MARCH 2014

virus BULLETIN

Covering the global threat landscape

CONTENTS

IN THIS ISSUE

FOREVER TARRY

Last month, Peter Ferrie described a Windows virus that turns Java class files into droppers for the virus, and concluded that it would be a simple matter to reverse that: for a virus writer to create a Java class file that turns Windows files into droppers for the virus. This is exactly what {W32/Java}/Tarry does.
page 4

RESURFACING INFECTOR

Expiro is a file infector that resurfaces from time to time, demonstrating more skills on each new appearance – infecting a service that gives a unique vantage point on traditional malicious activities; running the malware at computer restart without creating a start-up registry; using different mutexes for different types of infected process; escalating privileges; and executing infected files without calling the CreateProcess or WinExec APIs. Raul Alvarez takes a closer look.
page 17

COME PREPARED

The author of Simbot doesn't take anything for granted: all the necessary components for the malware's execution are bundled and dropped onto the system, including the relevant vulnerable application for exploitation and regular *Windows* system binaries.
page 21

ISSN 1749-7027

'There is a shift occurring ... around incident response. It's becoming clear that no organization is completely safe.'

Tim Armstrong, Co3 Systems

MAKING THE CASE FOR INCIDENT RESPONSE

Currently, there is a lot of talk of preventative security technologies being all but dead. I disagree, but their use might have changed. It would be a mistake to ignore the recommended best practice of installing anti-virus, but it would be an even bigger mistake to stop there. The threat detection market is alive and well with a plethora of advanced technologies that do incredible things, from on-demand virtual sandboxes to advanced APT detection.

The problem is this: in some cases the bad guys will still get in, and the way in which you react once your defences have been breached can make the difference between a security event and a security disaster. It could mean the end of your job, or even the end of your company. A vast amount of time and money is dedicated to trying to keep the bad guys out, but very little is spent on planning for what to do when that defence fails.

Every day, I talk to organizations that have great intentions, but little to no preparation. Making the case for incident response to management can be trying at best. Unless your company has seen the result of a serious incident, both in terms of clean-up costs and brand damage, it can be an uphill battle to convince budget holders of the value of incident response tools and techniques. While the continuing catalogue of high-profile breaches in the news can only help your case, this is a problem that is not going to go away any time soon.

You can understand why, after so much time, effort and money has been spent on preventative tools, security professionals are hesitant to bring up the need for more tools or resources to deal with the situation when someone defeats their defences. While preventative tools are necessary, make no mistake, *someone* will get around them. Given enough time and resources on the part of the attacker, no system is 100% secure. We can see this is especially true in the case of state-sponsored malware – these attackers have almost endless resources.

What is not often discussed is how valuable defensive tools can be to the incident response process. Anti-virus, IDS, SIEMs and other security tools are essential in recreating an incident timeline. They provide us with information about the attack vector, pathway through the network, and indicators of compromise. In fact, they allow us to make our defences stronger, once we know where to look.

There is a shift occurring in the security space around incident response. It's becoming clear that no organization is completely safe. Whether you're a retailer, a manufacturer, a hospital or insurance firm, you will need a plan. You will need a system of record. You will need a repeatable process, and the last thing you want is to be creating that process during the heat of an incident. In fact, you'll want to tie that incident response process into every tool that helps.

A large part of the incident response process is made up of research, and having the output of quality tools available will shorten your response time. Look at any breach report and you'll see that the response time is currently measured in months. As an industry, we need to bring that under control – it is the unfortunate truth that the bad guys currently have the upper hand. We are not winning at the gateway, we are not winning on the network, and we are not winning at the endpoint. We *must* win at incident response. Collecting data during incident response, sharing indicators of compromise, and making it as hard as possible for attackers is our best chance.

I believe that we need to accept that we will all deal with a breach at some point. But if we can act collectively to make attackers less effective, and get rid of the shame of disclosure, we can turn the tide of security to our favour.

NEWS

40% OF CRYPTOLOCKER VICTIMS PAY UP

Around 40% of people whose machines are infected by the CryptoLocker ransomware end up agreeing to pay a ransom of around £300 to recover their files, according to researchers from the University of Kent.

CryptoLocker has become known as the unfortunate crypto success story of 2013. While stories about broken cryptography implementations made the headlines throughout the second half of the year, no one has yet been able to find a serious weakness in this piece of ransomware. Victims who find their files encrypted by CryptoLocker and who do not have a back-up of their files are forced either to pay the ransom, or to consider the files lost forever.

Researchers at the University of Kent's Research Centre for Cyber Security surveyed 1,500 adults in the UK, asking them eight cybercrime and security-related questions. They discovered that ransomware represents one in every 30 malware cases – which is higher than previous estimates.

The researchers questioned 48 people who had been affected by CryptoLocker – of whom, 17 said they paid the ransom and 31 said they did not.

Although CryptoLocker has proved a tough case to crack and has caused many a headache for those unfortunate enough to find themselves infected with it, the creators of other pieces of ransomware are, thankfully, not quite as adept. Last month, French researchers Fabien Perigaud and Cedric Pernet uncovered a serious vulnerability in the Bitcrypt ransomware that allowed them to crack the keys used by the malware to encrypt the victim's files – as a result, they were able to restore the encrypted files. The researchers have made a Python script available so that others can also crack Bitcrypt's keys and restore their files.

SECURING THE INTERNET OF THINGS

Cisco is offering prizes of up to US$75,000 for solutions dealing with the issue of securing the Internet of Things (IoT).

The future will likely see our fridges, cars, TVs and even toothbrushes connected to the Internet – but while the IoT offers countless possibilities for increasing efficiency, streamlining day-to-day tasks, gathering data and so on, the biggest perceived disadvantage is security.

Cisco's 'Internet of Things Grand Security Challenge' offers prizes ranging from US$50,000 to US$75,000 for proposals that deal with securing the IoT.

The judges will assess submissions assessed for: feasibility, scalability, performance, and ease-of-use; applicability to multiple IoT verticals (manufacturing, mass transportation, healthcare, oil and gas, smart grid, etc.); technical maturity/viability of proposed approach; and the proposers' expertise and ability to feasibly create a successful outcome.

The winners will be announced in the autumn.

INSURERS REFUSE TO COVER POORLY PROTECTED POWER FIRMS

According to the *BBC*, energy and utility companies are being turned down by insurance firms when requesting insurance cover for cyber attacks because their defences are perceived to be too weak.

Underwriters at specialist insurance market *Lloyd's of London* told the *BBC* that there has been a huge increase in demand for cover from energy firms over the last year or so – but that in the majority of cases, cover is being refused as assessors are finding that the firms' IT security defences, policies and procedures are inadequate.

It has long been common for insurance firms to offer businesses cover against data breaches – for example to aid in their recovery in the event of their networks being penetrated and customer data stolen. Recently, however, there has been a significant rise in the number of power companies applying for multi-million-pound policies to cover them against damages caused by a cyber attack.

Clearly, the perception of the insurance assessors is that, in the majority of cases, firms are placing too much emphasis on transferring the risk to insurers, rather than paying enough attention to risk mitigation. It is to be hoped that the companies that have been refused insurance cover will revisit their cybersecurity policies and procedures and focus on strengthening the protection they have in place – using insurance cover as a means to transfer the remaining risk, rather than as a substitute for a robust security programme.

BLACK MARKET HAUL

360 million stolen credentials and 1.25 billion email addresses were found on the black market last month by security firm *Hold Security*.

The firm believes that the 360 million credentials come from multiple breaches that have not yet been publicly disclosed (and may not even be known about by the victims) – with a single breach accounting for 105 million of the stolen credentials. Many of the passwords paired with usernames were in plaintext.

The stolen email addresses are from all the popular email providers such as *AOL*, *Gmail*, *Microsoft* and *Yahoo!*, but also include addresses from a large number of the Fortune 500 companies and several nonprofit organizations.

MALWARE ANALYSIS 1

A SHORT VISIT WITH A VIRUS

Peter Ferrie
Microsoft, USA

Last month, I described a *Windows* virus that turns Java class files into droppers for the virus [1]. I concluded that it would be a simple matter for a virus writer to reverse that – in other words, to have a Java class file that turns *Windows* files into droppers for the virus. That is exactly what we have in {W32/Java}/Tarry.

SECOND PLACE GOES TO...

The virus begins by pushing the host original entry point onto the stack. It then adds the host ImageBase value from the Process Environment Block, to construct the virtual address of the host entry point. This allows the virus to support applications that opt into Address Space Layout Randomization (ASLR), even though it does not support files that support ASLR.

The virus registers a Structured Exception Handler, then retrieves the base address of kernel32.dll. It does this by walking the InLoadOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The virus assumes that kernel32.dll is the second entry in the list. This is true for *Windows XP* and later, but it is not guaranteed under *Windows 2000* and earlier because, as the name suggests, the list shows the order of *loaded* modules. If kernel32.dll is not the first DLL that is loaded explicitly, then it won't be the second entry in that list (ntdll.dll is guaranteed to be the first entry in all cases).

IMPORT/EXPORT BUSINESS

The virus resolves the address of the only API function that it requires: LoadLibraryA. Despite this being only a single entry, the virus uses a hash instead of a name. It uses a reverse polynomial to calculate the hash. The virus does not check that the export exists, relying instead on the Structured Exception Handler to deal with any problems that occur. Of course, the required API should always be present in the kernel, so no errors should occur anyway.

The hash table is terminated explicitly using a single byte. The position of this byte corresponds to the next hash value in the list, and the search exits when a particular value is seen. This is intended to save three bytes of data, but actually introduces a risk. The assumption is that no hash will have that value in that position. While this is true in the case of this virus, it could result in unexpected behaviour if other APIs are added, for which the low byte happens to match the current sentinel value. The reason the hash technique is used to resolve a single API address is simply because this virus is derived from an existing code base that makes use of the technique for resolving multiple API addresses. It would be quicker simply to resolve GetProcAddress() by string name, and then use that API to resolve LoadLibraryA().

JAVA VIRTUAL MISMATCH

The virus loads the jvm.dll with no path. This assumes that the DLL can be found in one of the locations in which *Windows* searches, but the host can affect this list by using DLL redirection or carrying a manifest. The virus resolves a single API from this DLL, again by using the hash method. The API is JNI_CreateJavaVM(). The virus uses this API to create a new instance of the Java VM, but in order for the API to succeed, the '_ALT_JAVA_HOME_DIR' environment variable must be defined and must point to the Java installation directory. This is not normally the case. The usual method for invoking Java is by adding the Java installation directory to the path environment variable, and then either passing the directory on the command line or allowing the Java executable to define the environment variable dynamically.

The virus creates a byte array that is large enough to hold the combined virus body, then defines a class that contains the Java-specific virus body. It retrieves a pointer to the infection method within the Java-specific virus body, then runs the method. This technique allows the virus to execute Java code directly from memory, instead of dropping a class file and executing it from disk. After the method call returns, the virus deletes the objects in memory, then raises an exception using the 'int 3' technique. The 'int 3' technique appears only once in the virus code, but is an elegant way of reducing the code size. Since the virus has protected itself against errors by installing a Structured Exception Handler, the simulation of an error condition results in the execution of a common block of code to exit a routine. This avoids the need for separate handlers for successful and unsuccessful code completion.

The exception handler restores the registers, then transfers control to the host entry point.

CAFEBABE

The Java-specific virus body begins by creating a list of the objects in the current directory. The virus enumerates

the entries in the list, looking specifically for files. It uses no bounds checking during the enumeration, relying instead on a defined exception handler to receive control when the list is exhausted. The virus attempts to open any file it finds, in writable mode. It does not check whether the file is writable, nor does it remove the read-only attribute first.

The virus is interested in 32-bit Portable Executable files for the *Intel* x86 platform that have no relocation items, line numbers, symbols, or appended data, and that are not system or DLL files. The virus checks the COFF magic number to ensure that the file is 32-bit. It requires the file to be targeting the GUI subsystem and to have no flags set in the DllCharacteristics field. This reduces the pool of candidates to those which are primarily not No-eXecute compatible, and which do not support Address Space Layout Randomization. This effectively rules out all modern applications. The virus also requires that the file has no Load Configuration Table structure, which further rules out any file that uses SafeSEH, among other things.

The virus has some strange code in a couple of places, whereby a query is made for the current file pointer position, despite a seek() operation being issued immediately before it. In that case, the position would be known already. The virus also searches to the end of the file by adding the SizeOfRawData and PointerToRawData values from the last section, instead of simply using the file.length property.

[NO] TEST, [NO] FIX

The technique that the virus uses to infect a file is to write a push instruction to the end of the file, followed by what should be the original AddressOfEntryPoint value, and then the array that contains the combined virus body. However, there is a bug in the code that performs writes to the file. The bug is that the third byte of each four-byte write references the wrong variable. Instead of accessing the third byte of the four-byte value, it accesses the first byte of the SizeOfRawData value from the last section. Such a bug would have been obvious immediately had any attempt at recursive infection been made. It seems likely that the first generation of the virus code was executed, it infected a file, and the result was examined statically.

After writing the combined virus body, the size of the file is increased by 4KB. This value is hard-coded, and is entirely independent of the size of the virus code. The virus introduces the infection marker for files whose file alignment is less than 4KB. However, since the increase in size does not take into account the actual file alignment value, a file whose file alignment value exceeds 4KB will be reinfectable because it lacks the infection marker. The virus increases the virtual and physical size of the last section by exactly 4KB each. In this case, since the increase in size does not take into account the actual file alignment value, a file whose file alignment value exceeds 4KB will appear to be truncated and will fail to load. Finally, the virus marks the section characteristics as writable and executable.

BEST INTENTIONS

The virus intends to set the entry point to point to the original end of the last section, but the file-write bug applies here, resulting in a corruption of the value. This means that unless the host had an entry point whose third byte happened to match the value of the first byte of the size of the last section (most commonly, a value of zero), then the infected file will not execute any code.

The virus also intends to increase the size of the image by 4KB, but depending on the original value of the size, the file-writing bug is likely to corrupt the value to the point where the new image size is far too small to cover all of the sections, and thus the infected file will not even load. Specifically, if the size of the image (not the file) is any multiple of 61,440 bytes up to about 15MB, which is the case for calc.exe in *Windows XP* (126,976 bytes), then the new size of the image will be zero. Other values will simply be truncated, but with the same effect.

Once the file has been infected, the virus closes the handle, and then continues to search for more files.

CONCLUSION

There is nothing inherently difficult about creating a Java virus that infects *Windows* executable files. The fact that the Java-specific portion of this virus code was written directly in byte code rather than the high-level code is merely a detail. However, the fact that something as simple as that could yield such a significant bug should give pause to the virus writer. This is clearly not where he/she should be spending his/her time. There are many other arts which are far more forgiving of mistakes in implementation. Why not go and paint something instead?

REFERENCE

[1] Ferrie, P. Getting one's hands dirty. Virus Bulletin, February 2014, p.4. http://www.virusbtn.com/pdf/magazine/2014/201402.pdf.

MALWARE ANALYSIS 2

PROXYCB, A SPAM PROXY UNDER THE RADAR

Wei Wang & Kyle Yang
Fortinet, Canada

ProxyCB is a trojan that acts as a proxy server to send spam via the HTTP, HTTPS or SMTP protocol. It has been active in the wild since December 2011. ProxyCB usually has a three-level file structure: an installer EXE file, a loader DLL component and ProxyCB payload (DLL file). In this article, we'll take a detailed look at its installation process, how it bypasses UAC, and the final payload loading process, before dissecting its communication protocol and commands.

INSTALLATION PROCESS

Preparing

Upon executing, ProxyCB first generates a 12-letter string, which is derived from the volume serial number of the victim machine (see Figure 1) and will be used later both as a mutex name and as the filename of the dropped file.

Next, it checks the mutex (see Figure 1) to make sure the current machine is not infected, and checks the version of *Windows* and the user privilege level. Depending on the results, it will decide how to install the ProxyCB bot on the victim machine.

Bypassing UAC

User Account Control (UAC) has been part of the *Windows* operating system since *Windows Vista*. It aims to improve the security of the operating system by limiting applications to standard user privileges until an administrator authorizes an elevation in privileges. In this way, only applications that are trusted by the user can receive administrative privileges, and malware should be kept from compromising the operating system [1].

With the *Windows 7* UAC default settings, if you want to copy a file into the system directory, a prompt will be displayed even if you are an administrator (see Figure 2).

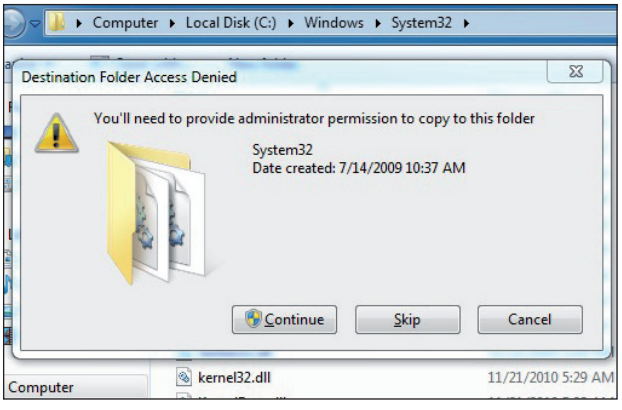

Figure 2: 'Access Denied' message.

In this case, the explorer.exe process runs at a medium integrity level that only has the Authenticated Users security access token [2]. This does not have the ability to write files to the system folder. If the bot wants to drop a file into the system folder or modify the registries, it needs to gain high integrity (Figure 3). This is the first and most important part of the installation process.

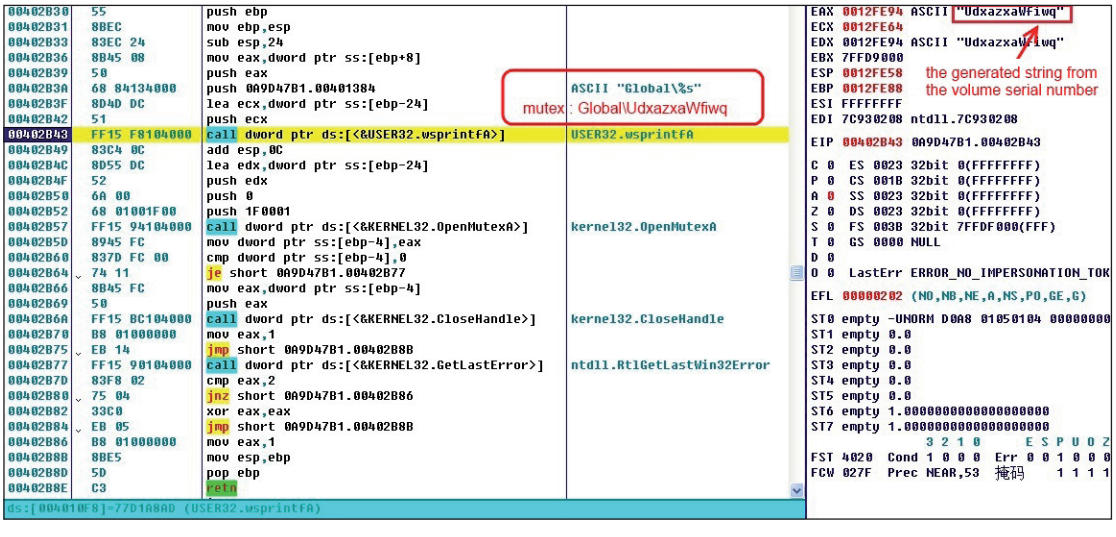

Figure 1: The generated string and checking the mutex.

SID in access token	Assigned integrity level
LocalSystem	System
LocalService	System
NetworkService	System
Administrators	High
Backup Operators	High
Network Configuration Operators	High
Cryptographic Operators	High
Authenticated Users	Medium
Everyone (World)	Low
Anonymous	Untrusted

Figure 3: Integrity levels linked to specific SIDs.

With the *Windows 7* UAC default settings, the ProxyCB bot can obtain high integrity without triggering any prompt, by using the following steps:

- It checks and creates a mutex for the UAC pass component.
- It drops a DLL file in the %TEMP% folder. The filename is made up of a random eight-digit number and the extension '.dat'.
- It injects into the explorer.exe process to move the dropped file to %SYSTEM%\sysprep\cryptbase.dll.

  In this step, the bot just needs to find a process that has been signed with the *Windows Publisher* certificate. These signed processes can copy files to the system folder without any prompt being displayed using the IFileOperation method, even if they are running at a medium integrity level. Explorer.exe is one of these signed processes and runs from *Windows* start-up, so it is a good target.
- It executes %SYSTEM%\sysprep\sysprep.exe, and the fake cryptbase.dll file will be loaded.

  Sysprep.exe is an auto-elevation program [3] – it can automatically and silently elevate itself to high integrity, no matter who runs it.

  When a process starts, it will look for the needed DLL files in its own folder first, and fall back on the system folder. Although *Windows* has a list of 'Known DLLs', in which files will always be loaded directly from the system folder (Figure 4), cryptbase.dll is missing from the list. As a result, the %SYSTEM%\sysprep\cryptbase.dll file will be loaded instead of the real cryptbase.dll file in the system folder.

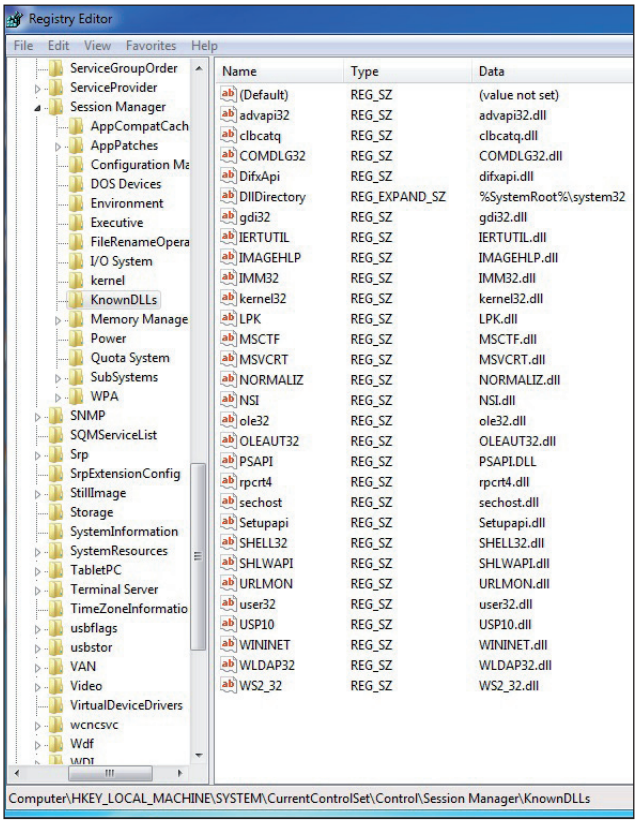

Figure 4: List of 'Known DLLs' – cryptbase.dll is missing.

- The fake cryptbase.dll is a tiny file measuring only 2,048 bytes. As soon as it is loaded, it will create a new instance of the ProxyCB installer and exit the process with a special ExitCode to notify the first installer process (see Figure 5).

```
; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
                public DllEntryPoint
DllEntryPoint   proc near

hinstDLL        = dword ptr  4
fdwReason       = dword ptr  8
lpReserved      = dword ptr  0Ch

                mov     eax, [esp+fdwReason]
                dec     eax
                jz      short loc_1000111C
                xor     eax, eax
                retn    0Ch
; ---------------------------------------------------------------------------

loc_1000111C:                           ; CODE XREF: DllEntryPoint+5↑j
                push    SW_SHOW         ; uCmdShow
                push    offset CmdLine  ; "c:\\ProxyCB\\sample.exe"
                call    WinExec                    the original ProxyCB bot file
                push    403212h         ; uExitCode
                call    ExitProcess
DllEntryPoint   endp
```

Figure 5: Code of cryptbase.dll.

Now, a new process of the ProxyCB installer starts with the high integrity level that has been inherited from sysprep.exe (see Figure 6).

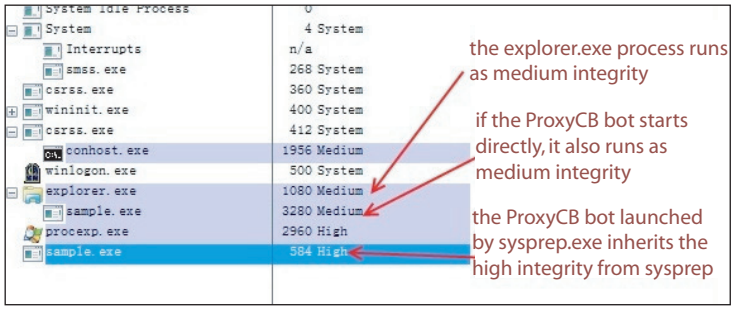

Figure 6: The ProxyCB bot inherits high integrity.

Dropping the file

The installer tries to drop a DLL component which was embedded as a resource inside itself. If the process has administrator privileges, it tries to drop the file in the %SYSTEM% folder and falls back on the %COMMON_APPDATA% folder.

Modifying the registry

When the DLL file has been dropped successfully, the installer tries to modify the registry so that the dropped file will be loaded each time *Windows* starts up (see Figure 7).

If the process has administrator privileges, it will try to append the path of the dropped file after the following registry entry:

Key: HKLM\CurrentControlSet\Control\SecurityProviders

Value: SecurityProviders

If it fails, or the process does not have administrator privileges, it tries to create the following new registry entry:

Key: HKCU\ Software\Microsoft\Windows\ CurrentVersion\Run

Value: Windows Time

Data: rundll32.exe '{The dropped file}',EntryPoint.

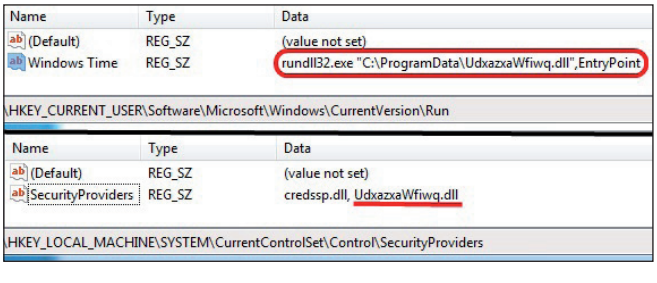

Figure 7: The installer tries to modify the registry so that the dropped file will be loaded on each system start-up.

LOADING PROCESS

Loading the DLL component

By now, the dropped DLL component has successfully been installed in the system and the registry has been changed so that the component will be loaded on system start-up. But the DLL has not been loaded yet. So the installer process does one last thing: it loads the DLL component.

It will try the following methods, depending on the version of the current system and the process privilege:

- It executes rundll32.exe using the ShellExecuteEx method with the parameter '{The DLL Component}',EntryPoint'.
- It shuts down and reboots *Windows* so that the DLL can be loaded when *Windows* starts. It will post the message shown in Figure 8, but you would be lucky to see it because the timeout is set to just one second.

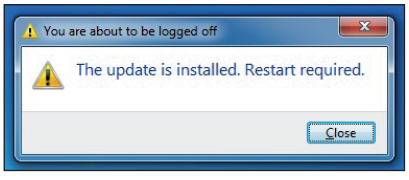

Figure 8: The shutdown message.

- It appends the DLL component to explorer.exe using the CreateRemoteThread method. The LoadLibraryA method is used as the starting address. The path of the DLL component will be injected into the explorer.exe process and used as the parameter of LoadLibraryA (see Figure 9). This is a popular method to load a DLL file into the system memory.

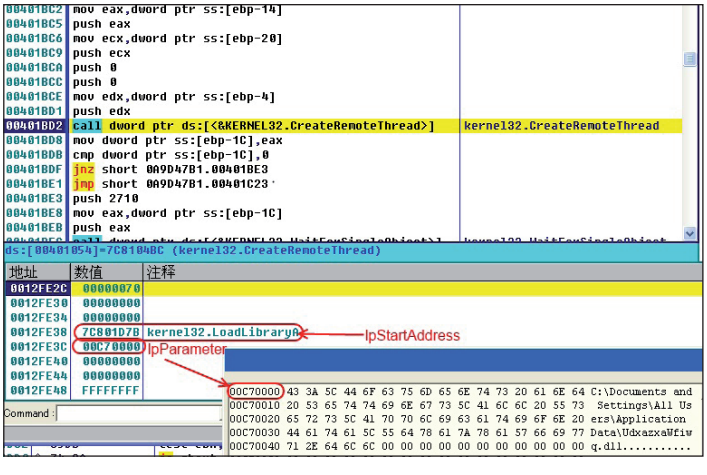

Figure 9: CreateRemoteThread method.

- It simply uses the LoadLibraryA method to load the DLL component into the current process. This is the simplest way to load the file, but this method needs the installer process to be kept alive and may cause some firewall alerts later, so it is used as a last resort.

Loading the real ProxyCB

The dropped DLL component loads the real ProxyCB payload, which will perform the malicious activities. The payload is a DLL file too, and it has no export functions except the DllEntryPoint.

When the dropped DLL component is being loaded, it checks the name of the current process first. If it finds that it is the rundll32.exe process that has loaded it (via loading method 1 or system start-up with the second registry), then it tries to append itself to the explorer.exe process by creating a remote thread.

Next, it tries to load the real ProxyCB payload. It uses the VirtualAlloc method to allocate new system memory, and copies the code and data of the payload into the memory by parsing the dropper's PE_Header. Finally, it fixes the IAT manually and invokes the DllEntryPoint.

It would be much easier to load the real ProxyCB payload using the LoadLibraryA method, but it would need to drop the file first. The bot loads the file manually, which won't drop the real ProxyCB payload. In this way, it might avoid some file-detection-based anti-virus programs.

COMMUNICATION ROUTINE

When the payload is being loaded, it first creates a mutex to avoid another instance of the installer. Then it tries to connect to the C&C server that is hard-coded in the binary. It keeps trying to connect to the C&C server at 10-second intervals until a successful connection is made.

Testing SMTP servers

Before it connects to the C&C server, it will test for some mail servers on port 25 (SMTP). If any are accessible, a flag will be set. The host names of the testing mail servers are hard-coded (Figure 10).

```
listMailServers dd offset aYahoo_com     ; "YAhoO.Com"
                dd offset aHotmail_com   ; "HOtMaIl.cOM"
                dd offset aGmail_com     ; "GMaIl.cOM"
                align 8
```

Figure 10: List of the testing mail servers.

Communication protocol

The bot creates a TCP connection with the C&C server on port 1001, then sets the socket timeout to two minutes. Now, the communication starts.

Structure of MessagePacket

The message packet received from the C&C server has the same structure as the message packet that is sent out. The packet is 26 bytes in length and has the following layout:

```
struct MessagePacket {
  BYTE(9) magic;  // static 85 B2 04 77 CE 38 E0 33 04
  BYTE    cmdType;
  WORD    dataWord1;
  WORD    dataWord2;
  DWORD   dataDword3;
  DWORD   dataDword4;
  DWORD   dataDword5;
}MessageSend, MessageRecv;
```

Phone home

When the communication starts, a phone home message is first sent to the server to register the victim machine as a client. This packet contains two DWORDs of information based on the victim machine. The first DWORD is one of the following – it checks and selects the first valuable data (not 0, 1 or -1):

- The volume serial number of the victim machine
- The hash value of the computer name
- The hash value of the user name associated with the current thread.

The second DWORD is the dwLowDateTime area of the %WINDOWS% directory creation time.

The first message packet is set up as follows:

```
MessageSend.cmdType = 1;
MessageSend.dataWord1 = 1;
MessageSend.dataWord2 = Flag for whether the C&C
server has been changed;
MessageSend.dataDword4 = VolumeSerialNumber/
HashComputerName/HashUserName;
MessageSend.dataDword5 = dwLowDateTime of %WINDOWS%
creation;
```

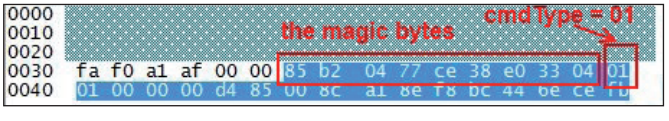

Figure 11: The phone home message packet.

Command types

After the phone home message has been sent, the bot starts to receive messages from the server and takes the

MessageRecv.cmdType as a command to determine what to do next:

MessageRecv.cmdType = 1: feedback for the phone home message.

MessageRecv.cmdType = 2: starts the proxy thread.

MessageRecv.cmdType = 3: retries the communication with the C&C server.

MessageRecv.cmdType = 4: keeps alive.

MessageRecv.cmdType = 5: changes the C&C server.

MessageRecv.cmdType = 6: restarts the communication with the C&C server.

Command 1

When the C&C server receives the register message, it will respond with a feedback message that contains timeout information for the communication. The cmdType area of the feedback message packet should be 1. This command will reset the timeout for the communication to MessageRecv.dataDword4×2 seconds:

```
MessageRecv.cmdType = 1;
MessageRecv.dataDword4 = TimeOut/2;
```

Command 4 and KeepAlive thread

When the communication starts, a KeepAlive thread will be launched.

As soon as the feedback from the phone home message is resolved, the KeepAlive thread starts to send a message packet (Figure 12a) to the C&C server periodically:

```
MessageSend.cmdType = 4;
MessageSend.dataWord1 = 1; static
MessageSend.dataWord2 = result of the test for SMTP servers
MessageSend.dataDword4 = VolumeSerialNumber/HashComputerName/HashUserName
MessageSend.dataDword5 = dwLowDateTime of WinDir creation
```

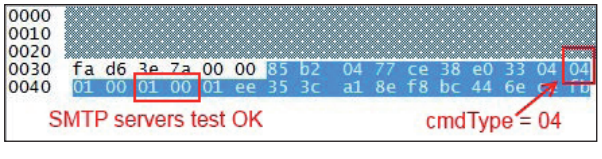

Figure 12a: KeepAlive message sent.

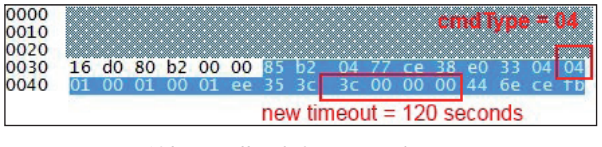

Figure 12b: Feedback for KeepAlive message.

When the C&C server receives the KeepAlive message, it just feeds back a message (Figure 12b), and the timeout of the communication will be reset:

```
MessageRecv.cmdType = 4;
MessageRecv.dataDword4 = TimeOut/2;
```

Command 5

Command 5 indicates that the C&C server should be changed.

```
MessageRecv.cmdType = 5;
MessageRecv.dataWord2 = flag to indicate that the server has been changed;
MessageRecv.dataDword4 = new IP address;
MessageRecv.dataDword5 = new port;
```

The bot first terminates the communication with the current C&C server, then uses the MessageRecv.dataDword4 and MessageRecv.dataDword5 as the new host and port, respectively, to create a new connection. It then initiates the new communication with a phone home message. In the meantime, a flag will be set to indicate that the C&C server has been changed.

Commands 3 and 6

These two commands are similar. They both terminate the current communication and start a new one. But there are some differences between them:

Command 3 will increase a counter each time it has tried to establish communication with the current server. If the C&C server has been changed and the counter reaches a maximum number (e.g. the maximum is 10 times for the sample we looked at), it will restore the original server for communication.

Command 6 just restarts the communication with the C&C server, there is no counter.

Command 2

This is the most important command for ProxyCB. It creates a new proxy thread. The proxy thread will try to connect to the C&C server on another port (e.g. port 1002 was used in this case). The new port is saved in the received message packet.

```
MessageRecv.cmdType = 2;
MessageRecv.dataWord1 = new port for proxy thread connection;
MessageRecv.dataWord4 = tagProxyThread1;
MessageRecv.dataWord5 = tagProxyThread2;
```

If the connection succeeds, or it has retried three times, a feedback message will be sent:

```
MessageSend.cmdType = 2;

MessageSend.dataWord1 = flag for the new connection
success or not;

MessageSend.dataWord4 = tagProxyThread1;

MessageSend.dataWord5 = tagProxyThread2;
```

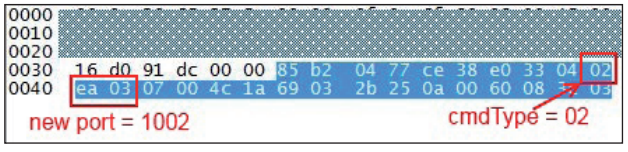

Figure 13a: Command 2 message packet received.

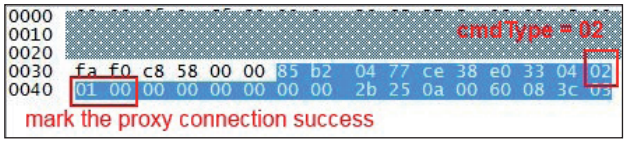

Figure 13b: Command 2 feedback message.

The proxy thread

When the C&C server receives the feedback message, it will send a new message that contains IP address and port information. It usually points to a mail server (e.g. yahoo.com) on port 25 (SMTP), port 80 (HTTP) or port 443 (HTTPS).

The mail server message may have two different structure types, as shown in Figure 14.

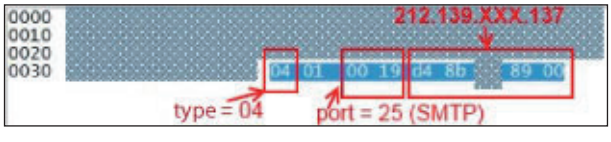

Figure 14a: Mail server message type 04.

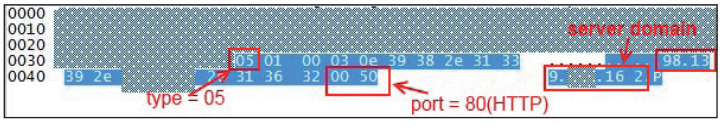

Figure 14b: Mail server message type 05.

When a TCP connection is created with the mail server, the bot will act as a proxy between the C&C server and mail server.

The C&C server starts to send SMTP commands on port 25 (such as 'HELO', 'MAIL FROM', etc.) or HTTP commands on port 80 or port 443 (such as 'GET', 'POST', etc.) to the ProxyCB client.

The bot receives the data and forwards it to the mail server. It then tries to get a response and send it back to the C&C server. After that, an email will be sent (see Figure 15).

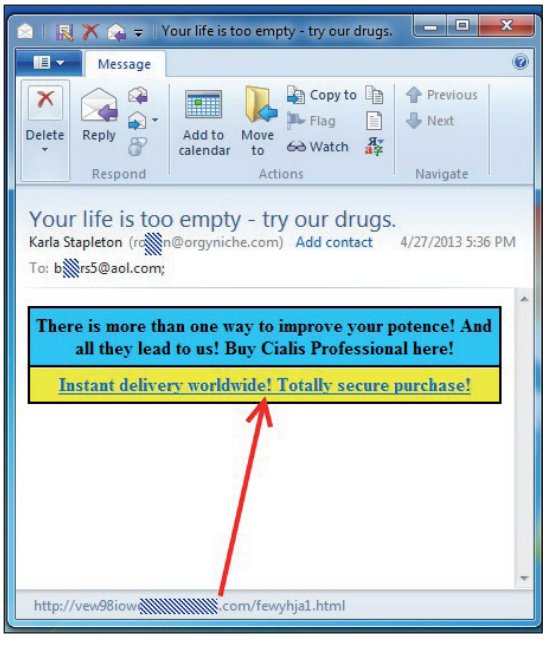

Figure 15: The sent mail.

CONCLUSIONS

Unlike other spam bots, ProxyCB does not have a component either to build the spam itself or to retrieve a spam template from the C&C server. As its name indicates, it only connects to the C&C server and forwards spam as a proxy. In this way, it is easy to update the spam content, email server and the target mailing addresses. The disadvantage of this method is that if the C&C server is not accessible, the bot will stop working immediately, and the C&C server is exposed in this case.

REFERENCES

[1] Wikipedia. User Account Control. https://en.wikipedia.org/wiki/User_Account_Control.

[2] MSDN. Windows Integrity Mechanism Design. http://msdn.microsoft.com/en-us/library/bb625963.aspx.

[3] List of Windows 7 (beta build 7000) auto-elevated binaries. http://withinwindows.com/2009/02/05/list-of-windows-7-beta-build-7000-auto-elevated-binaries/.

MALWARE ANALYSIS 3

SOLARBOT BOTNET

He Xu
Fortinet, Canada

Solarbot, a.k.a. Dapato or Napolar, is a classical botnet that has been around for a long time. It is usually used for spreading other malware. Like its competitors, this malware often comes with built-in DDoS and proxy modules. The most recent version of Solarbot attempts to add *Tor* network support to conceal its C&C server. However, it seems that this feature is either still undergoing development or has been disabled. The toolkit sells for around $200 and the source code is available for 100 bitcoins (approx. US$15,000) from the website hxxp://solarbot.net. Let's take a closer look.

TLS CALLBACK PROCEDURE

The bot carries an abnormal loader with a special PE header which has no entry point, and the ImageBase is not the usual default 0040000 or 01000000 (see Figure 1).

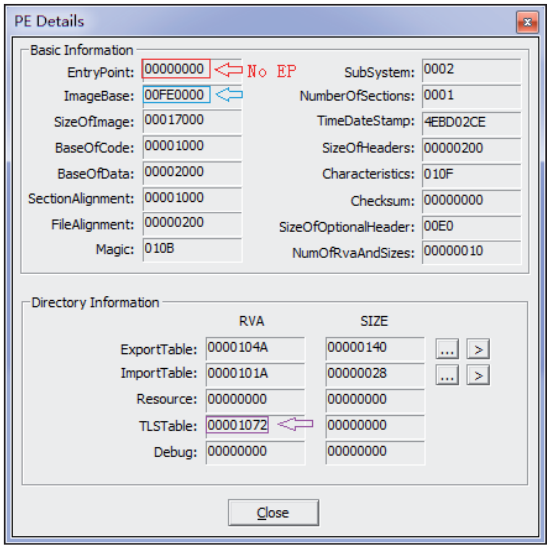

Figure 1: EntryPoint and ImageBase details.

The virus entry point is located in the TLS (Thread Local Storage) table, which is usually empty in the Data Directory list.

The structure of the IMAGE_TLS_DIRECTORY is as follows:

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
   ULONG StartAddressOfRawData;
   ULONG EndAddressOfRawData;
   ULONG AddressOfIndex;
   ULONG AddressOfCallBacks;
   ULONG SizeOfZeroFill;
   ULONG Characteristics;
 } IMAGE_TLS_DIRECTORY32, *PIMAGE_TLS_DIRECTORY32;
```

Let's look at the real data in the bot. There are two TlsCallback functions in the PE file (see Figure 2).

```
TlsDirectory    dd 0                    ; StartAddressOfRawData
                                        ; DATA XREF: HEADER:Data
                dd 0                    ; EndAddressOfRawData
                dd 0FE1096h             ; AddressOfIndex
                dd 0FE108Ah             ; AddressOfCallBacks
                dd 0                    ; SizeOfZeroFill
                dd 0                    ; Characteristics
TlsCallbacks    dd offset TlsCallback_0
                dd offset TlsCallback_1
                dd 0
TlsIndex        dd 0
```

Figure 2: Two TlsCallback functions.

When the bot is loaded by the system (PE loader), the TlsCallback function will be invoked ahead of the EPO.

The first TlsCallback, TlsCallback_0, is an empty function (see Figure 3). This might be used to trick anti-virus engines.

```
                public TlsCallback_0
TlsCallback_0   proc near               ;

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     eax, [ebp+arg_4]
                mov     eax, [ebp+arg_8]
                leave
                retn    0Ch
TlsCallback_0   endp
```

Figure 3: The first TlsCallback is an empty function.

The second TlsCallback function uses the dynamic TLS approach to insert a new callback procedure in memory. So when TlsCallback_1 returns, the TlsDirectory changes, as shown in Figure 4.

```
TlsDirectory    dd 0                    ; StartAddressOfRawData
                                        ; DATA XREF: HEADER:DataDir
                dd 0                    ; EndAddressOfRawData
                dd 0FE1096h             ; AddressOfIndex
                dd 0FE108Ah             ; AddressOfCallBacks
                dd 0                    ; SizeOfZeroFill
                dd 0                    ; Characteristics
TlsCallbacks    dd offset TlsCallback_0
                dd offset TlsCallback_1
                dd offset TlsCallback_2
TlsIndex        dd 0
```

Figure 4: The TlsDirectory changes.

The TlsCallback_2 function decrypts all code using the RC4 algorithm and the fixed double-word key 0x0F5BC5C9.

FAKE EXPORT DIRECTORY

The bot loader does not have real export functions, but it has an abnormal export directory that redirects to the ImageBase (MZ header), as shown in Figure 5.

```
ExportDir       dd 0                    ; Characteristics
                                        ; DATA XREF: HEADER:DataDir
                dd 0                    ; TimeDateStamp
                dw 0                    ; MajorVersion
                dw 0                    ; MinorVersion
                dd 0                    ; Name
                dd 0                    ; Base Rva, 0 means ImageBase
                dd 50000000h            ; NumberOfFunctions
                dd 0                    ; NumberOfNames
                dd 1000h                ; AddressOfFunctions
                dd 1000h                ; AddressOfNames
                dd 1000h                ; AddressOfNameOrdinals
```

Figure 5: Export directory redirects to the ImageBase.

We can see that the functions count declared in the structure is too large, and the Base is 0. This special structure will cause several debugger applications to enter an exception and thus be unable to analyse the bot.

DEBUGGER & DEBUGGEE MECHANISM

More and more bots are integrating debugger engines, and Solarbot is no exception. The bot uses this feature for anti-debugging purposes, and executes different code. The bot's internal debugger engine is much simpler than that of ZAccess, for example, but it is still effective.

Debugger

The bot will restart itself as a debuggee by calling the CreateProcessW API with the parameter CreateFlags DEBUG_ONLY_THIS_PROCESS, then it will enter the main loop to handle debuggee events such as CREATE_PROCESS_DEBUG_EVENT, EXCEPTION_DEBUG_EVENT and EXIT_PROCESS_DEBUG_EVENT. For other events, the bot's debugger just calls the ContinueDebugEvent API with the parameter dwContinueStatus DBG_CONTINUE.

The debugger will inject all code into the newly allocated memory of the debuggee while handling the debug event CREATE_PROCESS_DEBUG_EVENT.

After that, it will modify the debuggee's entry point code with the PUSH_RET instruction when it handles EXCEPTION_DEBUG_EVENT and the corresponding EXCEPTION_DEBUG_INFO structure with ExceptionCode EXCEPTION_BREAKPOINT. This modified code will be triggered when the debuggee runs before its entry point. This means that all executable code in the debuggee will be overwritten by the debugger.

The debugger also modifies another remote function, which we will discuss in the next section.

The debugger handles the INT3 breakpoint, but only sets a stack flag and calls the ContinueDebugEvent API with dwContinueStatus DBG_CONTINUE as the parameter.

The debugger will terminate itself if the stack flag marker is found.

Debuggee

As we know, the entry point of the debuggee has been replaced by the debugger. It will run the code shown in Figures 6–8.

```
00FE92D3 RunAs_Debuggee:                                ; CODE
00FE92D3                push    [ebp+pk9ECh]
00FE92D9                call    call_ZwTerminateProc
00FE92DE                int     3
00FE92DF                mov     eax, [ebp+pk9ECh]
```

Figure 6: Run as debuggee.

```
call_ZwTerminateProc proc near          ; CODE XREF: MainSub

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                push    0
                push    0FFFFFFFFh
                mov     eax, [eax+D.ZwTerminateProcess]
                call    eax
                leave
                retn    4
call_ZwTerminateProc endp
```

Figure 7: The bot appears to terminate itself.

It appears as if the bot will terminate itself permanently (see Figure 7). However, this does not happen, since the bot's debugger modifies the code, as shown in Figure 8.

```
call_ZwTerminateProc:                   ; CODE XREF: MainSub+13CF↓p
                push    offset RemoteSub
                retn
; ---------------------------------------------------------------------------
                push    0
                push    0FFFFFFFFh
                mov     eax, [eax+Data9ECh.ZwTerminateProcess]
                call    eax
                leave
                retn    4
```

Figure 8: The debugger modifies the code.

The redirected code will check the current process. If the path is %startup%\lsass.exe, the bot will return to the parent function. Otherwise, it will install itself.

The installed bot triggers the EXCEPTION_BREAKPOINT (INT3) debug event at address 00FE92DE. The bot's debugger will ignore the INT3 event and make sure the EIP points to the next instruction correctly.

Finally, the debuggee will try to inject malicious code into explore.exe, and quit.

VEH EXCEPTION

The malicious code injected into explore.exe uses a special trick to pick up and decrypt C&C information from the internal lists.

It installs a VectoredExceptionHandler callback function into the current VEH chain using RtlAddVectoredExceptionHandler (see Figure 9).

Then it executes the HLT instruction and causes the EXCEPTION_PRIV_INSTRUCTION exception, which will be processed by the KiUserExceptionDispatcher API. This API will call all callback functions in the VEH chain to solve the exception – so the newly added VEH callback function will be called to handle the exception.

```
00FE7EAC  mov     eax, [eax+D.RtlAddVectoredExceptionHandler]
00FE7EB2  call    eax
00FE7EB4  mov     edx, [ebp+arg_0]
00FE7EB7  mov     [edx+D.pExcHdr_List], eax
00FE7EBA  hlt
```

Figure 9: A VectoredExceptionHandler callback function is installed.

The VEH callback function saves the Context structure and then hooks the KiUserExceptionDispatcher API. Finally, it returns the EXCEPTION_CONTINUE_EXECUTION status, which means 'exception is dismissed, continue execution at the point at which the exception occurred'. As a result, the same exception will occur again.

As the KiUserExceptionDispatcher API has been hooked, the hook function will increase the EIP pointing exception address of HLT by 0x2E, then call the ZwContinue API with the parameter Context including the updated EIP (see Figure 10).

```
mov     eax, [esi+D.pContext2D0h]
mov     edx, [eax+CONTEXT._Eip]
add     edx, 2Eh
mov     eax, [esi+D.pContext2D0h]
mov     [eax+CONTEXT._Eip], edx
push    0
push    [esi+D.pContext2D0h]
mov     eax, [esi+D.ZwContinue]
call    eax
```

Figure 10: The ZWcontinue API is called with the parameter Context.

Finally, the EIP will point to address 0xFE7EE8 (see Figure 11), and the newly added VEH callback function will be removed.

```
00FE7EE8  mov     eax, [ebp+arg_0]
00FE7EEB  push    [eax+D.pExcHdr_List]
00FE7EEE  mov     eax, [ebp+arg_0]
00FE7EF1  mov     eax, [eax+D.RtlRemoveVectoredExceptionHandler]
00FE7EF7  call    eax
00FE7EF9  mov     eax, [ebp+arg_0]
00FE7EFC  push    [eax+D.pContext2D0h]
00FE7EFF  call    call_VirtualFree
00FE7F04  leave
00FE7F05  retn    4
```

Figure 11: VEH callback function is removed.

Why does the bot use this trick? First, KiUserExceptionDispatcher is the most important API that is always called by debuggers, and it's impossible to set a breakpoint in it as it will cause the system to become unstable. Second, the bot will fetch and decrypt the next C&C information in the hook function just before updating the EIP.

C&C COMMUNICATION

Traffic for downloading other malware

Figure 12 shows an example of the traffic the bot receives when it gets a command from the C&C server.

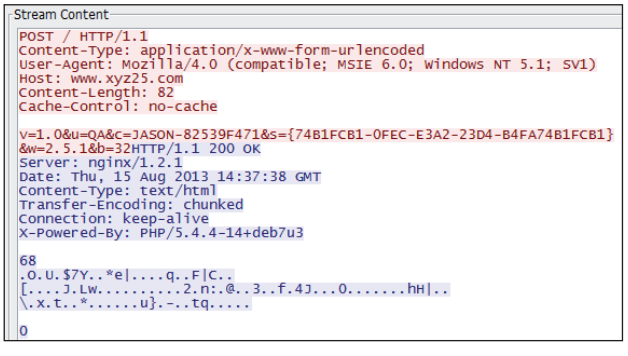

Figure 12: Command from the C&C server.

The send package is a clear string that is generated with following pattern:

```
v=%d.%d&u=%s&c=%s&s=%s&w=%d.%d.%d&b=%d
```

A real example is as follows:

```
v=1.0&u=QA&c=JASON-82539F471&s={74B1FCB1-0FEC-E3A2-
23D4-B4FA74B1FCB1}&w=2.5.1&b=32
```

As we can see, 'v' is the bot version, which is hard-coded; 'u' is the username, which is grabbed from a call to the GetUserNameA API; 'c' is the current computer name, which is generated from a call to the GetComputerNameA API; 's' is the ClsID, which is generated using various pieces of system information such as the Drive C Serial Number; 'w' is the *Windows* version from a call to the

GetVersionEx API; finally, 'b' indicates whether the system is running as 32 or 64 bits.

The received package is encrypted, as shown in Figure 13.

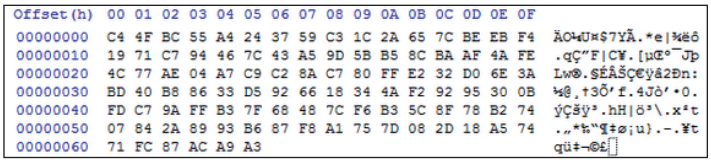

Figure 13: The received package is encrypted.

The encryption algorithm is RC4, and the key is the ClsID included in the sending package parameter as 's'.

Figure 14 shows the received data after decryption.

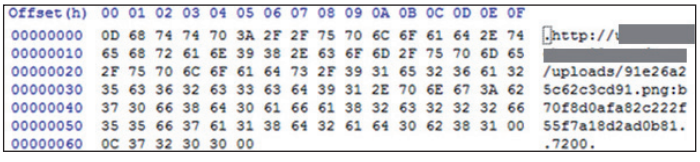

Figure 14: Received package after decryption.

Just like Andromeda, Solarbot uses different command IDs to identify different jobs. The current variant supports 14 commands, which range from 01 to 0x0E. The example above shows command 0D, which instructs the bot to download another piece of malware from a specified link and includes an MD5 tail for verification. The package also includes command 0C, which updates the default sleep time period.

The bot uses the GET method to download the additional malware, as shown in Figure 15.

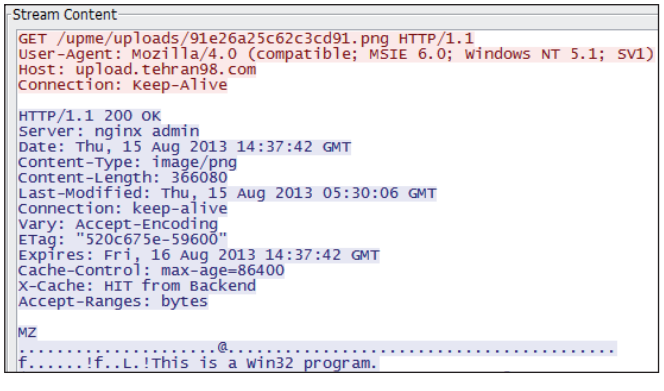

Figure 15: The bot uses GET to download malware.

Following the command 0D routine, the bot will check the downloaded file's MZ and PE signatures, then calculate the whole file's MD5 and compare it with the MD5 tail included in the received package. If everything matches, the bot will drop the malware into the %AppData% folder, using a random filename that follows the pattern '%08lX.exe'.

Traffic idle

If there are no more commands, there will be much less traffic than in the previous example (see Figure 16):

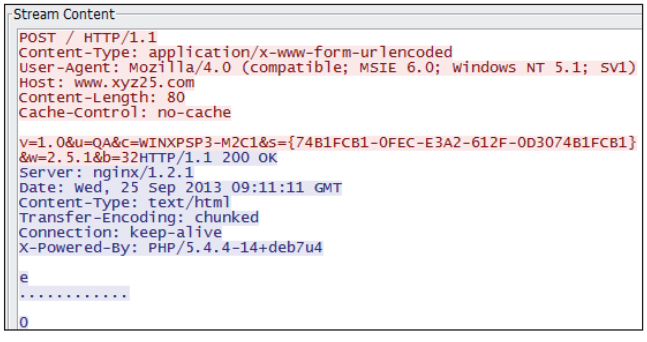

Figure 16: No more commands.

The received package is shown in Figures 17 and 18.

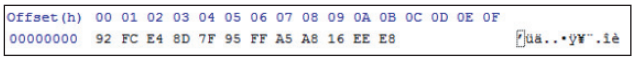

Figure 17: Received package binary data.

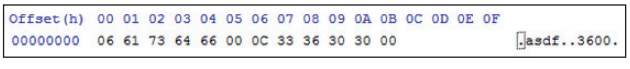

Figure 18: Received package after decryption.

Command 06 instructs the bot to set the tag to '1' (from the default 0) to indicate the end of the previous DDoS attack job, if one existed. The following command, 0C, instructs the bot to update the default sleep time to 3,600ms.

Traffic for DDoS

We know the bot has a DDoS attack feature, so let's look at some real attack traffic (Figure 19):

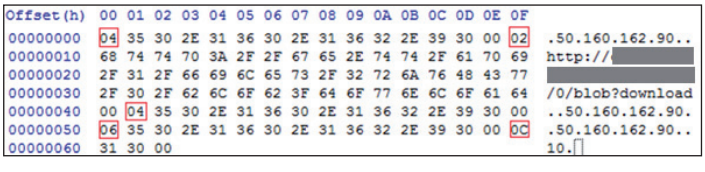

Figure 19: Attack traffic.

The command 04 signifies the start of a UDP DDoS attack. Figure 20 shows what the victim's traffic will look like.

According to the code, the bot opens 10,000 connections with the victim IP at the same time. The DDoS will not stop unless the bot receives a further command with ID

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.17.52.122	50.160.162.90	UDP	921	Source port: 1409 Destination port: 58233
2	381.000000	172.17.52.122	50.160.162.90	UDP	791	Source port: 1409 Destination port: 64251
3	942.000000	172.17.52.122	50.160.162.90	UDP	791	Source port: 1410 Destination port: 64251
4	1422.000000	172.17.52.122	50.160.162.90	UDP	1023	Source port: 1409 Destination port: 37651
5	1863.000000	172.17.52.122	50.160.162.90	UDP	1023	Source port: 1410 Destination port: 37651
6	2344.000000	172.17.52.122	50.160.162.90	UDP	1023	Source port: 1411 Destination port: 37651
7	2824.000000	172.17.52.122	50.160.162.90	UDP	909	Source port: 1409 Destination port: 18309
8	3305.000000	172.17.52.122	50.160.162.90	UDP	909	Source port: 1410 Destination port: 18309
9	3786.000000	172.17.52.122	50.160.162.90	UDP	909	Source port: 1411 Destination port: 18309
10	4246.000000	172.17.52.122	50.160.162.90	UDP	909	Source port: 1412 Destination port: 18309
11	5208.000000	172.17.52.122	50.160.162.90	UDP	947	Source port: 1409 Destination port: 35167
12	5668.000000	172.17.52.122	50.160.162.90	UDP	947	Source port: 1410 Destination port: 35167
13	-1694594.999989	172.17.52.122	50.160.162.90	UDP	947	Source port: 1411 Destination port: 35167
14	-1694594.999968	172.17.52.122	50.160.162.90	UDP	947	Source port: 1412 Destination port: 35167
15	-1694594.999948	172.17.52.122	50.160.162.90	UDP	947	Source port: 1414 Destination port: 35167
16	-1694594.999908	172.17.52.122	50.160.162.90	UDP	946	Source port: 1409 Destination port: 352

Figure 20: Victim's traffic.

06. In this case, command 06 was just behind the second command 04 (see Figure 19) and was included in the same package, so the attack time was not very long.

As far as we can tell, the victim IP does not belong to any organization or business website, so this example may be a test, and may not cause too much damage.

Command 02 instructs the bot to download malware without MD5 verification (see Figure 19). In this case, the bot just downloads the binary from the specified link then drops it into the %AppData% folder and runs it.

Traffic for dialler

Command ID 0A instructs the bot to open iexplore.exe or another default Internet browser to open the URL. This may currently just be for testing purposes.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   0A 61 6C 69 62 61 62 61 2E 63 6F 6D 00 0C 31 30   .alibaba.com..10
00000010   00                                                .
```

Figure 21: Command 0A.

Full description for all commands

The following is a detailed description of all commands as seen in our analysis:

01: SetEvent to activate the next C&C communication

02: Download malware without MD5 verification, and run it in the %AppData% folder

03: Create thread for DDoS attack under TCP protocol

04: Create thread for DDoS attack under UDP protocol

05: Create thread for DDoS attack under TCP protocol

06: Set tag to indicate that the current DDoS attack has finished

07: Download malware without MD5 verification, and drop it in the %AppData% folder (without running it)

08: Create thread for DDoS attack under TCP protocol

09: Create thread as Proxy server

0A: Run iexplore.exe to open URL with flag NORMAL_PRIORITY_CLASS

0B: Run iexplore.exe to open URL with flags NORMAL_PRIORITY_CLASS and CREATE_NO_WINDOW

0C: Update the default sleep time

0D: Download malware with MD5 verification, and run it in the %AppData% folder

0E: Download malware with MD5 verification, and drop it in the %AppData% folder (without running it)

0F: Update HTTP request header host string.

JOINING THE TOR NETWORK

Tor is a service for enabling anonymity and making Internet activity very difficult to track. Solarbot generates files such as %AppData%\tor.bin and %AppData%\torrc and stores them, but it does not appear to use them. We did not locate any *Tor* traffic in Solarbot sample replication, either. Our best guess is that this feature is still under development.

CONCLUSION

This botnet is very powerful and may become more aggressive in the future, with lots of evidence in the code to suggest that it is still undergoing development. We will continue to monitor its evolution.

REFERENCES

[1] http://blog.avast.com/2013/09/25/win3264napolar-new-trojan-shines-on-the-cyber-crime-scene/.

[2] http://www.infosecurity-magazine.com/view/34788/napolar-solarbot-trojans-share-dna/.

[3] http://www.malwaretech.com/2013/10/end-of-line-for-solar-bot-win32napolar.html.

MALWARE ANALYSIS 4

NOT EXPIR-ED YET

Raul Alvarez
Fortinet, Canada

Most advanced file infectors employ a strong encryption algorithm with a mix of anti-debugging and anti-analysis techniques, but once you overcome these challenges, they are straightforward with their file infection routine – searching for files to infect, computing the exact location to put the malware body, configuring the MZ/PE header, and there you have it: a file infector.

That's what Expiro looks like on the outside: a simple file infector with a straightforward encryption mechanism. But if we delve into its code, we will see something that sets it apart from regular file infectors.

This article will look into Expiro's simple, but meticulous sequence of steps that leads to the infection of a unique group of files before infecting the rest of the executable files in the system.

SIMPLE POLYMORPHIC ENGINE

Expiro starts by saving all register values to the stack and initializing the variable that contains the decryption key for the polymorphic engine.

For the first pass, the malware decrypts (0x32A00) 207,360 bytes using a key starting halfway through the .vmp0 section. It decrypts each byte from the middle of the .vmp0 section working backwards to the beginning of the section, where the least significant byte of the DWORD decryption key is used to XOR every byte in the given block of encrypted code.

After the first pass, the malware checks whether it has reached the required number of decryption passes. If it has not, it will execute the same decryption algorithm again.

On the second decryption pass, after decrypting 207,360 bytes, Expiro decrypts the same group of bytes back to its original form using the same algorithm. It makes the same decryption pass over and over again six times, always producing the same original form.

For every decryption pass, the malware increments a counter. The same counter is used within the decryption algorithm, combining it with the decryption key to produce a different result. But after six passes, the bytes are still in their original encrypted form. The changes can only be seen after the seventh decryption pass.

After performing the seventh decryption pass, the (0x32A00) 207,360 bytes are properly decrypted. A block of (0x24880) 149,632 bytes from the newly decrypted code is copied to the free space of the .vmp0 section. This is followed by copying another (0xA1C8) 41,416 decrypted bytes to the free space of the .vmp0 section, beyond the first block.

The decrypted code distributed in the free space of the .vmp0 section is not yet functional. Some of its binaries need further tweaking. These blocks of code need patching.

Expiro uses a table that contains keys with corresponding sizes. Each key is processed to produce patched code, with its size determining the number of times the patched code should be applied. The algorithm that produces the patched code also determines where in the decrypted code the patched code should be applied. Once a patch has been applied the correct number of times, the next key from the table is processed to produce patched code, and again applied to the decrypted code where the last patch was applied. The malware continues patching the decrypted code until all the keys from the table have been used.

HASHING INITIAL API NAMES

A common method of getting the imagebase address of kernel32.dll is by parsing the PEB (Process Environment Block). For file infectors, another method is to parse the import table of the host file.

Parsing the import table to locate kernel32 can be done in many ways. Expiro gets the DLL names and checks the seventh and third characters to determine if they match '3' and 'r' from 'kernel32.dll', respectively. If they match, the rest of the characters in 'kernel32' are also checked.

Then, Expiro gets the imagebase of kernel32.dll by zeroing out the least significant DWORD of the address of the first API found.

Afterwards, Expiro resolves the addresses of some of its APIs using their hash values.

Initially, a routine looks for the export table of kernel32.dll and locates the list of exported API names. The malware computes the hash value of each API name using its own hashing algorithm. Each hash value is compared against the hash value of 'GetCurrentThreadId' until they match. The address of the GetCurrentThreadId API is resolved by using the index pointing to the matched API name.

A similar routine resolves the following APIs: CreateThread, EnterCriticalSection, GetProcAddress, GetTickCount, InitializeCriticalSection, LeaveCriticalSection and VirtualProtect.

Commonly, the resolved API addresses are stored in a table with consecutive memory locations, but not in this case.

Once all the API addresses have been resolved, it is easy to determine the next actions of the malware by looking at the list of addresses. For Expiro, each resolved API address is

scattered in a different location in its virtual space, making it a challenge for analysts wanting to look at them all at once. Figure 1 shows the resolved APIs stored in different memory locations.

Once the initial APIs have been resolved, Expiro spawns a new thread.

memory loc	hash	API
010BA1A0	968C6217	InitializeCriticalSection
010C10E0	54484108	GetProcAddress
010C52A4	6C5C819E	GetCurrentThreadId
01199DD8	5454284E	VirtualProtect
0119A040	786B35A0	LeaveCriticalSection
0119BD80	786DCDD2	EnterCriticalSection
0119BEA8	483E47C0	GetTickCount
0119C62C	483D0F68	CreateThread

Figure 1: The resolved APIs which are stored in different memory locations.

ON-DEMAND DECRYPTION ALGORITHM

Upon execution of the new thread, Expiro executes another decryption algorithm to generate the API names that the malware needs.

The API names are grouped by libraries with the exception of the LoadLibraryA and GetModuleHandleA APIs. Each group of APIs passes through a routine that contains the decryption algorithm and address resolver.

The on-demand decryption algorithm computes each byte of data with a byte taken from the key string 'V@sna8TbCzTSrs:s[fR@6'. An incrementing pointer determines which byte will be used to decrypt the byte from the given memory location. When the last character from the key string is reached, the pointer will reset to point back to the first character.

There are separate routines for each group of APIs per library (DLL). After decrypting the API names, each routine checks if the module/library already exists by using the GetModuleHandleA API. If it does not exist, it will load the module by calling the LoadLibraryA API. Then, the routine will use the GetProcAddress API to determine the actual API addresses.

Every string or name used by Expiro passes through the on-demand decryption algorithm. All the API names, library names, mutex names, and all other strings use the same decryption. These names and strings are only decrypted when Expiro needs them (see Figure 2).

UNDER WOW64

After getting the addresses of all the required APIs, Expiro checks if it is running under WOW64 (32-bit *Windows* on 64-bit *Windows*).

Figure 2: Part of the on-demand decryption algorithm with sample encrypted and decrypted strings.

Initially, it checks if the operating system supports WOW64 by checking the OS version using the GetVersionExA API. This is followed by getting its own PID (process id) using the GetCurrentProcessId API, and getting the handle by opening it using the OpenProcess API. Finally, it uses the IsWow64Process API to determine if it is running under WOW64.

This is also the part where Expiro determines the kind of infection needed, since the malware is capable of infecting both 32- and 64-bit executables.

If the malware is not running under WOW64, it can function as a 32-bit piece of malware on 32-bit operating systems, or as a 64-bit piece of malware on 64-bit operating systems.

CHECKING SECURITY ACCESS LEVEL

After determining whether it is running under WOW64, Expiro checks its security access level using the following routines:

The malware gets the username of the current thread using the GetUserNameA API, and compares it to the newly decrypted strings 'SERVICE' and 'SYSTEM'. Expiro skips this routine if the username contains the strings 'SERVICE' or 'SYSTEM' – it assumes that it has a higher security level if the username contains these strings.

If it doesn't have the above credentials, it gets the computer name using a simple call to the GetComputerNameA API, and compares it against the username. If the username of the current thread is also the computer name, the malware will again try to skip the current routine.

Otherwise, it proceeds to acquire the environment block of the current process using the GetEnvironmentStrings API. Expiro searches for the strings 'systemprofile' and 'ervice' within the environment block. Since the malware checks byte by byte, 'ervice' will yield to true if it finds either 'service' or 'Service'. Normally, services that run in the system will contain these strings.

If the malware has a higher security access level on the system, it will skip the routine that escalates its privilege level.

HANDLING MUTEX

After determining the malware's security access level, Expiro decrypts more strings ('kkq-vx' and '%s_mtx%u') and checks if a mutex named 'kkq-vx-mtx28' exists, using the OpenMutexA API. If the mutex exists, it will terminate it using a call to the CloseHandle API.

The malware will loop back on the start of this routine to check for the existence of other mutexes by changing the value 28 in 'kkq-vx-mtx28'. (The value may be different in some infected samples.) The number increments by one each time, until it reaches 99. In simpler terms, Expiro checks for the existence of mutexes 'kkq-vx-mtx28' up to 'kkq-vx-mtx99'. If any of these exist, they will be terminated.

After terminating all of these mutexes, a new routine is started. This time, a mutex named 'kkq-vx-mtx1' is created using the CreateMutexA API. This is followed by creating a second mutex named 'kkq-vx-mtx28'. Finally, the WaitForSingleObject API is called, with parameter WAIT_FOREVER for the mutex 'kkq-vx-mtx28' (see Figure 3).

Afterwards, Expiro decrypts another string, 'gazavat-svc', and checks if a mutex named 'gazavat-svc' exists using another call to the OpenMutexA API. If it exists, it will also be terminated.

The mutex 'gazavat-svc' will be used later on, after the infection routine.

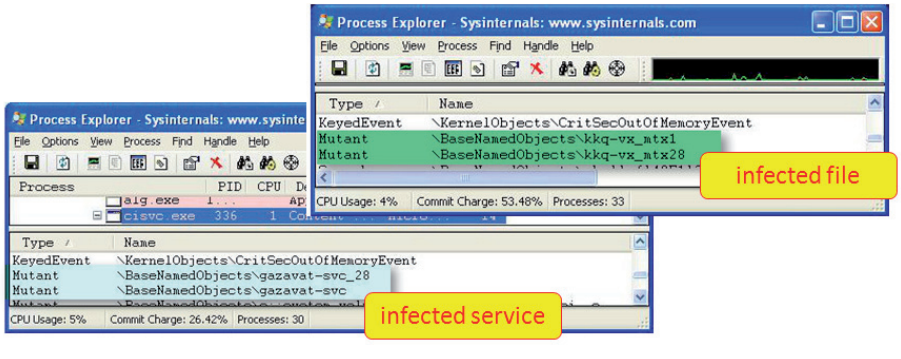

Figure 3: Mutexes for infected service and infected file.

ESCALATING ACCESS

After handling the mutexes, and if the security access level of the malware is not high enough, Expiro will try to escalate its privileges using the following routine:

Initially, the malware gets the PID (process ID) using the GetCurrentProcessId API, and gets the handle to the process using the OpenProcess API. Afterwards, it gets the access token of the process, and its data, using the OpenProcessToken and GetTokenInformation APIs, respectively.

Expiro uses the token in conjunction with the security descriptor. After initializing the security descriptor using the InitializeSecurityDescriptor API, the malware sets the discretionary access control list (DACL) using the SetSecurityDescriptorDacl API, followed by setting the owner information of the security descriptor using the SetSecurityDescriptorOwner API. The malware evaluates this information before it performs the escalation.

For the actual escalation, the malware acquires the locally unique identifier (LUID) of the SE_TAKE_OWNERSHIP_NAME privilege using the LookupPrivilegeValueA API, then sets the privilege using the AdjustTokenPrivileges API.

At this point, Expiro should have the necessary security privileges to take ownership of any process or file in the system.

PREPARING FOR INFECTION

After a considerable number of tasks and routines have been performed, Expiro is ready to choose which file to infect. But the preparation is not yet over: instead of just enumerating the executable files by performing simple calls to the FindFirstFileA and FindNextFileA APIs with '*.EXE' or '*.SCR' as parameters, Expiro wants a different set of executable files: services.

Let's look into the preparation.

First, Expiro gets the handle for the service control manager database using a call to the OpenSCManagerA API. This is followed by enumerating the services with name and status using the EnumServicesStatusA API with parameters:

dwServiceType = (0x30) SERVICE_WIN32_OWN_PROCESS and SERVICE_WIN32_SHARE_PROCESS

dwServiceState = (0x02) SERVICE_INACTIVE

Basically, Expiro is searching for inactive or stopped services in the system.

Expiro opens the service from the enumerated list using the OpenServiceA API, and retrieves the configuration parameters using the QueryServiceConfigA API. The service's configuration contains properties including service name, start-up type, service status, and path to the executable file of a given service.

Expiro is interested in the physical location of the executable file of a given service. It converts the whole path name to lower case, then checks if it contains '.exe' – making sure that it really is an executable file.

If the path name passes a series of checks, the malware converts it to a Unicode version of the string for use in a call to the SfcIsFileProtected API. Expiro wants to determine if the specified file is protected. If it is protected, the malware will remove the protection.

After removing the file's protection, the malware checks if the path name contains the strings 'rsvp.exe' (*Microsoft RSVP*), or 'chrome.exe' (*Google Chrome*) – if either of these strings is found, the malware will skip the infection routine.

INFECTION ROUTINE

After the necessary checks, Expiro will perform the infection routine for the selected service's executable file.

First, it opens the executable file using the CreateFileA API with GENERIC_READ and GENERIC_WRITE access. It acquires the file size using the GetFileSize API, then copies the file to the newly allocated memory using the ReadFile API, effectively creating an exact image of the service's executable file.

Once the image is loaded into memory, the malware parses the MZ/PE header to point to the import table and locate the DLL names. Expiro tries to locate kernel32.dll using the following routine:

After getting the DLL name from the import table list, the names will be converted to all caps. Then, the first filter is to check for the eighth byte (char '2') – most DLL names fail this check. The second filter is to check for the seventh byte (char '3') – ADVAPI32.DLL can still pass this one. The next filters are 'L', 'E', 'K', and 'R'. Finally, it saves the location of 'KERNEL32.DLL' for later use.

Once kernel32.dll is secured, the PE header of the image is expanded to make room for the new section header. After a few computations, Expiro modifies the new section header with the following sequence:

- The new section's virtual size gets the value (0x7d000) 512,000, which corresponds to the increase in size of the infected file.
- The starting relative virtual address (RVA) of the new section is also set, which varies between different infected files. Generally, the RVA of the new section is right after the end of the previous section.
- The NumberOfRelocations, NumberOfLineNumbers and PointerToLineNumbers fields are zeroed out.
- The malware uses '.vmp0' as the name of the new section, which we already know.
- The PointerToRelocations field is zeroed out.
- The value (0x7d000) 512,000 is placed in the SizeOfRawData field, similar to the VirtualSize field.
- The PointerToRawData or the file offset of the start of the new section is set with a value that depends on the size of the host file.
- The section's characteristics are set to (0xE0000000) Executable | Readable | Writable.

After setting up the new section header, the malware sets the SizeOfImage to 0x81000 (which varies between different infected samples). This is followed by incrementing the number of sections by one, to accommodate the newly added section.

Once the MZ/PE header has been modified and the new section header has been added, Expiro generates random values and patches a block of code containing a copy of the malware body.

This is followed by copying a portion of the image (from the entry point) to the patched block of code in memory.

Then, it copies the (0x7C6CD) 509,645 bytes of code (containing the patched code and the portion of the image) to the start of the new .vmp0 section.

Since the malware code at the new .vmp0 section is not yet encrypted (just recently patched), Expiro runs a simple encryption routine to encrypt the content of the .vmp0 section of the image, byte by byte. The encryption skips the copied portion from the image's entry point.

APPLYING THE CHANGES

After the modification of the service's executable image in memory, it will release the handle for the physical file using the CloseHandle API.

Afterwards, Expiro creates a '.vir' file, e.g. '{service_filename}.vir', using the CreateFileA API, which is followed by copying the infected service's image from memory to the .vir file, using the WriteFile API.

Then, the malware frees up the image using the LocalFree API and closes the handle to the .vir file. Expiro

copies the .vir file to its .exe counterpart (e.g. it copies 'XYXYXservice.vir' to 'XYXYXservice.exe'), using the CopyFileA API. After a successful copy, the malware deletes the .vir file using the DeleteFileA API.

Expiro checks if the service name of the infected file is found in the string '|wscsvc|WinDefend|MsMpSvc|NisSrv|', which contains the names of the services for Security Center, Windows Defender, Microsoft Antimalware Service and Microsoft Network Inspection, respectively.

If the service name is found, the malware disables the service permanently by calling the ChangeServiceConfigA API with the (dwStartType) SERVICE_DISABLED parameter. Then it exits the current routine.

If the service name of the newly infected file is not in the list, it will set the service's configuration to the following parameters: (dwServiceType) SERVICE_INTERACTIVE_PROCESS | SERVICE_WIN32_SHARE_PROCESS, and (dwStartType) SERVICE_AUTO_START, using the ChangeServiceConfigA API. Earlier, in selecting which services to infect, Expiro searched for inactive and stopped services. Once these services are infected, the malware changes their start type to start automatically when the operating system restarts. This is another technique for making sure the malware runs during the boot process.

Expiro then runs the infected service simply by calling the StartServiceA API. One of the markers that indicates that a service is infected is the presence of the mutex named 'gazavat-svc' (see Figure 3).

Afterwards, the malware will proceed with the infection of the rest of the inactive services, while the rest of Expiro's malicious activities will be executed within the service's process.

WRAPPING UP

Expiro relies heavily on the use of its on-demand decryption algorithm. Although not too complex, it serves its purpose well – not revealing everything all at once.

Expiro is not a new file infector, but it resurfaces from time to time, demonstrating more skills on each new appearance – infecting a service that gives a unique vantage point on traditional malicious activities; running the malware at computer restart without creating a start-up registry; using different mutexes for different types of infected process; escalating privileges; and executing the infected files without calling the CreateProcess or WinExec APIs.

Expiro has other malicious activities which are beyond the scope of this article. Simple or not, this is not the last time that we will hear from this malware.

TECHNICAL FEATURE

BYOT: BRING YOUR OWN TARGET

Gabor Szappanos
Sophos, Hungary

It is nothing new for a piece of malware to exploit a vulnerability found in an application – in fact, that is the routine procedure for infecting a computer. This approach does, however, have a weak point: the application in question must be installed on the target computer; furthermore, it must be a vulnerable version of it.

One malware sample we analysed recently breaks the traditional mould in two ways: the purpose of the exploitation is not intrusion, but to minimize the detectable system footprint, and it does not rely on preinstalled applications. Apart from that, so as not to break with tradition completely, the system infection is achieved via a common *Word* exploitation technique.

Following successful infection, only a handful of clean applications are left on the system, along with the encrypted payload file and a single registry key, which is a crucial element of the infection scheme.

The issue of whether or not the appropriate version of the vulnerable application is installed on the target system is eliminated simply: the trojan drops the vulnerable application onto the system itself, and uses it for its own purposes. In fact, the author of this malware does not take anything for granted: all the necessary components for the malware's execution are bundled and dropped onto the system, including regular *Windows* system binaries.

INSTALLATION PROCESS

The installation of the malware is a little complicated. It starts with a document exploit, runs through multiple intermediate dropper stages, and concludes in the final infected state with a handful of clean components and the encrypted payload on the system. The process is summarized in Figure 1.

1. Exploited carrier document

File size: 830,336 bytes
SHA1: 0ddae43498e1b03a274f8ca8b32cd48a1a440e7d
MD5: 6282568857a120a93de3af57e21952e1

The starting point of the infection chain is an encrypted *Excel* workbook with default hard-coded null password, the meaning of which was explained in [1]. The same vulnerability as described in [1] (CVE-2012-0158) was used in this case as well.

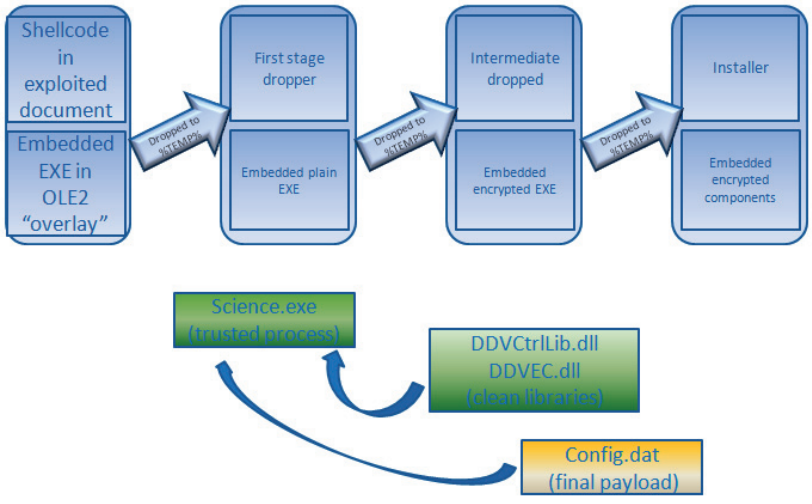

Figure 1: Installation flow of Simbot.

Figure 2: Exploit document structure.

The carrier document is a very unique compound, as illustrated in Figure 2. Normally, in document exploitations we see either *Excel* workbooks or *Word* RTF documents that contain the (usually multi-staged) shellcode with the encrypted payload executable appended. In this case, the first-stage shellcode is within an encrypted *Excel* workbook, the second-stage shellcode is in an appended *Word* RTF fragment, and then comes the encrypted executable. It gives the impression of a project that has been copy-pasted from different sources with minimal integration effort.

The encrypted workbook contains the first-stage shellcode, which enumerates open file handles, checking for the file size. The file size must be exactly 830,336 bytes – the size of the carrier workbook. Then it reads in, decodes and executes the content from file offset 0x1de00, at which the hexadecimal text representation of the second-stage shellcode is located.

The second-stage shellcode once again checks for the correct file size for the carrier workbook and searches for the start marker for the embedded .exe (TSRQPONMP). If the marker is found, the DWORD following it is used as the length of the embedded file, followed by the whole payload content, which will be encrypted with a single-byte XOR encoding, the key being decremented by one after each byte.

After the second-stage code, further shellcode fragments are found, which are not used and are corrupted when decrypting the shellcode (running over the real length). This is an indication that the carrier was created by reusing older components and overwriting the (longer) shellcode with the new code, not caring about what happens to the trailing remainder of the old code. Again, this underlines the minimal integration effort made in the creation of the exploited workbook.

2. First-stage dropper

File size: 654,675 bytes
SHA1: 16fbb14ef6c7ae9c401859aedf99cfd762f00794
MD5: dfed4bdf77892f2c62b8c68782c16132

This component is a very simple dropper. It reads the next stage executable from offset 0x1800 in 0x400 byte chunks, saves it to a temporary file in the %TEMP% folder, then executes the dropped file.

3. Intermediate dropper

File size: 647,168 bytes
SHA1: 79ef9296a2a0913e60a925da2f9d061ae3a364c7
MD5: 91d26990f22a4584e631395f5ae234c3

This dropper searches for a mutex named 'Sample06' to determine whether another instance of the dropper is already running – if it finds the mutex, it exits.

It checks for the presence of a debugger by looking for magic bytes in the allocated heap:

- 0ABABABABh (used by *Microsoft*'s debug-built HeapAlloc() implementation to mark 'no man's land' guard bytes after allocated heap memory)
- 0BAADF00Dh (used by *Microsoft*'s debug-built HeapAlloc() implementation to mark uninitialized allocated heap memory)
- 0FEEEFEEEh (used by *Microsoft*'s debug-built HeapFree() implementation to mark freed heap memory).

If a debugger is found, only an empty window with the title 'NewSetup' is displayed.

Otherwise, in an untainted environment, it decodes an offset-independent code (using a single-byte XOR algorithm, with key 0x97), and executes it.

This component creates the HKLM\SOFTWARE\Microsoft\Windows\Help -> Config registry key and saves the encrypted configuration data there (see Figure 3).

If, for any reason, saving the configuration data to the registry fails, then as a backup method, the same data is dumped into the file C:\Documents and Settings\All Users\NetWork\t1.dat.

This configuration data is used by the final stage payload, with the C&C server address extracted from the key value of the file.

Finally, it decodes and executes the next stage dropper.

4. Installer

File size: 466,872 bytes
SHA1: 5a22efba829c259f1cb17f9ffe529c398397e25c
MD5: 138f32de8f53fe651a7b6967c63cf7ac

This component is actually a *Windows* DLL with obfuscated entry code and a lot of redirections.

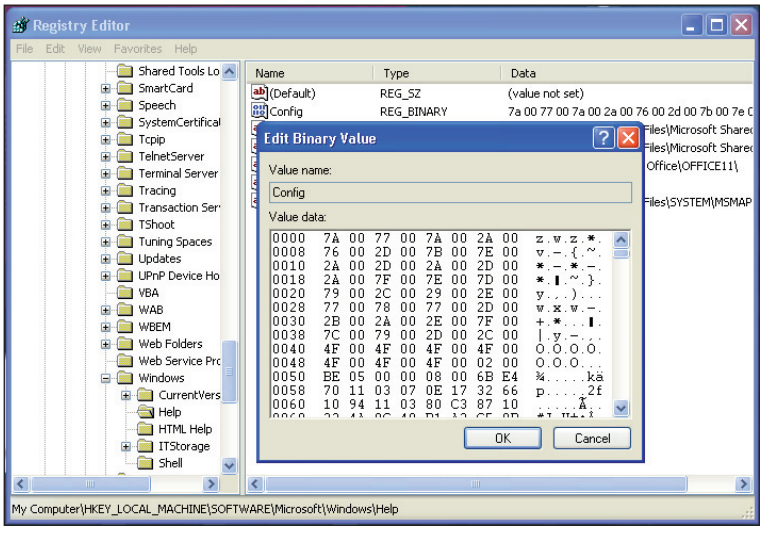

Figure 3: Configuration data stored in the registry.

It drops the following files:

C:\Documents and Settings\All Users\NetWork\Config.dat (encrypted main payload)

C:\Documents and Settings\All Users\NetWork\DDVCtrlLib.dll (clean DLL, needed for science.exe to execute)

C:\Documents and Settings\All Users\NetWork\DDVEC.dll (clean DLL, needed for science.exe to execute)

C:\Documents and Settings\All Users\NetWork\science.exe (clean executable).

In order to survive a reboot, it registers the dropped executable as a service, passing an enormously long command line with three command-line parameters:

```
HKLM\SYSTEM\CurrentControlSet\Services\NetWork
Service       ImagePath:

C:\Documents and Settings\All Users\NetWork\science.
exe LLLLYIIII7QZAkA0D2A00A0kA0D2A12B10B1ABjAX8A1uIN2
uNkXlMQJLePvbUPePJgW59t7kwOKDSPJgg5hh2ZezxFVXJg75xlr
ebuXbtKyWqUXp5FKfZvYPKwpEzTm7xosdLUO7w5zXLnN0dVNKO72
eKLYKJs3ROEucKypdnkgEVP5PgpUPLKRVtLLKT6ELLKw6WxlKQnw
PLKp6u6vYPOr8RUzRnkyHlKRs7LNkpTvzt8w

...{7760 further ASCII characters skipped}...

pW2kOhRD2A00A0kA0D2A12B10B1ABjAX8A1uIN2unkZLk1jLGpdB
Wpwpo73uKTWkwOIdU0iWW5kX0z5zjfTxO7rexlsu2uM2TKxGbejP
5Fn6HVyPXG1Ul4M7XoRtZ5yW2ezXNNxP4VlkO73uilYKhSSR856S
HIsTnkgE6PGpUPUPLKPvtLNkafWllKr 100 PC@
```

The first parameter is intended to cause a buffer overflow in the clean executable and, by invoking a shellcode, run the loader for Config.dat. It should be noted that, of the three parameters passed in the command line, the second bears no relevance, but the last one is of crucial importance.

Not wanting to wait for the next reboot, CreateProcess is called with the same parameter to execute the payload immediately (the final couple of bytes differ; also the second command-line argument, 100, is replaced with 300).

LOADING PROCESS

When all of the required pieces have been installed on the system, the malware deletes the temporary components, and the infected computer is ready for (ab)use. During system start-up, the dropped science.exe file is loaded as a service, with a malicious command line.

Science.exe

File size: 112,064 bytes
SHA1: 6261e967baf09e608e5d5b156a3701339c73fb95
MD5: 0070a38553997de066b2aba8c0574d6f

This is a legitimate, digitally signed clean application (certificate issued to *Jinhua 9158 Network Science and Technology Co. Ltd*), the original name of which is Download.exe. Looking up other files signed by the same certificate, we found a handful of other application installers that dropped similar versions of Download.exe (see Appendix). All of them proved to be vulnerable to the same abuse, but due to reorganization of the code and memory layout, modifications would be needed in order for them to be used in this way.

As shown in Figure 4, the science.exe file is intact, not modified by the malware author.

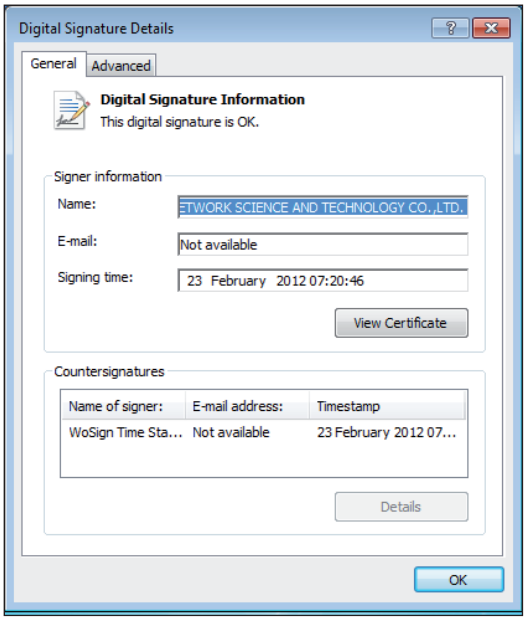

Figure 4: Digital signature checked OK.

The program is executed either via the registry key, or using the CreateProcess API. In both cases the extremely long command line is passed to it. Either way, the long command line causes a stack overflow, and leads to the execution of a piece of shellcode. Although not obvious at first, the shellcode is actually hidden within the command line argument itself.

Crash dumps show that an access violation occurs at virtual offset 404350h in science.exe, which is an interesting coincidence (actually, a lot more than a coincidence), given that the last command-line argument, PC@, is exactly this value in hexadecimal representation.

Looking at the executable in a disassembler, one can observe that at this virtual address there is a POP ECX, RET sequence:

```
.text:00404350 59      pop   ecx
.text:00404351 C3      retn
```

A bit of debugging reveals that, upon reaching this point, the stack contains the command-line parameter address and a zero; the code above pops the zero and transfers execution to the first byte of the command line.

The mechanism of this exploitation is exactly the reason why the *MSDN* library documentation contains warnings such as the following for some of the function references:

> 'Using **vsprintf**, there is no way to limit the number of characters written, which means that code using this function is susceptible to buffer overruns. Use _vsnprintf instead, or call _vscprintf to determine how large a buffer is needed.'

The overflow occurs when the command-line arguments are written out to the log file (Download.log) and vsprintf is used on this buffer without any precaution. This will cause an overflow if the command line is longer than 0x2000 bytes.

```
char *write_log(int a1, char *Format, ...)
{
   va_list va; // [sp+200Ch] [bp+Ch]@1
   char *result; // eax@1
   char Dest; // [sp+0h] [bp-2000h]@2

   va_start(va, Format);
   result = Format;
   if ( Format )
   {
      result = (char *)vsprintf(&Dest, Format, va);
      if ( (unsigned int)result < 0x2000 )
         result = (char *)CLog__ADD_Log(g_Log,
&Dest, result, a1);
   }
   return result;
}
```

The function calls vsprintf to print the argument list into a string buffer allocated with a size of 0x2000 bytes; the format string is the command-line argument, which in our case turns out to be longer than the allocated space for the buffer. As a result, vsprintf will overwrite the return address on the top of the stack.

The command-line argument is filled with junk characters just to make sure that the PC@ at the end will end up at the location at which the return address is stored.

This way, the return at the end of the function:

```
add     esp, 2000h
retn
```

will position the stack pointer to the 0x404350 DWORD on the overwritten stack.

To illustrate this, the top of the stack on the entry of the write_log() function looks like this:

return address
Param 1: log entry ID
Param 2: address of command line

Then, after the stack overrun on the exit of write_log(), the stack will contain:

0x404350
Param 1: log entry ID
Param 2: address of command line

When the execution returns to offset 0x404350, the first value is popped from the stack, leaving only the entry ID and the address of the command line:

Param 1: log entry ID
Param 2: address of command line

At offset 0x404350 in the program, a function epilogue is found:

```
.text:00404350     pop    ecx
.text:00404351     retn
```

This will pop the log entry ID from the top of the stack, and return to the next address found on the stack, which is the address of the command line. Consequently, the execution starts at the first byte of the command-line argument.

I should mention that this is a very simple stack overflow exploitation – a textbook example that was commonly being practised over 10 years ago. Nowadays, secure coding methods make applications a lot harder to break. Nevertheless, the malware writers only needed to find one vulnerable application, and use it for their purpose.

Shellcode from science.exe

At first glance, the command-line parameter looks like a random string:

```
LLLLYIIII7QZAkA0D2A00A0kA0D2A12B10B1ABjAX8A1uIN2uNkXl
MQJLePvbUPePJgW59t7kwOKDSPJgg5hh2ZezxFVXJg75xlrebuXbt
KyWqUXp5FKfZvYPKwpEzTm7xosdLUO7w5zXLnN0dVNKO72eKLYKJs
3ROEucKypdnkgEVP5PgpUPLKRVtLLKT6ELLKw6WxlKQnwPLK...
```

But in fact it is a valid 32-bit *Intel* code, starting with a short decoder, which is followed by the decrypted shellcode. It is very likely that it was created by the unicode_upper encoder of the Metasploit toolkit. This encoder generates a final form where each byte of both the decoder and the decoded content is an alphanumeric character – very suitable if it has to be passed as a command line. However, an important part of the shellcode usually cannot be represented in ASCII bytes. This is the prologue, which is responsible for determining the exact memory position. Without knowing this, it is not possible to decode the main shellcode body.

Normally, the Metasploit decoders begin with a 'get EIP' fragment, similar to this:

```
fabs
fnstenv byte ptr [esi-0Ch]
pop     ebp
```

First, a random floating point instruction is executed, and then the fnstenv instruction is used to get the floating point environment structure. Among many properties, at offset 0x0C this structure contains the EIP of the last executed floating point instruction (fabs, in this case). The structure is aligned 0x0C bytes into the stack, thus the top of the stack will just contain the EIP value, which is later popped into the EBP register. This is a commonly used, portable solution, but has one major disadvantage: the byte code of the floating point instructions contains non-printable characters, thus can't be used in a string command-line parameter.

The shellcode used in the Simbot infection scheme is limited by the fact that it also has to serve as a command-line parameter, and can thus only contain printable characters. This means that it can't contain the usual code to find its own memory offset, but it can make use of the fact that it knows the exact stack layout during the exploitation – thanks to the very controlled environment (i.e. only the specific science.exe, dropped during the installation, has to be exploited).

Simbot's shellcode uses the following 'get EIP' snippet:

```
dec     esp
dec     esp
dec     esp
dec     esp
pop     ecx
```

```
00000000 4C                              dec     esp
00000001 4C                              dec     esp
00000002 4C                              dec     esp
00000003 4C                              dec     esp
00000004                         at this point ECX gets the address of the start of the command line argument
00000004 59                              pop     ecx
00000005 49                              dec     ecx
00000006 49                              dec     ecx
00000007 49                              dec     ecx
00000008 49                              dec     ecx
00000009 37                              aaa
0000000A 51                              push    ecx
0000000B 5A                              pop     edx
0000000C 41                              inc     ecx
0000000D 6B 41 30 44                     imul    eax, [ecx+30h], 44h ; 'D'
00000011 32 41 30                        xor     al, [ecx+30h]
00000014 30 41 30                        xor     [ecx+30h], al
00000017 6B 41 30 44                     imul    eax, [ecx+30h], 44h ; 'D'
0000001B 32 41 31                        xor     al, [ecx+31h]   ; ECX = EDX +1
0000001E 32 42 31                        xor     al, [edx+31h]
00000021 30 42 31                        xor     [edx+31h], al   ; running key decoding, keys are taken from the encoded bytes
00000024 41                              inc     ecx
00000025 42                              inc     edx
00000026 6A 41                           push    41h ; 'A'       ; end marker for decoding
00000028 58                              pop     eax
00000029 38 41 31                        cmp     [ecx+31h], al
0000002C                         the jz offset is already modified by he previous XOR to performs a loop back to address 0C
```

Figure 5: Unicode_upper decoder.

```
                push    ebp
                mov     ebp, esp
                sub     esp, 200h
                mov     dword ptr [ebp-1Ch], 60F43F1Bh ; GetModuleFileNameA
                mov     dword ptr [ebp-18h], 38C62A7Ah ; CreateFileA
                mov     dword ptr [ebp-14h], 0BE25545h ; ReadFile
                mov     dword ptr [ebp-10h], 0C0D6D616h ; CloseHandle
                mov     dword ptr [ebp-0Ch], 9554EFE7h ; GetFileSize
                mov     dword ptr [ebp-8], 0AB16D0AEh ; VirtualAlloc
                mov     dword ptr [ebp-4], 0B562D3DBh ; VirtualFree
                xor     ecx, ecx
                mov     esi, fs:dword_30 ; ESI <- PEB
                mov     esi, [esi+0Ch]  ; PPEB_LDR_DATA
                mov     esi, [esi+1Ch]  ; <- InInitOrderModuleList

next_module:                            ; CODE XREF: seg000:00002455↓j
                mov     eax, [esi+8]    ; <- DllBase
                mov     edi, [esi+20h]  ; <- BaseDllName
                mov     esi, ds:off_0[esi] ; <- next entry
                cmp     [edi+18h], cx   ; check name length - search for KERNEL32.DLL
                jnz     short next_module ; <- DllBase
                mov     ebx, eax
                mov     edx, [ebx+3Ch]
                mov     edx, [edx+ebx+78h] ; <- Export table
                add     edx, ebx
                mov     [ebp-24h], edx
                mov     esi, [edx+20h]  ; <- Export table: Address of names
                add     esi, ebx
                mov     ecx, [edx+18h]  ; <- Export table: number of names
                xor     edx, edx
                mov     [ebp-28h], edx

next_function:                          ; CODE XREF: seg000:000024C5↓j
```

Figure 6: A more or less traditional piece of shellcode is found.

The advantage is obvious: all of these instructions are represented by printable characters. But the exploitation must be very strict; this prologue requires the stack pointer to be controlled to an exact value. In the previous section we saw that this is the case – the stack pointer is well known by the time the execution reaches this point.

The code was reached via a RET instruction from science.exe, therefore decreasing the stack pointer by four will position it back to the memory address of the command line, which coincides with the start of the shellcode, the two being the same.

The first part of the shellcode is the unicode_upper decoder, which performs a single-byte XOR decryption, the key value being modified in each loop.

After the decoding, a more or less traditional piece of shellcode is found.

The API resolver code (a combination of shift left by three

bytes, and a bytewise XOR of the last byte of the checksum with the actual character of the name) is unusual, and has not yet been seen in other samples.

The shellcode reads the content of Config.dat (the main payload) from the folder from which the exploited science.exe was executed, and decrypts it.

The decryption has two layers: first is a single-byte XOR, the key being the first byte of the file; the second is a running-key single-byte XOR, which starts with 1, and is incremented in each loop.

Finally, it executes the decrypted content.

Memory loader

The decrypted Config.dat contains the embedded main payload, which starts at offset 0xc13, and a loader code.

The loader code, executed by the shellcode, does the necessary housekeeping to transfer this embedded data (which is actually a *Windows* PE executable) to an executable memory image: it fixes the section permissions, resolves the imports, and performs the necessary relocations. This way, the payload can be decoded and executed without hitting the hard disk (and without giving on-access anti-virus products the chance to check and detect it).

MAIN PAYLOAD

The final payload is a *Windows* DLL with an obfuscated entry code, using a couple of redirections before reaching the DllMain function, which itself is also obfuscated to make tracing more complicated.

It contains yet another encrypted PE loader code and a large, 0x18A00-byte-long encrypted embedded DLL which is packed using the zlib algorithm and dropped as instsrv.dat in the %TEMP% directory. This loader is very similar to the loader of Config.dat. It serves as a back-up loader (in case the execution runs into access restrictions), which checks the OS version: if it is 5.2 (*Windows 8*), then it injects the loader code into the explorer.exe process; if it is anything else, it injects the loader code into dwm.exe.

The injected code uses a UAC bypassing technique that is very similar to [2]. Using this, it executes the instsrv.dat file dropped in the %TEMP% directory.

Instsrv.dat is a PE executable that first adds %ALLUSERSPROFILE%\NetWork\science.exe to the DEP exclusion list by invoking the NoExecuteAddFileOptOutList export of the sysdm.cpl applet, passing the path name as a parameter. After that, it terminates the science.exe process, deletes the NetworkService service, and registers science.exe with the exploiting buffer as a service again. Finally, it restarts the service.

Now back to the final payload.

It connects to 59.188.23.121 (which is a dial-up IP located in Hong Kong) on ports 8001 and 8433.

It loads configuration data from the registry key HKLM\SOFTWARE\Microsoft\Windows\Help -> Config (two-byte XOR with key 0x004f). Alternatively, if the key for some reason cannot be created in the dropping process, it reads from the file %ALLUSERSPROFILE%\NetWork\t1.dat. The decoded content has the value 585e9b41ebebe0126cfa878bdea036bc.

This is the encoded form of the C&C IP address. Interestingly, the trojan does not decrypt it, rather it is later brute-forced to match the IP – all possible IP address strings are generated and tested. The IP addresses (two of them, both the same) are decoded character-by-character.

Given the complexity of the installation and the loading process, the backdoor component has disappointingly little functionality: once the connection is established, it sends and receives data. The data is BASE64-encoded and zlib compressed (version 1.2.3 code is compiled into the code), it is decompressed in memory, and executed. An uncompressed PE executable in the network traffic would be too obvious a sign of suspicious activity, hence the compression.

So the result of all the efforts described here is 'only' to open a channel to the infected computer and facilitate the execution of arbitrary code.

At the time of writing this article, we are not aware of the components that are pushed to the infected computer, but it would be safe to say that the usual data-stealing and remote access components are the most likely candidates.

CONCLUSION

It is common in APT-related attack scenarios for an application vulnerability (usually in one of the *MS Office* suite) to be used to breach a system and infect it. The unique feature in Simbot is that an additional exploitation is utilized, this time to hide the presence of the malware on the infected system, and persist after the system restarts.

This malware does not rely on a preinstalled application for infection, rather it carries and drops the target itself – a very convenient approach to ensure that the system contains a vulnerable version of the application in question. Even if the vulnerable application is fixed by the vendor, and the fix is distributed to all users, this will not affect the malware:

as long as the malware authors have a single vulnerable version, no matter how old, they can bundle it with the installation package, and drop it onto the system. As mentioned previously, this malware does not take anything for granted, carrying all the necessary components (both malicious and clean) itself.

Ironically, the original purpose of science.exe, as its developer intended, was to download executable updates. Indeed, the Simbot backdoor makes use of this downloader application to download executable updates, but not by using the natural functionality of the downloader, rather by exploiting its logging function to load and execute a binary payload that, after some twists and turns, does the downloading itself.

After a successful infection we will find the following on the system:

- A clean signed application registered for start-up
- Two clean DLL files needed for the execution of the clean executable
- An encrypted payload file
- A registry subkey that contains an encrypted shellcode.

This is not very much on which to base a reliable detection. And this is a functioning backdoor infection – I can't think of a case with a less detectable fingerprint on the infected system.

REFERENCES

[1] Szappanos, G. Needle in a haystack. Virus Bulletin, February 2014, p.19. http://www.virusbtn.com/pdf/magazine/2014/201402.pdf.

[2] Windows 7 UAC whitelist: Proof-of-concept source code. http://www.pretentiousname.com/misc/W7E_Source/Win7Elevate_Inject.cpp.html.

APPENDIX: FILES WITH THE SAME CERTIFICATE

The clean science.exe application was signed by 'JINHUA 9158 NETWORK SCIENCE AND TECHNOLOGY CO., LTD.' This company is tied to the website 9158.com, which is registered to mikefu@t2t2.com, to the organization Jinhua 9158Network Science and Technology Co., Ltd, in Hangzhou, China.

We were able to identify a number of further files in our collection that use a certificate from the same issuer; all of them were clean installers. Some of them (the 9158 KTV installers) dropped Download.exe as a component. Clearly, this application would be the source of the exploited binary.

4d2f9aac4408237a56dadb89e256e637a703b4ee: 9158 Virtual Camera installer – looks legitimate

4d64bb02d287f2f4e3707f8f7c64a92fbe6621b5: 9158 KTV installer (a version of Download.exe is installed) – looks legitimate

4f1e67bfe5c2698698f7abffbfa740507aaaeb49 : CHOUZHOUGame (an add-on of some sort, not a standalone application) – looks legitimate

878f09552e7277544f6b3702e310757c0bde1b42: DuoDuoVideoGame installer (a version of Download.exe is installed) – looks legitimate

9e7cb141eb97e4a83946b3494344b55bbbf0691a: 9158 KTV installer (a version of Download.exe is installed) – looks legitimate

a8fb2fa2d1fdbeb45831c3ba08d6d73cd08cb44b: 9158 KTV installer (a version of Download.exe is installed – same as with 9e7cb141eb97e4a83946b3494344b55bbbf0691a) – looks legitimate

f1dae1ee4ece2d5e30b199663f721a3718a661b9: XinGuang installer – looks legitimate

Altogether, four different versions of Download.exe were found (including the one carried by the malware). Differentiating between them was made difficult by the fact that all versions had exactly the same version information, as seen on the following output of the *Sysinternals* sigcheck tool:

```
Verified:       Signed
Signing date:   07:20 23/02/2012
Publisher:
Description:    DownLoad Microsoft ???????
Product:        DownLoad ????
Version:        1, 0, 0, 1
File version:   10, 3, 19, 1
```

Testing the other versions of Download.exe (replacing science.exe right before the CreateProcessA) caused a crash and a debug dialog pop-up. All of these variations were vulnerable to the exploitation, with the same bogus write_log() function, but due to reorganization of the code in the development process, the 0x404350 address, where the execution is re-routed does not contain the required POP-RET instruction sequence. Fixing the return value could make these variants vulnerable to full exploitation as well.

SPOTLIGHT

GREETZ FROM ACADEME: CENSORED

John Aycock
University of Calgary, Canada

From time to time, I visit schools to give outreach talks about computer security. One question I always ask the students is how they know that what they see in their web browser is the same information as was sent from the web server. It's easy to forget that for many – possibly most – users, the Internet is a magical thing. For regular users, *of course* the information they receive comes straight from the source; why would they suspect that what they see has been broken apart like countless *Lego* bricks, strewn across multiple devices and systems, and reconstructed seamlessly for their pleasure?

The reality is different, of course, and even if both the user's machine and the server(s) providing content are assumed to be uncompromised, there are many points along the way at which content changes can occur. As security professionals, we might be inclined to think first of man-in-the-middle attacks, and while that is a possibility, it is just one of several. What makes some other content-changing scenarios interesting is that they are legitimate (for certain values of the word 'legitimate'), and are not any easier to detect despite their legitimacy.

A case in point: companies market products to Internet service providers, providing 'in-browser messaging' for a variety of purposes [1]. The corresponding patents are much more detailed, and specifically say 'content may be modified or replaced along the path to the user' [2].

Back in 2008, Reis *et al.* took a crack at detecting content modification by using what they called 'web tripwires' [3]. Their idea was for a server to provide content as usual, but to include a script to run in the browser that would compare the content received with some 'ground truth' version of the same content. As they tested the web tripwires, they were indeed able to spot some content modifications made *en route*, as well as tripping across some instances where ad-blocking software had thoughtfully injected exploitable vulnerabilities into the content.

Another 'legitimate' content modification scenario is censorship. (Done *only* to protect the children / catch terrorists / facilitate a stable and moral society, you understand; pick your favourite.) There are more than enough examples of censorship to go around – the OpenNet Initiative's reports [4] are an informative, if somewhat depressing, place to start reading – but the question is how such content modifications might be detected.

As it happens, December is ACSAC season, i.e. the Annual Computer Security Applications Conference, and a paper included in the proceedings of the 2013 event looked at exactly this problem. Wilberding *et al.*'s 'Validating web content with Senser' [5] works '*even when SSL/TLS is not supported by the web server*' – a fact I mention because they thought to include that exact phrase in the paper's abstract and in the introduction, italicized in both places. It has excellent Internet meme potential, I think, e.g. 'Cleans your dirty dishes… even when SSL/TLS is not supported by the web server.' When the meme goes viral, just remember you heard it in *VB* first. But I digress.

What Senser does is build a 'consensus' view of the content of a web page by querying a number of proxies distributed around the Internet. The premise is that, as long as the majority of the proxies receive an unfettered view of the web page, a version of it can be reconstructed in the browser. There are a number of practical problems, in that localized or customized content may be delivered even in the absence of censorship, but also the fact that there may be 'AS-level adversaries who control large segments of the network and may attempt to manipulate web content' [5, p.340]. Not to name names. Senser devotes considerable effort to diversifying the AS-level paths between the proxies and the server with the content, in an attempt to route around this problem.

An underlying issue I have with systems like Senser (and also *Tor*) when it comes to censorship, is that meta-data reveals many potentially compromising things. Depending on where a user resides, the simple fact (meta-meta-data, almost) that they used Senser or *Tor* is suspicious. Until these systems are shipped, enabled by default, in *Windows*, nothing will change that… and that's not a happy *Disney* ending to reach out to school children with.

REFERENCES

[1] PerfTech. Solutions. http://www.perftech.com/solutions.html, last accessed 18 February 2014.

[2] Donzis, H. M.; Donzis, L. T.; Frey, R. D.; Murphy, J. A.; Schmidt, J. E. Internet connection user communications system. U.S. Patent 8,108,524, 31 January 2012.

[3] Reis, C.; Gribble, S. D.; Kohno, T.; Weaver, N. C. Detecting in-flight page changes with web tripwires. 5th USENIX Symposium on Networked Systems Design and Implementation, 2008, pp.31–44.

[4] OpenNet Initiative. https://opennet.net/, last accessed 18 February 2014.

[5] Wilberding, J.; Yates, A.; Sherr, M.; Zhou, M. Validating web content with Senser. 29th Annual Computer Security Applications Conference, 2013, pp.339–348.

END NOTES & NEWS

The Commonwealth Telecommunications Organisation's 5th Cybersecurity Forum takes place 5–7 March 2014 in London, UK. For more information see http://www.cto.int/events/upcoming-events/cybersecurity-2014/.

European Smart Grid Cyber and SCADA Security will take place 10–11 March in London, UK. For more information see http://www.smi-online.co.uk/2014cybergrids31.asp.

Cyber Intelligence Asia 2014 takes place 11–14 March 2014 in Singapore. For full details see http://www.intelligence-sec.com/events/cyber-intelligence-asia-2014.

ComSec 2014 takes place 18–20 March 2014 in Kuala Lumpur, Malaysia. For details see http://sdiwc.net/conferences/2014/comsec2014/.

The Future of Cyber Security 2014 takes place 20 March 2014 in London, UK. For booking and programme details see http://www.cyber2014.psbeevents.co.uk/.

Black Hat Asia takes place 25–28 March 2014 in Singapore. For details see http://www.blackhat.com/.

Information Security by ISNR takes place 1–3 April 2014 in Abu Dhabi, UAE. For details see http://www.isnrabudhabi.com/.

SOURCE Boston will be held 9–10 April 2014 in Boston, MA, USA. For more details see http://www.sourceconference.com/boston/.

Counter Terror Expo takes place 29–30 April 2014 in London, UK. The programme includes a cyber terrorism conference on 30 April; the event is co-located with Forensics Europe Expo. For details see http://www.counterterrorexpo.com/.

The Infosecurity Europe 2014 exhibition and conference will be held 29 April to 1 May 2014 in London, UK. For details see http://www.infosec.co.uk/.

AusCERT2014 takes place 12–16 May 2014 in Gold Coast, Australia. For details see http://conference.auscert.org.au/.

The 15th annual National Information Security Conference (NISC) will take place 14–16 May 2014 in Glasgow, Scotland. For information see http://www.sapphire.net/nisc-2014/.

CARO 2014 will take place 15–16 May 2014 in Melbourne, FL, USA. For more information see http://2014.caro.org/.

SOURCE Dublin will be held 22–23 May 2014 in Dublin, Ireland. For more details see http://www.sourceconference.com/dublin/.

Oil and Gas Cybersecurity takes place 3–4 June 2014 in Oslo, Norway. For details see http://www.smi-online.co.uk/energy/europe/conference/Oil-and-Gas-Cyber-Security-Nordics.

The 26th Annual FIRST Conference on Computer Security Incident Handling will be held 22–27 June 2014 in Boston, MA, USA. For details see http://www.first.org/conference/2014.

Hack in Paris takes place 23–27 June 2014 in Paris, France. For information see http://www.hackinparis.com/.

Black Hat USA takes place 2–7 August 2014 in Las Vegas, NV, USA. For details see http://www.blackhat.com/.

VB2014 will take place 24–26 September 2014 in Seattle, WA, USA. For more information see http://www.virusbtn.com/conference/vb2014/. For details of sponsorship opportunities and any other queries please contact conference@virusbtn.com.

SUBSCRIPTION RATES

Subscription price for Virus Bulletin magazine (including comparative reviews) for one year (12 issues):

- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000
- *Bona fide* charities and educational institutions: $175
- Public libraries and government organizations: $500

Corporate rates include a licence for intranet publication.

Subscription price for Virus Bulletin comparative reviews only for one year (6 VBSpam and 6 VB100 reviews):

- Comparative subscription: $100

See http://www.virusbtn.com/virusbulletin/subscriptions/ for subscription terms and conditions.

Editorial enquiries, subscription enquiries, orders and payments:

Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: http://www.virusbtn.com/