VOLATILITYBOT: MALICIOUS CODE EXTRACTION MADE BY AND FOR SECURITY RESEARCHERS

Martin G. Korman
IBM Trusteer, Israel

Email martink@il.ibm.com

ABSTRACT

Part of the work security researchers have to go through when they study new malware or wish to analyse suspicious executables is to extract the binary file and all the different satellite injections and strings decrypted during the malware's execution.

Usually, this initial process is done manually, and it can be lengthy or even end up incomprehensible, in case some actions the malware has taken are not traced back to it.

Enter VolatilityBot. This is a tool I have developed myself, leveraging the Volatility Framework. This new automation tool for researchers cuts all the guesswork and manual tasks out of the binary extraction phase. Not only does it automatically extract the executable (exe), but it also fetches all new processes created in memory, code injections, strings, IP addresses and so on.

Beyond the obvious value of having a complete extraction automated and produced in under a minute, VolatilityBot is highly effective against a wide variety of malware codes and their respective load techniques. It can take on complex malware including banking trojans such as ZeuS, Ramnit, and Dyre, just as easily as it extracts payloads from downloaders such as Upatre and Pony, or even from targeted malware like Havex.

Once VolatilityBot has finished the extraction, it can further automate repair or prepare the extracted elements for the next step in analysis – for example, by fixing the Portable Executable (PE), preparing for static analysis via tools like *IDA*, performing a YARA scan, etc.

The Volatility Framework at the core of this automation tool is an open-source framework for memory analysis and forensics; it analyses the runtime state of a system using the data found in volatile storage (RAM). You can find out more about Volatility at http://www.volatilityfoundation.org/.

WHAT CAN VOLATILITYBOT DO?

VolatilityBot is an automatic modular framework that extracts malicious code from packed binaries, leveraging the functionality of the Volatility Framework. As such, VolatilityBot can dump all malicious injected code, loaded kernel modules and new processes created, from memory.

VolatilityBot is made up of four major components:

1. The manager

This is the core of the VolatilityBot tool. The manager executes the automatic extraction as well as the post-processing modules. This module also controls the associated machines' activity to streamline the workflow.

2. Machines module

This is an abstract design of a research machine; it contains five functions: Revert, Start, Suspend, Clean-up and Get Memory Path.

Each machine has a very small python agent running, which listens and waits for a malware sample. When a sample is sent, it executes it by double-clicking. The agent is of minimal size in order not to affect the behaviour of the malware in the machine. The agent does not perform any API hooking and does not control the machine in any form besides executing the malware.

The machines are controlled and monitored by the manager component, which knows not to send more samples to a machine if it has fatal errors and will mark the machine as unavailable.

3. Code extractor

The code extractor component is a number of modules grouped together for the purpose of extracting all the different malicious code components from the memory. These are separate for code injections, new processes, etc. This component is modular, which allows researchers to write new code for other extractors they may need.

The existing modules for this component are as follows:

- injected_code: uses the Volatility 'malfind' plug-in to find suspect memory areas. After dumping them it tries to determine if the section contains a valid PE or valid shell code. If it finds a valid PE, it fixes the PE header. Either way, it will extract strings and execute YARA on the section dumped from memory.
- module_scan: uses the Volatility 'modscan' plug-in to detect and dump newly loaded kernel modules.
- create_process_dump: uses the Volatility 'procdump' and 'pslist' plug-ins to dump new processes created by the malware. It executes YARA and extracts malware strings.
- create_process_dump_as: uses the Volatility 'dump_as' plug-in to dump address space. It executes YARA and extracts malware strings.
- Hooks: extract API hooks made by the malware (both user-mode and kernel-mode).

4. Post-processing modules

The post-processing modules kick in once the extraction is complete. They are tasked with automated actions like fixing the PE, or availing the resulting elements to static analysis, YARA scans, strings and IP address logging, etc.

These modules can help with:

- Executing the configured YARA rules on post-extraction input as defined by the researcher.
- Extracting strings from the input defined. Other modules can be layered on top in order to extract IP addresses, URLs, etc.
- Producing a report for static analysis with basic PE analysis of the input file.

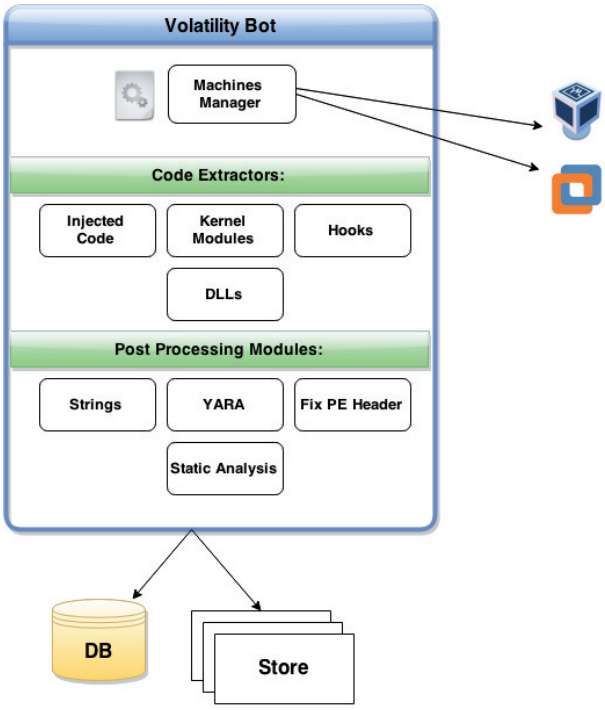

Figure 1: VolatilityBot high level design.

THE VOLATILITYBOT WORKFLOW

The flow of events follows the previous section's numbering, starting with the manager and ending with post-processing.

The manager (.py) module is executed first, alongside a folder containing binaries or a single file set as a parameter. The manager proceeds to build a queue of all the samples to process and adds them to a database. The manager module then locates an idle research machine/VM that can carry out the next step, and sends the file over to it.

Next, the agent on the research machine/VM executes the file and 'sleeps' for a predefined length of time that can be set by the researcher as per his or her preferences. The machine is then put in a suspended state.

In the third step, all configured code extractor modules are executed. Memory dumps are stored, and metadata is saved in a designated database.

Finally, once the extraction phase has completed, all the configured post-processing modules are executed.

The manager is multi-threaded and is configured to take advantage of all machines at the same time. Each sample processing on a machine has its own thread, which is terminated once the processing of the sample has finished. Processing of a sample refers to both the machine executing the sample and to all the code-extraction and post-processing modules after the machine is suspended.

CORE CAPABILITIES

Some of the core capabilities of this automation tool were designed to make it as scalable as possible and easy to work with over time:

- VolatilityBot's manager can handle an unlimited number of machines at once, all depending on the performance of the researcher's equipment.
- Automatic 'golden image' generation is provided in order to ease the process of creating all 'golden image' data. Just create your configuration file and execute the script.
- To simplify scalability, any database supported by SQLalchemy can be set or replaced as needed. The Bot-Excavator's backend is based on SQLalchemy and saves data to an SQLite database.
- Memory dumps and data from all configured post-processing modules are saved to a predefined storage directory.
- Tags can be added to each execution of the script for quick visual reference to information you may need later.
  - Dynamic tags can be added to post-processing modules. For example, malware that loads a kernel-mode driver can be tagged with 'loads_kmd'.

In terms of additional modules that may be deemed necessary in the future, researchers working with VolatilityBot will find that its modular structure makes it very easy to write additional modules, whether code-extractor modules or post-processing ones.

EFFICACY TESTING

Hundreds of samples have been tested in VolatilityBot. I used a research environment comprised of five *Windows XP* (x86) machines (*VMware*-powered). Each sample was executed for exactly a minute and a half.

A couple of malware subsets were created and run through VolatilityBot:

VirusShare subset

I took two subsets from the latest VirusShare archive and ran all of them through VolatilityBot. Table 1 shows some statistical information regarding the results.

VirusShare Subset I	
Total samples	1,750
Samples with at least one successful dump	1,658
Injected code extractions	256
New processes dumped	1,637
Kernel modules dumped	72
Total storage used	3.72 GB
VirusShare Subset II	
Total samples	2,125
Samples with at least one successful dump	1,737
Injected code extractions	736
New processes dumped	1,726
Kernel modules dumped	47
Total storage used	4.7 GB

Table 1: Statistical information regarding the results of running samples from VirusShare through VolatilityBot.

Note that not all of the samples referred to in Table 1 inject code or load a kernel driver. Some of them are just installers of potentially unwanted programs and some of them might be corrupted executables.

I noticed a high success rate in general, for all samples in the subset. Success is defined by the ability to extract injected code, a kernel module or dump of a process.

Malware families' zoo

Another test was carried out that was more malware family and category-oriented. Table 2 shows the statistical information regarding these results.

In this subset we see an even higher success rate. The tested samples were malware of several different categories: downloaders, financial malware and targeted malware. The analysis output provided indicators such as strings and YARA

Malware families' zoo	
Total samples	68
Samples with at least one successful dump	63
Injected code extractions	41
New processes dumped	31
Kernel modules dumped	4
Total storage used	~200 MB

Table 2: Statistical information regarding the results of running samples from a malware familites zoo through VolatilityBot.

ID	Timestamp	Hashes	Source	process_name	Additional Information
10017	2015-05-31 13:45:24.931254	b342f99823809a8ce19155e37ec50c5e 0404101b883189eacefa9d15d8a815ca0f94a6349e06013c8d595e710f6eb013	kmd	okpggn.sys	Details
10016	2015-05-31 13:45:20.562806	1ba87b2eb87ecd8c82c98c6d1abe573b 6575293ea5f3a1ed0fe2bc4d62cfbf4abcd902503749a8591c7b80a807937fab	kmd	ipfltdrv.sys	Details
10015	2015-05-31 13:45:13.175521	a8cb27b7fc39afb3752434fd2239408c 863667bf47766431b89b5e7f28b110139cfebdf207a3e35c329b0173e007eda6	new_process	dcj.exe	Details
10014	2015-05-31 13:45:04.701508	614a1b2d1e5b3ed3de995dd3f7539416 228c9a936613cfc7df189b746ab76969da18056c5b053cdba453437ab4fe0e46	injected	explorer.exe	Details

Figure 2: A sample that loaded a kernel driver.

ID	Timestamp	Hashes	Source	process_name	Additional Information
10044	2015-06-01 15:50:38.002673	493db11ba436f843d6e9f756708a99ec c1cd9fb273f19b7555f96d558425e57ea0cf19fb393298c67d85d2ddff7ef66d	injected	svchost.exe	Details
10043	2015-06-01 15:50:18.204467	98c9e6ad6e627f80acf6b8c0db88cbf2 49584f40a3d8d94abd6b4bf3fe3edf16ba78a8bd0dbf0b99e9ab2ac8dc6fc410	injected	firefox.exe	Details
10042	2015-06-01 15:49:41.210865	163914433f479005c2bc3ecae612d82a 1d298dd8fbeba791c9c672a5a68a9de922bd27780be693ccbbc2fa130b97cb57	injected	IEXPLORE.EXE	Details
10041	2015-06-01 15:49:22.658139	6b9af453620b6f9d85ace35fb322d886 3979454bb3f2077bbd30795053907723e2a85d9f728f1c79aa7325e63644e431	injected	IEXPLORE.EXE	Details
10040	2015-06-01 15:48:49.734455	fb88bff8e9b3c619b88a73f29c888c27 c873a2dc09d60b291d6ee874248fc4c8896e0e04303d4eac2a24ec37f776b9d7	Hooks	Code Hooks	Hooks Information

Showing 1 to 5 of 5 entries Previous 1 Next

Figrue 3: A sample that injected into all browsers.

```
    {
        "export": "PR_Write",
        "hook disassembly": [
            "push ebp",
            "mov ebp,esp",
            "sub esp,0x18",
            "push ebx",
            "push esi",
            "push edi",
            "mov edi,[ebp+0x8]",
            "mov eax,[edi]",
            "mov eax,[eax]",
            "xor ebx,ebx",
            "mov [ebp-0xc],ebx",
            "mov [ebp-0x8],ebx"
        ],
        "hook_code": "558bec83ec185356578b7d088b078b0033db895df4895df8",
        "hooking_module": "<unknown>",
        "module": "nss3.dll",
        "process_name": "firefox.exe"
    }
]
```

Figure 4: Hook disassembly.

signatures which focus in-depth analysis on the relevant and significant parts of the malware.

To illustrate VolatilityBot's versatility, Figures 2–4 show a few examples of successful extractions.

KNOWN WEAKNESSES IN VOLATILITYBOT

VolatilityBot is still a work in progress, and it's not perfect. The following are some of the weaknesses I have found during my research:

- False positives can be caused because legitimate *Windows* kernel drivers might be loaded during malware execution (which generally might happen as a result of plug-n-play devices in the VM) and might be dumped as well. Additionally, any new process started after the malware is executed will be dumped too. In order to avoid this it is helpful to cancel all automatic updates on the machine (browsers, *Windows Update*, etc.).
- The malware might detect the virtual environment, no matter how much effort is put into hiding it. Malware might have long sleep timers too, in order to avoid research and prevent us from getting the complete malicious code.

FUTURE DEVELOPMENT

There are a lot of ideas and directions for the future development of VolatilityBot a researcher can choose in order to use VolatilityBot for his/her own research:

1. Shell code extraction – extract the injected shell code of an exploit or malware.
2. Submitting to different machines or platforms in parallel as part of the same analysis, and getting different extracted code. For example, submit a malware sample to *Windows XP* and *Windows 7*, either x86 and x64, and get all the possible injected payloads.
3. Automated extraction of malware configuration (specific to malware family).
4. Extraction of IP addresses, mutexes, etc.

APPENDIX

VolatilityBot can be used in two different modes:

1. Daemon mode

The VolatilityBot daemon runs in the background and looks in the database for malware samples awaiting analysis

New samples can be submitted using:

```
python VolatilityBot.py -e -r -filename ~/samples_
folder
```

The daemon itself is executed using:

```
python VolatilityBot.py -D --sleep 60
```

2. Script mode

VolatilityBot is executed, and once the analysis queue is empty, it will exit and display a summary with statistics:

A. Single file – a single file to be processed (60 seconds timeout):

```
python VolatilityBot.py -filename ~/sample.exe --
sleep 60
```

B. Folder – submit a folder containing PE files (recursive) and create a work queue:

```
python VolatilityBot.py -r -filename ~/sample_folder
-sleep 60
```

C. Submit a folder while skipping existing files in the database:

```
python -r -s VolatiliotyBot.py -filename ~/sample_
folder -sleep 60
```

D. Re-submission of failed files (according to the status in the database):

```
python -Q VolatilityBot.py -sleep 60
```