Scan of the Month: Scan 32

Lutz Schildt
ls@mcb-bremerhaven.de
September 20, 2004

Contents

1. Introduction

This month's challenge was to analyze a home-made malwary binary, in an effort to reinforce the value of reverse engineering malware, and improve (by learning from the security community) the methods, tools and procedures used to do it.

These are the tools I used:

IDA Pro 4.05 by DataRescue	http://www.datarescue.com
OllyDbg 1.10 by Oleh Yuschuk	http://home.t-online.de/home/Ollydbg
Hacker's View 6.40 by SEN, Kemerovo	unfortunately I do not have an URL for it

IDA Pro 4.05 is an Interactive Disassembler for pretty much most architectures and executable formats with lots of nice features for Reverse Engineering Files one does not have the sourcecode of.

OllyDbg 1.10 is a Win32 Debugger.

Hacker's View 6.40 is a little outdated Version of an excellent HexEditor which is capable of showing the opcodes corresponding to binary data.

2. Analysis

After downloading the binary, I verified the md5sum. I then started by taking a look at the binary itself using Hiew to get a brief overview of what I'm dealing with. The first step was to check the Section Table which shows 3 sections:

Section 1:JDR0 is 44kByte in size, none of the data is present in the binary itself.
Section 2:JDR1 is 16kByte in size, which is completely present in the binary itself.
Section 3:.rsrc is a 4 kByte standard section, containing resources like the icon, some version information and the pretty small imports table.

Taking a look at the code portion starting at the entrypoint in Section 2, one can pretty much figure out that Section 2 consists of compressed data and the needed decompression code. The section names are not used in any common pe-compression-tools which means the compression/decompression engine is likely to be home-made aswell.

To analyse the binary using IDA I first needed to rebuild the missing parts of the original binary: the real codesection and import section/table. As the file has the decompression algorithm built in and I did want to safe some time, I used OllyDbg and loaded the binary. Before letting the debugger execute any of the code of the binary I took a closer look at the decompression code.

The algorithm starts at the entrypoint at 0x40FD20 and ends at 0x40FE78 with a jump to the real entry point. The decompression code itself is rather simple and not using any obfuscating techniques to "fool" disassemblers/debuggers. It consists of 2 parts. Part 1 decompresses the compressed data and writes it into Section JDR0 starting at 0x401000. At 0x40FE26 decompression is finished and

Part 2 of the loader begins. Part 2 is a small portion of code building the import table for the real code at 0x401000.

I stopped at this point, to prepare a copy of the original executable. I added a fourth Section to the copy which was going to be used to store a rebuilt import section. Adding a new Section, I also had to adjust a bit of the PE-Header. I started OllyDbg again, this time using the slightly modified Version of the binary. The decompression portion of the loader was just as I needed it, the part building the import table is what I needed to be a bit different. As I didn't want to run the malware code in the JDR0 section, I replaced the the jump to the real entrypoint with a Call to ExitProcess to stop the process when I was finished. I set a breakpoint at the end of the decompression part so I could check out part 2 with real data which would make it easier. I let the process run until the breakpoint which produces the missing codesection. Now I analysed part 2 and rewrote it's code in the paused process to generate a complete import section in my previously added fourth section. I dumped codesection and import section into 2 files.

Using the two dumps I was able to build a third version of the binary which only consisted of uncompressed codesection, resourcesection and importsection. The datasection normally found in PE-Files was merged into the codesection by the selfmade pe-compressor of the author which is not a problem at all. Using IDA and the third rebuilt uncompressed version of the binary I could start analysing the purpose and features of the malware allowing me to answer the questions of the challenge.

2.1. Identify and provide an overview of the binary, including the fundamental pieces of information that would help in identifying the same specimen.

The binary is compressed using a home-made pe-compressor. It consists of 1 empty section (JDR0), 1 section containing the loader and compressed code and datasections (JDR1) and 1 resourcesection containing icon, versioninformation and a small importtable (.rsrc). The way the compressor created the binary and the loader decompresses it, this technique could corrupt some executables the compressor/loader are used with as a part of the datasection will still contain parts of compressed data after when the jump to the real entrypoint is executed. The original binary was coded and compiled using Microsoft Visual Basic. Binaries coded by the same author could most likely be identified using the sectionnames, the decompression code and another small portion of the PE-Header containing the texts "9.99" and "JDR!", given the author doesn't change those.

2.2. Identify and explain the purpose of the binary.

The binary enables the author to automatically update and control an unlimited number of infected clients using a prepared webserver. The malware is started on system startup and from then on queries the website once every 60 seconds by default.

2.3. Identify and explain the different features of the binary. What are its capabilities?

Download:	The binary is able to automatically download files from the configured website
Execution:	The binary able to execute any file on the client, most likely those previously downloaded.
Upload:	The binary can upload any file to the webserver posting Multipart-Formdata.
Screenshots:	The binary is able to take screenshots.
Sleep:	Tells the client to sleep for x seconds before executing the next command

The client can be completely customized using commandline parameters to change any default settings:

--server	sets the main URL of the Website to be queried
--commands	sets the name of the document to retrieve which will contain commands to execute
--cgipath	sets the path to the cgi-files for upload/download on the webserver
--cgiget	sets the name of the cgi-file used for uploading files to the webserver
--cgiput	sets the name of the cgi-file used for download files from the webserver
--tmpdir	sets the path of the temporary directory on the client for RaDa
--period	sets the period of time to wait in seconds before the commandscript is retrieved and executed the again
--cycles	sets the number of times the script should be retrieved from the webserver and then just exit
--help	displays syntax information for the binary
--gui	enables the built in GUI with buttons for Install, Uninstall, Configuration, Usage, Retrieve commandscript and exit
--installdir	Sets the path where to copy the binary on installation
--noinstall	Starts the binary without adding it to HKLM/Software/Microsoft/Windows/CurrentVersion/Run
--uninstall	Removes the binary and turns off autorun on startup
--authors	explained in detail when answering the Bonus Questions

2.4. Identify and explain the binary communication methods. Develop a Snort signature to detect this type of malware being as generic as possible, so other similar specimens could be detected, but avoiding at the same time a high false positives rate signature.

The binary uses the HTTP protocol to communicate with a webserver, which sends back a html document with html forms containing fields. The field names contain the commands to execute and the field values contain the parameters needed. The binary in this version understands 5 commands:

exe	executes the command given as parameter example: <input name=”exe” value=”calc”> would start the windows calculator
get	downloads the file given as parameter from the webserver example: <input name=”get” value=”foo.baa”> would download the file foo.baa from the the webserver posting the filename in multipart-formdata to download.cgi
put	uploads the file given as parameter to the webserver posting it in multipart-formdata to upload.cgi
screenshot	takes a screenshot and safes it to the temporary directory of RaDa in a file given as parameter
sleep	waits for the number of seconds given as parameter

Here are 5 snort signatures to detect probable commands issued using html forms

```
# name=”get”
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RaDa probable
download command"; content:"|6E 61 6D 65 3D 22 67 65 74 22|”; nocase;
flow:to_client,established;classtype:trojan-activity;)

# name=”put”
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RaDa probable
upload command"; content:"|6E 61 6D 65 3D 22 70 75 74 22|”; nocase;
flow:to_client,established;classtype:trojan-activity;)

# name=”sleep”
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RaDa probable sleep
command"; content:"|6E 61 6D 65 3D 22 73 6C 65 65 70 22|”; nocase;
flow:to_client,established;classtype:trojan-activity;)

# name=”screenshot”
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RaDa probable
screenshot command"; content:"|6E 61 6D 65 3D 22 73 63 72 65 65 6E 73 68 6F
74 22|”; nocase; flow:to_client,established;classtype:trojan-activity;)
```

```
# name=”exe”
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RaDa probable
execute command"; content:"|6E 61 6D 65 3D 22 65 78 65 22|”; nocase;
flow:to_client,established;classtype:trojan-activity;)
```

2.5. Identify and explain any techniques in the binary that protect it from being analyzed or reverse engineered.

The only technique of real protection against reverse engineering used is the compression of the code and data sections. If someone wanted to reverse engineer a compressed binary file, he would need to rebuild the original binary or at least a valid fully functional one that enables that someone to use a disassembler like IDA. You need to have the code and data sections and the import section working in case you want to use tools like IDA. In order to achieve this you nearly always need to invest quite some amount of time to rebuild the binary. Especially in this case where the author did not use one of the common tools like UPX, PElite etc. for compressing the binary and the reverse engineer has to rebuilt the binary manually.

Why the author used Visual Basic to code RaDa might have been because it is mostly a lot easier to code, and harder to analyse/reverse engineer the binary after compilation. Many people almost already give up when they see that the binary was created using Visual Basic. But I would not really call that a technique of protection against reverse engineering. It might stop a few people, but those that need to analyse the binary will probably not care if it is Visual Basic or not.

2.6. Categorize this type of malware (virus, worm...) and justify your reasoning.

I would categorize this as a Trojan. It does not spread in any way nor does it infect files or does any damage on it’s own. It can only be used after infection either through a worm, or mail virus etc. In this version its only use can be to download and install anything else the author needs.

2.7. Identify another tool that has demonstrated similar functionality in the past.

There are a lot of Trojans/Backdoors which can be remotely controlled using IRC-Channels, download other binaries and execute them or upload data/files to webservers. But this binary is the first one that I can remember that combines all of this using just HTTP traffic.

2.8. Suggest detection and protection methods to fight against the threats introduced by this binary.

In my opinion there are not really any new threats introduced by this binary. Downloading and installing “extensions” or updates is a common feature amongst rootkits for example, they all do have scripts that once a host has been compromised, download and install other tools needed. Sending files/data, taking commands etc. is also a common feature amongst RATs like IRCBots, Trojans,

Backdoors and the like. There is not much one can do to detect and protect against such tools, except for doing what one should always be doing. Stay uptodate with software updates, service packs and virus definitions, be suspicious about any email attachement received and not to just run them because it's attached and the mail has an interesting text telling one to do that. The Solution is in the binary itself "Security through obscurity is the key." There is no protection against a tool like this if it comes as an attachement and the receiver just doubleclicks on it without being suspicious why someone he doesn't even know sent him an email asking if he was the one that sent this attachement to him.

2.9. Bonus Questions

2.9.1. Is it possible to interrogate the binary about the person(s) who developed this tool? In what circumstances and under which conditions?

It is possible to get information about the author(s). It is possible to add the parameter "--authors" to the commandline. In that case the binary will check if the os it is running on is inside a VMware session by checking the MAC-Addresses of the Interfaces for VMnet specific MAC-Addresses and for the presence of a registry key belonging to VMware tools which are installed on a windows installation inside VMware. If it finds it is a real Windows installation it is running on, it will display a message box with the following content:

"Authors: Raul Siles & David Perez, 2004".

If not it will display "Unknown argument: --authors" and open an Internet Exlorer Window containing RaDa Usage Information.

2.9.2. What advancements in tools with similar purposes can we expect in the near future?

As I have already said, there are already a lot of other tools that are much more powerful than this binary, with features that include selfpropagation, processhiding and a lot more than 5 commands to be issued remotely. The only thing I can imagine that would be a real advancement for such tools would be polymorphic pe-encryption code which would make it nearly impossible to detect the malware in any way because every version of the binary would look completely different.