LOUISIANA STATE UNIVERSITY

Investigating the Atom text editor for Linux with memory forensics – A plugin for the Volatility framework

By
Raphaela Mettig, Adam Yu

CSC 7360 Fall 2018 – Memory Forensics

Investigating the Atom text editor for Linux with memory forensics – A plugin for the Volatility framework

Raphaela Mettig[a], *Adam Yu*[a*]

[a] *School of Electrical Engineering & Computer Science, Louisiana State University, USA*

ARTICLE INFO

Keywords:
Memory forensics
Computer forensics
Memory analysis
Volatility framework
Linux

ABSTRACT

Atom is a popular cross-platform, open source text editor developed by GitHub. It was built using the Node.js-based Electron framework for JavaScript. One of the main reasons Atom is such a popular choice among developers is because it is "hackable" by either developing packages (i.e. plugins, add-ons) or altering the core structure of the editor. We decided to investigate what kind of footprints the editor could leave behind in volatile memory. However, our tool of choice, the Volatility memory forensics framework currently has a very limited amount of userland application-specific plugins. Therefore, in this paper, we'll introduce a basic plugin we developed for the Volatility framework that combines existing kernel space plugins and applies certain filters to those outputs to aid performing a basic memory forensics investigation on the Atom text editor. In the end, we were able to extract the filepath for existing files that were currently open in the active pane of the application. Many popular applications today are built on top of the Electron framework, and we believe that our plugin is a start to help write other plugins that can help analyze other Electron-based applications.

1. Introduction

Atom is a popular cross-platform, open source text editor developed by GitHub. It was built using the Node.js-based Electron framework for JavaScript. One of the main reasons Atom is such a popular choice among developers is because it is "hackable" by developing packages (i.e. plugins, add-ons) that can alter the core structure of the editor, its appearance, and much more. Being such a versatile and popular tool, we decided to investigate what kind of footprints the editor could leave behind in volatile memory for our Memory Forensics class project.

We use the Volatility memory forensics framework, which currently has a very limited amount of userland application-specific plugins, so we decided to write a basic plugin we for Volatility that combines existing kernel space plugins and applies certain filters to those outputs to aid performing a basic memory forensics investigation on the Atom text editor.

Our end-goal was to develop a plugin capable of retrieving basic information such as open filenames, file paths and, time permitting, a snippet of the contents of each file from the memory dump of an image that had the Atom text editor installed.

In this paper, we'll introduce the plugin we developed and share about what the research process was in order to achieve the end product.

2. Background

In this section, we'll provide a short overview about some of the technologies that were involved during the development of our plugin.

2.1 Atom

Atom is a free, open source text editor that was

* *Corresponding authors:*
E-mail address: rsanto2@lsu.edu (R. Mettig), ayu4@lsu.edu (A. Yu)

developed by GitHub with the intent to offer the customizability of a text editor such as Emacs or Vim with a more user-friendly graphical interface. It's "hackable text editor for the 21st century" (Atom, 2018) that was built on the Electron framework, and one of its main features is allowing users to modify the editor as they please by writing and installing packages with features that were developed by the GitHub developers or by the community to extend the functionality of the editor, which could range from changing the theme of the interface, adding a new language grammar, or even integrate other applications such as terminal into the editor (Atom Flight Manual, 2018.)

2.2 The Electron framework

The Electron framework "is an open source library developed by GitHub for building cross-platform desktop applications with HTML, CSS, and JavaScript. Electron accomplishes this by combining Chromium and Node.js into a single runtime and apps can be packaged for Mac, Windows, and Linux." (Electron Doc., 2017) Essentially, it facilitates the development of desktop applications by allowing the user to write code as if it were a website regardless of the operating system the application is running on. Some note-worthy popular applications that were developed using the framework are: Skype, Slack, Visual Studio Code, the JetBrains suite, Discord, and WordPress for Desktop, just to name a few.

2.3 Volatility

The Volatility memory forensics framework is a completely open source and free to use tool that is used by computer forensics experts to perform the extraction of digital artifacts from volatile memory (RAM) samples (The Art of Memory Forensics, 2014.) Just like the above-mentioned tools, it is also hosted on GitHub for the community to access all its source code, documentation, and community projects such as plugins.

While Volatility is not capable of acquiring memory samples, it supports the analysis of memory images taken from a wide range of operating systems: 32- and 64-bit Windows, Linux, Mac system, and 32-bit Android (AMF, 2014). It is also written in Python, which means that it can be installed on any machine that can compile Python code giving it an advantage over other memory forensics tools that are exclusive to Windows.

Our main feature of interest was the ability to write plugins compatible with the framework. Volatility can be used as a Python library to develop plugins that allow the user to achieve a specific goal. Most the native Volatility plugins are generic plugins that could be used to obtain information from the kernel regardless of the applications installed in that memory dump, so our goal was to contribute to the expansion of the userland application plugins available.

3. Methodology

3.1 Setup

We started by setting up the environment in the virtual machine (VM) that we were going to get our memory dumps from. VMWare Fusion was our virtualization application of choice as it facilitates the process of taking snapshots and extracting the dumps as *.vmem* and *.vmss* files. Our operating system of choice was Ubuntu 16.04 and in order to install Atom we used the `apt` tool in the command line.

The next step was to obtain a profile for the operating system that we were using so that it could be analysed with Volatility, so we did a quick install of Volatility and ran the *dwarfdump* tool (Volatility Linux Wiki, 2016) in the VM and transferred it to another VM where the analysis would be performed.

Lastly, we opened the Atom editor and loaded some files on it that would be used to obtain data about the application throughout the analysis of the memory dump.

3.2 Analysis

We began the analysis by looking into the atom processes to see what we could find with basic Volatility commands such as `pslist` and `pstree` to search for the atom process running on the machine, and we found that Atom had a process named "atom" which then spawned a few different processes (all with the same name), so we decided to dump its memory to check if we could find anything related to any open filenames, but that did not yield any interesting results for us.

Since we couldn't easily obtain much knowledge about the application solely by looking at the dump, we looked up the source code for Atom to see if we could find any hints of the buffer data structures that would hold the contents of what we were looking for such as the filenames of files open in the Atom text editor at the time the memory dump was taken. We found a couple of interesting objects in our research: the *TextBuffer* object and the *PaneContainer* object. To further inspect those objects, we ran the `linux_yarascan` plugin to check for any patterns of where the filenames of the open files are in memory and for more information on the objects that we found in the source code. We also created some test files with names and content that would be easy to find in the dump and opened them in the Atom text editor before taking the memory dump. Again, we initially ran the `linux_yarascan` plugin to find the interesting objects and their location in memory. Several addresses that looked a little promising were returned along with their addresses located in which atom process. With that information, we decided to use `linux_volshell` to further inspect each of those addresses found from the Yarascan plugin. After checking the contents that came before and after those addresses, nothing of value was found.

We searched for the test filenames we created to see if we could find any patterns within the Atom processes. A couple of interesting function names like *application:reopen-project*, save-path, and *openPaths*, were found and seemed close enough to the file paths of the opened files in Atom. Running a Yarascan on *application:reopen-project* yielded some potentially useful results: we came across a filepath to what seemed to match one of the filepaths of the files that were open in the active pane (the pane that had the focus on so that a file can be edited). The filepaths were also at a consistent offset from the *application:reopen-project* name in the memory dump. Looking into the *save-path* function, we were also able to find the filepaths of the files that were opened in Atom that were also in the active pane. The Yarascan on the *openPaths* function did not return any interesting information. Also, when we ran a Yarascan searching for a specific filename we created and had open in Atom, we found some HTML tags involving the filenames and filepaths of the files that were in the active pane containers in Atom. The filepaths were found in a `file://` URI within the `<href>` HTML tag. This was one of the results we had that had the greatest potential to aid in our development of our plugin.

We then checked for the "**Error! Hyperlink reference not valid**" string in the Atom processes also using a Yarascan to make sure that this was an easy pattern to search for. This came up with a several other random files Atom also used, so we had to find another way to find the "**Error! Hyperlink reference not valid"** string in the `<href>` tag. We searched the atom process based on the byte pattern: `22 66 69 6c 65 3a 2f 2f`, which were the ASCII values for the string `file://`. That narrowed our results down to only the filenames and filepaths of what was open in Atom.

When it came time to write our plugin, we were not able to figure out how to call the `linux_yarascan` plugin from our own plugin, so we had to find a different way to find the filenames. We decided to use the `linux_lsof` plugin and filtering the results to the atom process to see what kinds of handles Atom had open and check if any of the filenames we had open were listed in there. After running it we were able to find some of the filenames that were opened in Atom, which were only the files that we had open in the active pane, allowing us to find a pattern between the files that the atom process had open. That led to the exclusion of any file that was found in some of the root directories and single named files. This was the information that allowed us to write our plugin.

4. The plugin

4.1 Development

To start the development process we first had to find some examples of already written plugins so that we would have an idea of how to write our plugin. There is not a lot of information available on the development of plugins for Volatility: a few resources online that helped us get started with our development – mostly blog posts written by others in the DFIR community – though most of them dealt with writing plugins for Windows memory dumps. Those were useful to some extent, however there is even less public documentation on writing plugins for a Linux memory dump, so we ended up looking at the ones that are included in Volatility. We were able to get an idea of how to structure our plugin, how the Volatility library works, and what functions to write and use.

We had to analyse the results we gathered from the various Yarascans we ran, the `linux_pslist` plugin, and the `linux_lsof` plugin. There was a great amount of useful information we obtained from some of the Yarascans, but we ran into a problem when we tried writing our plugin. Because of the limited amount of documentation there is on writing Volatility plugins, we were not able to figure out how to use the `linux_yarascan` plugin from our own. Our results from the `linux_lsof` plugin to look for some sort of pattern between each of the filenames we found that were open in Atom. We noticed that any file that was open from any of the root directories */dev, /sys, /opt, /usr*, and */var* were the ones we could easily rule out. The other outputs had a pattern of `socket:[number]`, `pipe:[number]`, or `anon_inode[number]`, so we were able to rule those out as well. The biggest issue we had in developing our plugin was figuring out how to get the output from the `linux_pslist` plugin and using it within ours. Looking through the Linux plugins that already came with Volatility really helped us understand how to use the `linux_pslist` in our plugin since there were more common occurrences of the use of the `linux_pslist` plugin. We also found out which functions were used through each of the plugins that we should also write in ours, along with finding what each of those functions are supposed to do through a couple of short tutorials on how to write a Volatility plugin.

4.2 Implementation

After finding out how we should structure out plugin and what exactly we were going to with it, we started defining the core functions of a Volatility plugin. We wanted to use `linux_pslist` to find the atom processes and filter them out; using our output from `linux_pslist`, we would run `lsof()` on all the atom processes to find the filenames and filepaths of the opened file in Atom that are in the active pane. We started with the `render_text`, `unified_output`, and `generate` functions, which all came together to help print out the table and how our output would look. We were able to define our column names and the spacing between each column through the `render_text` function. The rows are then printed based off of what we filtered out from the `lsof()` function ran on each atom process. We had a helper function called `filter_paths`, which takes the path that is output in the second column of the `lsof()` function and splits the string based off of the "/" character. Any string/file that does not have the "/" character in it is disregarded, along with any string/file that is in any of the root directories */dev*, */sys*, */opt*, */usr*, and */var*.

```
# returns true if a relevant file opened
in atom was found
def filter_path(self, filepath):
     filter_list = ["dev", "sys",
                    "opt", "usr", "var"]
     path = filepath.split("/")
     if len(path) == 1:
         return False
     if path[1] in filter_list:
         return False
     for index in path:
         if len(index) > 0
                 and index[0] == '.':
             return False
     return True
```

Fig. 1. `filter_path` function from our `linux_atom_scanner` plugin.

The generator and unified_output functions structure the output received from the other functions into an appealing manner.

```
Address            PID    PPID   Name       Path
------------------ ------ ------ ---------- --------------------------------
------------------
0xffff88009d535280   2023   1447 atom      /home/raphaela/Documents/code/osb-
cli/cli-menu.cpp
0xffff88009d535280   2023   1447 atom      /home/raphaela/Documents/code/osb-
cli/YARAFINDME
0xffff88009d535280   2023   1447 atom      /home/raphaela/Documents/code/osb-
cli/YARAFINDME
```

Fig. 2. `linux_atom_scanner` plugin output.

4.3 Limitations

There are a couple of limitations our plugin has. Our plugin only works for Linux memory dumps, and our plugin can only get pre-existing files that were opened in Atom that and that are in the active pane (not all of the files opened in the Atom program).

5. Conclusions and future work

Applications that are built on top of the Electron framework are very interesting to analyze. Since the applications based on Electron, it makes the memory dump of the application harder to analyze because there are multiple processes with the name of the application with an overarching parent along with many other child processes with the application name. In our case, looking through Atom, we found that most of what we were interested in was in the parent atom process, but that may not hold true for other Electron-based applications. Since Atom is written in JavaScript and Electron also in C++, we had trouble trying to figure out what structure held the information we were looking for and where those structures were, especially due to the fact that the framework provides programming for Atom an entire layer of abstraction from its internal components, making the code analysis and reverse engineering process even more difficult.

For our plugin, there are a couple of features that we plan to implement in the future. One of them would be to get the `linux_yarascan` plugin to work inside our plugin so that we would be able to get all the files and filepaths that were open in Atom. We also plan to implement this plugin for other operating systems such as Windows and Mac OS, since Atom is cross-platform. It would also be interesting to look into the differences, if any.

Many applications used today are built on top of the Electron framework. In the future, there may be a need to be able to look into Electron-based applications to get interesting information from it. We believe that our plugin is a start to help write other plugins that can help analyze other Electron-based applications.

References

Atom, https://atom.io/, 2018

Atom Flight Manual, https://flight-manual.atom.io/, 2018

Electron Docs, https://electronjs.org/docs/tutorial/about, 2018

Ligh M.H., Case A., et al., Art of Memory Forensics (AMF). 1st Edition. 2014

Tomchop, Tutorial – Volatility Plugins & Malware Analysis, http://tomchop.me/2016/11/21/tutorial-volatility-plugins-malware-analysis/, 2016

Volatility Plugin Tutorial, https://github.com/iAbadia/Volatility-Plugin-Tutorial, 2017

Volatility Usage, https://github.com/volatilityfoundation/volatility/wiki/Volatility-Usage, 2017

Volatility Linux Wiki, https://github.com/volatilityfoundation/volatility/wiki/Linux, 2016