.exe` and a file `config.bin`. Creation time of all of these files is set to the creation time of `ntdll.dll`. It then opens the dropped file as a sub-process of itself. To save space, we are not showing that full code here, but the final call to CreateProcessA (CreateProcessACaller to be precise) is shown in Figure 36.

![SpyEye Dropper launching dropped binary - CreateProcessA call](image)

Some information regarding the dropped executable is shown below:

**Identifiers:**
- MD5: d1d591f21543f25e203054b73c07ff58
- SHA1: 7761b2666656ab5748128954de17bd9136257ad8

This binary is unpacked and has strings that are mostly un-obfuscated. Some important referenced strings are shown in Figure 37, Figure 38, Figure 39 and Figure 40.

---

4 A more comprehensive list of AV names is given in Appendix

Harshit Nayyar, harshit.nayyar@telus.com
This is a GIAC Gold Template

Figure 37: SpyEye Strings: Mutex, Config Password, POST request and parameters

Figure 38: SpyEye Strings: Server Commands, More Mutexes

Figure 39: SpyEye references ZBot like named pipes

Harshit Nayyar, harshit.nayyar@telus.com
To better control the new process being created, we terminated the existing process before it calls CreateProcessA. We then launched the process (cleansweep.exe) independently of the dropper in a debugger.

The main loop of this binary does the following tasks:

1. Check that the OS version is higher than 5 (Figure 41). Else exit.

2. Create a Mutex to mark presence and confirm that the host has not already been infected (Figure 41).
3. Read the config.bin file created in the same folder by the dropper (Figure 42). The config.bin is actually just a password protected zip file whose first two bytes (in the header) have been replaced with 0xFF. If the first two bytes are changed to “P” and “K”, making the file a legitimate PKZip archive, the file seem to be a password protected archive. The password is known from the strings seen earlier. Upon extraction, the file config.bat is retrieved. Figure 43 shows the contents of the config.bat file. It also shows the extracted URL.
4. Try to inject code into `explorer.exe`. If this succeeds the process terminates. Else go to step 5. (Figure 44)

5. Try to inject code into any process. (Figure 44)

6. Exit (Figure 44)

We followed the second injection, this time done by the dropped binary `cleansweep.exe`, into `explorer.exe`. The injected code was dumped using the LordPE and

Harshit Nayyar, harshit.nayyar@telus.com
opened in IDAPro for static analysis. The function is shown in Figure 45. The following steps describe the basic functionality:

1. Create Mutex named __CLEANSWEEP__ (Figure 45)

2. Create a thread that monitors the existence of Mutex called __CLEANSWEEP_UNINSTALL__. This Mutex stops SpyEye when an updated binary is downloaded from the C&C server. (Figure 45, Figure 46)

3. Create an auto-run registry key to ensure that cleansweep.exe is executed upon Windows startup as shown in Figure 46.

4. Do further code injection into any process – this generally ends up being the generic host process svchost.exe. The injected code seems to do further code injections and other malicious functions. (Figure 45)

5. Create a thread which initiates C&C communication with server sending some information such as a GUID, the version of the bot, the version of Internet Explorer, User-Type, Cpu Load, Config file CRC etc. Also create another thread which monitors existence of the Mutex __CLEANSWEEP_UNINSTALL__. (Figure 45)

6. Get current module name and check for if contains *explorer.exe*. If it does go to step 8. Else go to step 7. (Figure 45)

7. Call function to create threads, which implement other malicious functionalities including a user-mode rootkit component to hide files and registry keys created. Since a comprehensive description of malicious functionalities is out of the scope of this paper, these are not described here. To give an idea of the complexity and number of functional components involved, please refer to the Figure 47, obtained from an article based on a conversation with the SpyEye creator (Koehl & Mieres, 2010). It also creates a thread which monitors the existence of the Mutex __CLEANSWEEP_UNINSTALL__. Go to Step 8.

8. Deal with ZeuS and Windows Defender. We will discuss interaction with
ZeuS in Section of this paper. Figure 48 shows SpyEye disabling windows defender.
Figure 46: SpyEye AutoRun Registry Creation

Figure 47: SpyEye Components from Older Version. Reproduced from (Koehl & Mieres, 2010)
Before we move on to the topic of interaction, let us take a moment to summarize and compare the features of the ZBot and SpyEye that we have discussed so far.

Feature Comparison

Now that we have discussed some of the features of ZeuS and SpyEye, we are in a position to make a comparative analysis of their features as discussed in (Figure 49). Note that this comparison is based on features observed in the analyzed samples – other samples may show different behavior. Moreover, this table is a small subset of features and do not include their malicious capabilities such as key logging and form grabbing.

<table>
<thead>
<tr>
<th>Feature/Bot Sample</th>
<th>SpyEye</th>
<th>ZeuS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packing</td>
<td>The spyeye sample analyzed was not found to be packed.</td>
<td>The ZeuS sample analyzed was found to be packed with an unknown packer</td>
</tr>
<tr>
<td></td>
<td>SpyEye has a complex string obfuscation technique and code refers to strings by means of a hash, which is passed to a function that returns a string.</td>
<td>ZeuS has a relatively simple string obfuscation technique.</td>
</tr>
<tr>
<td>Code Obfuscation</td>
<td>SpyEye has a code obfuscation technique where almost all Win32 API calls are replaced with a complicated scheme making code analysis quite hard.</td>
<td>ZeuS sample analyzed did not have any obfuscation other than packing.</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Debugger Detection</td>
<td>Although SpyEye is known to have some AntiDebugger tricks, we did not come across them in our analysis. This might be because we did not follow all code injections.</td>
<td>ZeuS sample was found to have an anti-debugger technique involving Invalid Handles.</td>
</tr>
<tr>
<td>Virtual Machine Detection</td>
<td>SpyEye sample was not found to have any VM Detection techniques.</td>
<td>ZeuS sample was not found to have any VM Detection techniques.</td>
</tr>
<tr>
<td>Anti AV/Firewall Techniques</td>
<td>SpyEye has a routine, which disables Windows Defender product from Microsoft.</td>
<td>ZeuS sample has a function, which disables Windows, Zone Alarm and Outpost firewalls.</td>
</tr>
<tr>
<td>Dropper Routines</td>
<td>SpyEye sample arrives in the form of a dropper, which drops and installs the sample.</td>
<td>ZeuS sample was also seen to arrive as a dropper, which drops and installs the sample. However, it also appends random amount of random data to ensure that the hash and file size is different for each sample. ZeuS uses the &quot;userinit&quot; subkey of: HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Winlogon. ZeuS also injects its components into many different processes such as svchost.exe, winlogon.exe and csrss.exe. ZeuS uses Mutexes as well as Named Pipes for IPC</td>
</tr>
<tr>
<td>Autorun Technique</td>
<td>SpyEye uses the key: HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Run</td>
<td></td>
</tr>
<tr>
<td>Code Injection</td>
<td>SpyEye does many code injections. Even the dropper code is injected into explorer.exe, which leads to several other code injections.</td>
<td>ZeuS sample does not have a default configuration. It needs to get its configuration from the C&amp;C server. The sample analyzed uses the nrv2b algorithm along with non-key specific encoding. However, it is known that newer versions of ZeuS</td>
</tr>
<tr>
<td>Interprocess Communication (IPC) Configuration File</td>
<td>SpyEye sample seems to implement IPC using Mutexes only.</td>
<td></td>
</tr>
</tbody>
</table>

Harshit Nayyar, harshit.nayyar@telus.com
now use a private key based encoding scheme similar to RC4.

Figure 49: SpyEye v ZeuS - Feature Comparison

The battle begins.

Now is the moment we’ll all been waiting for – seeing the malware samples exchange blows or at-least one of them knocking out the other. Let’s turn to some dynamic analysis first:

Test 1:

The Figure 50 shows the response of our fake web-server (Appendix ) in a scenario where a system is first infected with Zbot and then restarted. Initially Zbot sample sends a request to download its configuration file and then sends some POST requests. Upon restart, the request for configuration file and subsequent POST requests are seen again. As expected, Zbot is still present on the victim’s host. Now we run the spy-eye sample which sends an initial website checking request and a GET request. The victim host is restarted – to check if Zbot is still present. The absence of request to download the ZBot configuration file shows that it has been removed from the system.
Test 2:

In this test, as shown in Figure 51, we first the SpyEye sample is run. It sends its requests and the server responds. Next Zbot is run – it also sends its requests. The two bots seem to co-exist peacefully – hence SpyEye does not in any way prevent Zbot from being installed. Upon restarting, however, Zbot requests are not seen any more. This indicates that Zbot has been removed. This is because upon system restart, SpyEye gets executed before the Zbot sample and removes it.
As discussed before, and shown in Figure 45, in step 8, a call is made to a function responsible for removing ZeuS and disabling windows defender. This function call is shown in Figure 48. In this section, we will focus on the function, which implements the “Kill ZeuS” functionality. This functionality involves the following steps:

1. Get the names of ZeuS Binary and Mutex by sending the command 0x0B to the ZeuS named pipes. (Figure 52)
2. Stop the threads of ZeuS by sending them Kill Command (0x03). (Figure 52)
3. Close handle of ZeuS Mutex (since we duplicate the handle). (Figure 53)

4. Delete the binary whose path was obtained in step 1 as shown in Figure 54. (Figure 53)

Before we go into more details for each of these steps, let us first understand how SpyEye connects to named pipes created by ZeuS. It uses the following steps to communicate with ZBot named pipes:

1. Get a list of all open handles for each process in the System. (Figure 55)
2. For each process, for each handle:

   a. Get the name of the handle. (Figure 56)

```
loc_4099788:
    push 0
    push [ebp+bufferLen]
    push [ebp+SystemHandleInfoListPointer]
    push 10
    ; SystemHandleInformation
    call ds:NTQuerySystemInformation
    mov [ebp+status], eax
    cmp [ebp+status], 00000000h
    jnz short loc_40970C
```

Figure 55: Zeus Communication Step 1

   b. Find if the handle-name has any of the following substrings as shown in Figure 57:

      i. __SYSTEM__

      ii. __AVIRA__

```
loc_40992F:
    lea eax, [ebp+bufferLen]
    push eax
    push [ebp+bufferLen]
    push [ebp+ObjectNameLen+1]
    push 1
    ; - 1 = ObjectNameInformation
    push [ebp+currentHandle]
    call ds:NTQueryObject; <- Get current object name from handle
    mov [ebp+status], eax
    mov eax, [ebp+status]
    shr eax, 16h
    cmp eax, 3
    jz short loc_409687
```

Figure 56: Zeus Communication Step 2a
If it does, try to open a pipe with the name

```
\|\pipe\<handle_name>
```

And set the Named pipe handle state to “PIPE_READMODE_MESSAGE”. (Figure 58)
d. Now write the command (received as parameter) to the Pipe opened before. (Figure 59)

```
loc_409865:
    mov  eax, [ebp+8]  ; First Parameter = namedPipeCommand
    push  0             ; lpOverlapped
    lea  eax, [ebp+numberOfBytesReadWritten]
    push  eax
    push  4             ; nNumberofBytesToWrite
    lea  eax, [ebp+lpPipeCharStr]   ; Parameter Size (Fixed 0)
    push  eax
    push  0             ; lpBuffer
    call  dwsWritefile
    test  eax, eax
    jnz  short loc_409865
```

Figure 59: Zeus Communication Steps 2d-2e

e. Write Parameter Length to the Pipe (0 in our case). (Figure 59)
f. If Parameters are present, then write the parameters (There are none in our case) (Figure 60)

```
loc_409C07:  STEP 2f
  and [ebp+lpVideCharStr], 0
push 0 ; lpOverlapped
lea eax, [ebp+numberOfBytesReadOrWritten]
push eax ; lpNumberOfBytesWritten
push 0 ; nNumberOfBytesToWrite
lea eax, [ebp+lpVideCharStr]; <- Parameters (None)
push eax ; lpBuffer
push [ebp+hObject] ; hFile
call dwWriteFile

test eax, eax
jnz short loc_409C53
```

Figure 60: Zeus Communication Step 2f - 2g

```
loc_409C59:  STEP 2g
and [ebp+lpVideCharStr], 0
push 0 ; lpOverlapped
lea eax, [ebp+numberOfBytesReadOrWritten]
push eax ; lpNumberOfBytesRead
push 4 ; nNumberOfBytesToRead
lea eax, [ebp+lpVideCharStr]
push eax ; lpBuffer
push [ebp+hObject] ; hFile
call dwReadFile ; <- Read Response

test eax, eax
jnz short loc_409C45
```

g. Read the Response. (Figure 60)

h. Read the length of data and allocate buffer of that length. (Figure 61)

i. Read the data into the buffer and copy it to the fileName argument provided as input to this function. (Figure 61)
In addition, if the caller provided a pointer for the Mutex-Name argument, then set the handle-name as the name of the Mutex.

Note that as mentioned in the section, the Mutex name and pipe name are the same. (Figure 62)

Figure 62: Zeus Communication Step 2j

Close the Pipe and Object handles.

Note that the above loop ensures that the command is sent to all ZeuS pipes. The Figure 63 and Figure 64 show the ZeuS side of things i.e. how ZeuS handles these commands. As shown in Figure 63, upon receiving the command 0x0B, it returns back
the path of the Zbot binary, in this case – “C:\WINDOWS\system32\ntos.exe”. Figure 64 shows the code setting an event, which is used to wake all threads and terminate them. It is likely that Zbot uses this event during bot binary updates.

It should be noted here that SpyEye does not remove the registry keys created by Zbot. In addition, it also does not remove the other files such as the configuration files belonging to ZeuS from the host wither. However, having removed the executable (Figure 54) and terminated the threads (Figure 64), the ZBot infection has clearly been eliminated from the host. In this round of battle, SpyEye is the clear victor.

**Conclusion**

In this paper, we have discussed the details obtained from reverse engineering of samples belonging to the SpyEye and ZeuS botnets. Our research has shown that the two bots uses similar and comparable infection strategies. For example both have obfuscated
code to deter static analysis. Both also make extensive use of process injection techniques and set up auto-run registry keys to ensure that they are executed when the system restarts. Both have techniques to bypass, evade or disable host-based firewalls. They also use similar command and control structure, where the controlling server provides a configuration file, which is encrypted. This configuration file generally contains details of sites to be monitored, form grabbing details, in addition to URLs belonging to C&C sites.

We also studied the interaction between the two bots. While it is generally known that SpyEye has capabilities to remove ZeuS, the exact technique used was not publicly known at the time of this research. In this paper we have given the details of how SpyEye exploits the named pipe communication mechanism that ZeuS uses to communicate between its components in order to command ZeuS to terminate all its threads and then removes its binary from the infected host. In effect, it hijacks the IPC mechanism of ZeuS and uses it to clean up Zbot from the machine. This technique is likely to work for all ZeuS samples that use named pipes with names containing the strings “__SYSTEM__” and/or “_AVIRA_”.

As a final comment, we would like to assure the readers that no bots were harmed during the course of this research!

Acknowledgements

I would like to thank my friends and co-workers – Nima Dezhkam, Bin Lin and Steven Zhou, members of the Malware Research Team at Telus Security Labs for their help and support during this research. I am also thankful to Thayaparen Thankarajah, Program Director, Telus Security Labs for his help and guidance.

In addition, I am indebted to my research advisor, Pedro Bueno, who not only helped in selecting this research topic, but was also extremely encouraging with his generous compliments and patient reviews.

References


Harshit Nayyar, harshit.nayyar@telus.com

Harshit Nayyar, harshit.nayyar@telus.com
http://www.python.org/

Russinovich, M. ((1998)). *Process Explorer [Software].* Retrieved from


almighty.html


Harshit Nayyar, harshit.nayyar@telus.com
## Appendix

### VirusTotal Results – ZeuS/ZBot sample:

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Version</th>
<th>Last Update</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-squared</td>
<td>4.5.0.50</td>
<td>2010.02.23</td>
<td>PWS.Win32!IK</td>
</tr>
<tr>
<td>AhnLab-V3</td>
<td>5.0.0.2</td>
<td>2010.02.23</td>
<td>Win-Trojan/Zbot.71168.H</td>
</tr>
<tr>
<td>AntiVir</td>
<td>8.2.1.172</td>
<td>2010.02.23</td>
<td>TR/PSW.Zbot.PK.2</td>
</tr>
<tr>
<td>Antiy-AVL</td>
<td>2.0.3.7</td>
<td>2010.02.23</td>
<td>Trojan/Win32.Zbot.gen</td>
</tr>
<tr>
<td>Avast</td>
<td>4.8.1351.0</td>
<td>2010.02.23</td>
<td>Win32:Malware-gen</td>
</tr>
<tr>
<td>AVG</td>
<td>9.0.0.730</td>
<td>2010.02.23</td>
<td>SHeur2.CJLQ</td>
</tr>
<tr>
<td>BitDefender</td>
<td>7.2</td>
<td>2010.02.23</td>
<td>Trojan.Spy.ZBot.EMI</td>
</tr>
<tr>
<td>CAT-QuickHeal</td>
<td>10</td>
<td>2010.02.23</td>
<td>TrojanSpy.Zbot.aefh</td>
</tr>
<tr>
<td>Comodo</td>
<td>4035</td>
<td>2010.02.23</td>
<td>TrojanProxy.2842</td>
</tr>
<tr>
<td>DrWeb</td>
<td>5.0.1.12222</td>
<td>2010.02.23</td>
<td>Win32.PWSZbot.Pk</td>
</tr>
<tr>
<td>eSafe</td>
<td>7.0.17.0</td>
<td>2010.02.23</td>
<td>Win32/FakeAlert.B!generic</td>
</tr>
<tr>
<td>eTrust-Vet</td>
<td>35.2.7323</td>
<td>2010.02.23</td>
<td>Trojan.Spy.ZBot.EMI</td>
</tr>
<tr>
<td>F-Prot</td>
<td>4.5.1.85</td>
<td>2010.02.22</td>
<td>-</td>
</tr>
<tr>
<td>F-Secure</td>
<td>9.0.15370.0</td>
<td>2010.02.23</td>
<td>Trojan.Spy.ZBot.EMI</td>
</tr>
<tr>
<td>Fortinet</td>
<td>4.0.14.0</td>
<td>2010.02.21</td>
<td>W32/PackAgent.A38F!tr</td>
</tr>
<tr>
<td>GData</td>
<td>19</td>
<td>2010.02.23</td>
<td>Trojan.Spy.ZBot.EMI</td>
</tr>
<tr>
<td>Ikarus</td>
<td>T3.1.1.80.0</td>
<td>2010.02.23</td>
<td>PWS.Win32</td>
</tr>
<tr>
<td>K7AntiVirus</td>
<td>7.10.980</td>
<td>2010.02.22</td>
<td>Trojan.Win32.Malware.1</td>
</tr>
<tr>
<td>Kaspersky</td>
<td>7.0.0.125</td>
<td>2010.02.23</td>
<td>Trojan-Spy.Win32.Zbot.aefh</td>
</tr>
<tr>
<td>McAfee</td>
<td>5900</td>
<td>2010.02.22</td>
<td>Spy-Agent.bw</td>
</tr>
<tr>
<td>McAfee+Artemis</td>
<td>5900</td>
<td>2010.02.22</td>
<td>Spy-Agent.bw</td>
</tr>
<tr>
<td>McAfee-GW-Edition</td>
<td>6.8.5</td>
<td>2010.02.23</td>
<td>Trojan.PSW.Zbot.PK.2</td>
</tr>
<tr>
<td>Microsoft</td>
<td>1.5406</td>
<td>2010.02.23</td>
<td>PWS:Win32/Zbot.PK</td>
</tr>
<tr>
<td>NOD32</td>
<td>4890</td>
<td>2010.02.23</td>
<td>Win32/Spy.Zbot.JF</td>
</tr>
<tr>
<td>nProtect</td>
<td>2009.1.8.0</td>
<td>2010.02.23</td>
<td>Trojan/W32.Agent.71168.DJ</td>
</tr>
<tr>
<td>Panda</td>
<td>10.0.2.2</td>
<td>2010.02.22</td>
<td>Trj/Sinowal.WWJ</td>
</tr>
<tr>
<td>PCTools</td>
<td>7.0.3.5</td>
<td>2010.02.23</td>
<td>Trojan.Zbot</td>
</tr>
<tr>
<td>Prevx</td>
<td>3</td>
<td>2010.02.23</td>
<td>Medium Risk Malware</td>
</tr>
<tr>
<td>Rising</td>
<td>22.34.01.03</td>
<td>2010.02.11</td>
<td>Trojan.Win32.Generic.51F9162E</td>
</tr>
<tr>
<td>Sophos</td>
<td>4.50.0</td>
<td>2010.02.23</td>
<td>Mal/FakeAV-BT</td>
</tr>
<tr>
<td>Sunbelt</td>
<td>5694</td>
<td>2010.02.23</td>
<td>Trojan-Spy.Win32.Zbot.gen</td>
</tr>
<tr>
<td>Symantec</td>
<td>2009.1.2.0.41</td>
<td>2010.02.23</td>
<td>Trojan.Zbot</td>
</tr>
<tr>
<td>TheHacker</td>
<td>6.5.1.6.206</td>
<td>2010.02.23</td>
<td>Trojan/Kryptik.cex</td>
</tr>
<tr>
<td>TrendMicro</td>
<td>9.120.0.1004</td>
<td>2010.02.23</td>
<td>TROJ_ZBOT.SVR</td>
</tr>
</tbody>
</table>

Harshit Nayyar, harshit.nayyar@telus.com
Zbot Network Communication:

The code injected into svchost.exe is responsible for network communication including downloading the configuration file. The configuration file is encrypted and may have some compressed strings. We are giving here, some details about the encryption and compression schemes used. Some of this information is mentioned in the blog entitled, “Time to Revisit Zeus Almighty” (Shevchenko, 2009). The following python code decrypts the configuration file. Note that this script works on the encryption scheme of ZBot versions where shared secrets keys are not used. Also please note that this code is provided with no support or guarantee of correctness.

```python
def getBit(pos, recordDataEncoded, fourBytes, count):
    #Get the bit at position count. If count == 0, reinitialize count and move to #next decompression.
    if count == 0:
        count = 31
        fourBytes = struct.unpack('<L', recordDataEncoded[pos:pos+4])[0]
        print 'Read Four Bytes: 0x%.8X at Pos: %d' % (fourBytes, pos)
        pos += 4
    else:
        count -= 1
    bit = ((fourBytes >> count) & 1)
    return (bit, pos, fourBytes, count)
```

The following python code gives the decompression algorithm identified as nrv2b.
def secondDecode(recordDataEncoded):
    recordDataDecoded = ''
    sPos = 0
dPos = 0
lastMOff = 1
shift = 0
fourBytes = 0

# Main Loop
while True:
    if sPos >= len(recordDataEncoded):
        return recordDataDecoded
    print 'first shift is: 0x%x' % (shift)
    (gb, sPos, fourBytes, shift) = getBit(sPos, recordDataEncoded, fourBytes, shift)
    while (gb != 0):
        recordDataDecoded += recordDataEncoded[sPos]
        sPos += 1
        if sPos > len(recordDataEncoded):
            'Record Data Len Exceeded 1'
            return recordDataDecoded
dPos += 1
    (gb, sPos, fourBytes, shift) = getBit(sPos, recordDataEncoded, fourBytes, shift)

    # mOff calculation
    if sPos >= len(recordDataEncoded):
        return recordDataDecoded
    (gb, sPos, fourBytes, shift) = getBit(sPos, recordDataEncoded, fourBytes, shift)
mOff = 2 + gb
    if sPos >= len(recordDataEncoded):
        return recordDataDecoded
    (gb, sPos, fourBytes, shift) = getBit(sPos, recordDataEncoded, fourBytes, shift)
    while (gb == 0):
        if sPos >= len(recordDataEncoded):
            'Record Data Len Exceeded 2'
            return recordDataDecoded
        (gb, sPos, fourBytes, shift) = getBit(sPos, recordDataEncoded, fourBytes, shift)
mOff = 2 * mOff + gb
    if int(mOff) == -1:
        break;
    else:
        mOff += 1
    lastMOff = mOff

Harshit Nayyar, harshit.nayyar@telus.com
Using the above algorithm for decryption, we found that the configuration file being downloaded by this Zbot sample had over 50 different URLs mostly for banking institutions.

Harshit Nayyar, harshit.nayyar@telus.com
### SpyEye Sample

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Version</th>
<th>Last Update</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-squared</td>
<td>4.5.0.50</td>
<td>2010.03.25</td>
<td>Trojan-Spy.Win32.SpyEyes.d!A2</td>
</tr>
<tr>
<td>AhnLab-V3</td>
<td>5.0.0.2</td>
<td>2010.03.24</td>
<td>Win-Trojan/Spyeye.126464</td>
</tr>
<tr>
<td>AntiVir</td>
<td>8.2.1.196</td>
<td>2010.03.24</td>
<td>TR/Agent.qqs</td>
</tr>
<tr>
<td>Antiy-AVL</td>
<td>2.0.3.7</td>
<td>2010.03.24</td>
<td>Trojan/Win32.SpyEyes.gen</td>
</tr>
<tr>
<td>Avast</td>
<td>4.8.1351.0</td>
<td>2010.03.24</td>
<td>Win32:Spyware-gen</td>
</tr>
<tr>
<td>Avast5</td>
<td>5.0.332.0</td>
<td>2010.03.24</td>
<td>Win32:Spyware-gen</td>
</tr>
<tr>
<td>AVG</td>
<td>9.0.0.787</td>
<td>2010.03.25</td>
<td>Generic16.BFGI</td>
</tr>
<tr>
<td>BitDefender</td>
<td>7.2</td>
<td>2010.03.25</td>
<td>Trojan.Generic.3159827</td>
</tr>
<tr>
<td>CAT-QuickHeal</td>
<td>10</td>
<td>2010.03.24</td>
<td>TrojanSpy.SpyEyes.d</td>
</tr>
<tr>
<td>Comodo</td>
<td>4373</td>
<td>2010.03.24</td>
<td>TrojWare.Win32.TrojanSpy.Agent.–AFZ</td>
</tr>
<tr>
<td>DrWeb</td>
<td>5.0.1.12222</td>
<td>2010.03.25</td>
<td>Trojan.MulDrop1.815</td>
</tr>
<tr>
<td>eSafe</td>
<td>7.0.17.0</td>
<td>2010.03.24</td>
<td>Win32.TRAgent.Qqs</td>
</tr>
<tr>
<td>eTrust-Vet</td>
<td>35.2.7387</td>
<td>2010.03.25</td>
<td>Win32/Spyeye.A</td>
</tr>
<tr>
<td>F-Secure</td>
<td>9.0.15370.0</td>
<td>2010.03.25</td>
<td>Trojan-Spy.Win32/Spyeye.A</td>
</tr>
<tr>
<td>Fortinet</td>
<td>4.0.14.0</td>
<td>2010.03.24</td>
<td>W32/SpyEyes.Dltr</td>
</tr>
<tr>
<td>GData</td>
<td>19</td>
<td>2010.03.25</td>
<td>Trojan.Generic.3159827</td>
</tr>
<tr>
<td>Ikarus</td>
<td>T3.1.1.80.0</td>
<td>2010.03.25</td>
<td>Trojan.Win32.Spyeye</td>
</tr>
<tr>
<td>Jiangmin</td>
<td>13.0.900</td>
<td>2010.03.24</td>
<td>Trojan/Pincav.dfy</td>
</tr>
<tr>
<td>K7AntiVirus</td>
<td>7.10.1004</td>
<td>2010.03.22</td>
<td>Trojan-Spy.Win32.SpyEyes.d</td>
</tr>
<tr>
<td>Kaspersky</td>
<td>7.0.0.125</td>
<td>2010.03.25</td>
<td>Trojan-Spy.Win32.SpyEyes.d</td>
</tr>
<tr>
<td>McAfee</td>
<td>5930</td>
<td>2010.03.24</td>
<td>BackDoor-Spyeye</td>
</tr>
<tr>
<td>McAfee+Artemis</td>
<td>5930</td>
<td>2010.03.24</td>
<td>BackDoor-Spyeye</td>
</tr>
<tr>
<td>McAfee-GW-Edition</td>
<td>6.8.5</td>
<td>2010.03.24</td>
<td>Trojan.Agent.qqs</td>
</tr>
<tr>
<td>Microsoft</td>
<td>1.5605</td>
<td>2010.03.24</td>
<td>Trojan:Win32/Spyeye</td>
</tr>
<tr>
<td>NOD32</td>
<td>4972</td>
<td>2010.03.24</td>
<td>Win32/SpyEye.B</td>
</tr>
<tr>
<td>Norman</td>
<td>6.04.10</td>
<td>2010.03.24</td>
<td>Spyeye.A</td>
</tr>
<tr>
<td>nProtect</td>
<td>2009.1.8.0</td>
<td>2010.03.24</td>
<td>Trojan-Spy/W32.SpyEyes.126464</td>
</tr>
<tr>
<td>Panda</td>
<td>10.0.2.2</td>
<td>2010.03.24</td>
<td>Suspicious file</td>
</tr>
<tr>
<td>PCTools</td>
<td>7.0.3.5</td>
<td>2010.03.24</td>
<td>Trojan.Spyeye</td>
</tr>
<tr>
<td>Prevx</td>
<td>3</td>
<td>2010.03.25</td>
<td>High Risk Spyware</td>
</tr>
<tr>
<td>Rising</td>
<td>22.40.03.01</td>
<td>2010.03.25</td>
<td>Trojan.Win32.Generic.51F9905F</td>
</tr>
<tr>
<td>Sophos</td>
<td>4.51.0</td>
<td>2010.03.24</td>
<td>Mal/Spyeye-A</td>
</tr>
<tr>
<td>Sunbelt</td>
<td>6072</td>
<td>2010.03.25</td>
<td>Trojan.Win32.Generic!BT</td>
</tr>
<tr>
<td>Symantec</td>
<td>20091.2.0.41</td>
<td>2010.03.25</td>
<td>Trojan.Spyeye</td>
</tr>
<tr>
<td>TheHacker</td>
<td>6.5.2.0.242</td>
<td>2010.03.24</td>
<td>Trojan/SpyEyes.d</td>
</tr>
<tr>
<td>TrendMicro</td>
<td>9.120.0.1004</td>
<td>2010.03.24</td>
<td>TSPY_EYEBOT.SMA</td>
</tr>
<tr>
<td>VBA32</td>
<td>3.12.12.2</td>
<td>2010.03.24</td>
<td>Trojan-Spy.Win32.SpyEyes.d</td>
</tr>
<tr>
<td>ViRobot</td>
<td>2</td>
<td>2010.03.24</td>
<td>Trojan.Win32.Spyeye.126464</td>
</tr>
<tr>
<td>VirusBuster</td>
<td>5.0.27.0</td>
<td>2010.03.24</td>
<td>TrojanSpy.SpyEyes.BD</td>
</tr>
</tbody>
</table>

Harshit Nayyar, harshit.nayyar@telus.com
Script to Map SpyEye Hash Value to Functions:

The following script creates a hash table whose keys are the 4 Byte hash and the value is the name of the DLL export. It uses PEFile (Carrera, 2005) python package to obtain the names of exports. Note that it uses the nameToHash function described in Section 31.

```python
#!/usr/bin/python
import os,sys
sys.path.append('./pefile')
import pefile
import NameToHash

#CommonDLLs contains the following DLLs:
#advapi32.dll  kernel32.dll  ntdll.dll  oleaut32.dll
#shlwapi.dll  user32.dll   ws2_32.dll
#imagehlp.dll  msvcrt.dll    ole32.dll
#shell32.dll   urlmon.dll   wininet.dll

DirPath = './CommonDLLs'

def listExportsInFile(inFileName):
    pe = pefile.PE(inFileName)
    expList = []
    try:
        exportsDir = pe.DIRECTORY_ENTRY_EXPORT.symbols
    except:
        exportsDir = None
    if exportsDir:
        for exp in exportsDir:
            expList += [exp.name]
    return expList

def hashAllExportsInCommonDLLs():
    dllNames = os.listdir(DirPath)
    funcDict = {}
    for dll in dllNames:
        dllFile = os.path.join(DirPath,dll)
        exportList = listExportsInFile(dllFile)
        for fName in exportList:
            if fName:
                hash = NameToHash.nameToHash(fName)
                funcDict["%.8X"%hash] = (dll,fName)
    return funcDict

if __name__ == '__main__':
    listAllExportsInCommonDLLs()
```

SpyEye Function calling Technique

1. Select Library Name based on argument given (Figure 65).
10. Find base of Ntdll.dll. (Figure 66)

11. Find address of LdrLoadDll within Ntdll.dll. (Figure 66)

12. Load the library whose name was found in Step 1 using LdrLoadDll. (Figure 66)

13. Find the function name from the hash value as discussed before. (Figure 67)
14. Find the offset of the function inside the loaded DLL.

15. Pass the DLL name and the function offset to a function that does the following (Figure 68):
   a. Open the DLL file using ntdll!NtOpenFile (Figure 69)
b. Create a new section using ntdll!NtCreateSection(Figure 70)

c. Map the file into the newly created section using ntdll!MapViewOfSection. Hence the DLL is mapped into a new section in memory. (Figure 70)

d. Find the function address inside the new section from the offset passed. Return this address. (Figure 71)
16. Now function at that address is called using the appropriate arguments. (Figure 72)

17. After the function call, the DLL is unmapped. (Figure 72)

SpyEye Dropped Binary Sample:

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Version</th>
<th>Last Update</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-squared</td>
<td>4.5.0.50</td>
<td>2010.04.1</td>
<td>Trojan-Spy.Win32.SpyEyes!IK</td>
</tr>
</tbody>
</table>

Harshit Nayyar, harshit.nayyar@telus.com
<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Version</th>
<th>Date</th>
<th>Malware</th>
</tr>
</thead>
<tbody>
<tr>
<td>AhnLab-V3</td>
<td>5.0.0.2</td>
<td>2010.04.1</td>
<td>Win-Trojan/Agent.104448.DK</td>
</tr>
<tr>
<td>AntiVir</td>
<td>7.10.6.76</td>
<td>2010.04.1</td>
<td>TR/Agent.qqs</td>
</tr>
<tr>
<td>Antiy-AVL</td>
<td>2.0.3.7</td>
<td>2010.04.1</td>
<td>Trojan/Win32.SpyEyes.gen</td>
</tr>
<tr>
<td>Authentium</td>
<td>5.2.0.5</td>
<td>2010.04.1</td>
<td>W32/SpyEyes.A.gen!Eldorado</td>
</tr>
<tr>
<td>Avast</td>
<td>4.8.1351.0</td>
<td>2010.04.1</td>
<td>Win32:Spyware-gen</td>
</tr>
<tr>
<td>Avast5</td>
<td>5.0.332.0</td>
<td>2010.04.1</td>
<td>Win32:Spyware-gen</td>
</tr>
<tr>
<td>AVG</td>
<td>9.0.0.787</td>
<td>2010.04.1</td>
<td>Generic16.BHEP</td>
</tr>
<tr>
<td>BitDefender</td>
<td>7.2</td>
<td>2010.04.1</td>
<td>Trojan.Generic.3102763</td>
</tr>
<tr>
<td>CAT-QuickHeal</td>
<td>10</td>
<td>2010.04.1</td>
<td>-</td>
</tr>
<tr>
<td>ClamAV</td>
<td>0.96.0.3-git</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Comodo</td>
<td>4596</td>
<td>2010.04.1</td>
<td>TrojWare.Win32.TrojanSpy.Spyeye.~A</td>
</tr>
<tr>
<td>DrWeb</td>
<td>5.0.2.03300</td>
<td></td>
<td>Trojan.Dialer.11</td>
</tr>
<tr>
<td>eSafe</td>
<td>7.0.17.0</td>
<td>2010.04.1</td>
<td>Win32.TRAgent.Qqs</td>
</tr>
<tr>
<td>eTrust-Vet</td>
<td>35.2.7425</td>
<td>2010.04.1</td>
<td>Win32/SpyEye.H</td>
</tr>
<tr>
<td>F-Prot</td>
<td>4.5.1.85</td>
<td>2010.04.1</td>
<td>W32/SpyEyes.A.gen!Eldorado</td>
</tr>
<tr>
<td>F-Secure</td>
<td>9.0.15370.0</td>
<td></td>
<td>Trojan-Spy:W32/Spyeye.A</td>
</tr>
<tr>
<td>Fortinet</td>
<td>4.0.14.0</td>
<td>2010.04.1</td>
<td>W32/SpyEyes.B!tr</td>
</tr>
<tr>
<td>GData</td>
<td>19</td>
<td>2010.04.1</td>
<td>Trojan.Generic.3102763</td>
</tr>
<tr>
<td>Ikarus</td>
<td>T3.1.1.80.0</td>
<td></td>
<td>Trojan-Spy.Win32.SpyEyes</td>
</tr>
<tr>
<td>Jiangmin</td>
<td>13.0.900</td>
<td>2010.04.1</td>
<td>-</td>
</tr>
<tr>
<td>Kaspersky</td>
<td>7.0.0.125</td>
<td>2010.04.1</td>
<td>Trojan.Win32.SpyEyes.b</td>
</tr>
<tr>
<td>McAfee</td>
<td>5.400.0.1158</td>
<td></td>
<td>BackDoor-Spyeye</td>
</tr>
<tr>
<td>McAfee-GW-Edition</td>
<td>6.8.5</td>
<td>2010.04.1</td>
<td>Trojan.Agent.qqs</td>
</tr>
<tr>
<td>Microsoft</td>
<td>1.5605</td>
<td>2010.04.1</td>
<td>Trojan:Win32/Spyeye.B</td>
</tr>
<tr>
<td>Norman</td>
<td>6.04.11</td>
<td>2010.04.1</td>
<td>-</td>
</tr>
</tbody>
</table>
from http.server import HTTPServer, SimpleHTTPRequestHandler
import cgi
import base64
import time
import binascii
from datetime import datetime

PORT = 80

def printMessage(msg):
    print( "[%s] %s"%(datetime.now(), msg) )

class MRTRequestHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        if self.path.find('x98x10.bin') > 0:
            printMessage('Got ZeuS ConfigFile request: '+self.path)
            resp = 'HTTP/1.1 200 OK
\r\nDate: Tue, 26 Jan 2010 23:30:22 GMT
\r\nContent-Type: text/plain;
\r\ncharset=UTF-8\r\nConnection: close
\r\nLast-Modified: Tue, 26 Jan 2010 23:30:22 GMT
\r\nAccept-Ranges: bytes\r\nContent-Length: '+str(len(f.read()))
        resp += '
\r\n'+toSend = binascii.a2b_qp(resp)+f.read()
        printMessage('Sending Configuration file: '+self.path)
        self.wfile.write(toSend)
    elif self.path.find('bt_version_checker.php') > 0:
        printMessage('Got SpyEye version_checker Request: '+self.path)
        resp = 'HTTP/1.1 200 OK
\r\nVary: Accept-Encoding
\r\nContent-Type: text/html\r\nContent-Length: %s
\r\n'%(len(dataToSend))
        resp += dataToSend
        toSend = binascii.a2b_qp(resp)
        printMessage('Sending a 200OK Response: ') self.wfile.write(toSend)
    else:
        SimpleHTTPRequestHandler.do_GET(self)
        return

def do_POST(self):
    if self.path.find('websitechk.php') > 0:
        printMessage('Got SpyEye Website Check')
    elif self.path.find('s.php') > 0:
        printMessage('Got ZBot POST Request')
    else:
        printMessage('Got Unkonwn Request: %s'%self.path)
        printMessage('Sending 200OK Response')
        fullResp = 'HTTP/1.1 200 OK\r\nDate: Mon, 15 Feb 2010 18:56:24 GMT
\r\nServer: Apache/2.2.9
\r\nX-Powered-By: PHP/5.2.12\r\nConnection: close\r\nContent-Length: 0\r\n\r\n'+toSend = binascii.a2b_qp(fullResp)
        self.wfile.write(toSend)
        return

def main():
    global PORT
    server_address = ('', PORT)
    httpd = HTTPServer(server_address, MRTRequestHandler)
    httpd.serve_forever()
## Upcoming SANS Training

Click here to view a list of all SANS Courses

<table>
<thead>
<tr>
<th>Training Details</th>
<th>City, Country</th>
<th>Dates</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS Amsterdam August 2020 Part 1</td>
<td>Amsterdam, NL</td>
<td>Aug 03, 2020 - Aug 08, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS FOR508 Canberra August 2020</td>
<td>Canberra, AU</td>
<td>Aug 17, 2020 - Aug 22, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Amsterdam August 2020 Part 2</td>
<td>Amsterdam, NL</td>
<td>Aug 17, 2020 - Aug 22, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Virginia Beach 2020</td>
<td>Virginia Beach, VAUS</td>
<td>Aug 31, 2020 - Sep 05, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Philippines 2020</td>
<td>Manila, PH</td>
<td>Sep 07, 2020 - Sep 19, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS London September 2020</td>
<td>London, GB</td>
<td>Sep 07, 2020 - Sep 12, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Baltimore Fall 2020</td>
<td>Baltimore, MDUS</td>
<td>Sep 08, 2020 - Sep 13, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>Threat Hunting &amp; Incident Response Summit &amp; Training 2020</td>
<td>New Orleans, LAUS</td>
<td>Sep 10, 2020 - Sep 17, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>OnlineUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>