ZEIT8028: Digital Forensics

Lab 3: Registry Forensics

Background

Your boss is so pleased with how you conducted your first digital forensic investigation that she has given you the day off! Unfortunately, you and I both know that doesn't mean you're free to take off down the pub. Operational leave from investigations should always be relished and is the perfect time to conduct research and development. As you've recently been learning about the Windows Registry and how it works, you decide that's a good place to focus your learning efforts.

The Windows Registry is a hive of malware persistence blind spots, most of which are either poorly documented or not documented at all. Although it would make for an exciting BSides presentation to find some new and novel Windows Registry persistence mechanisms, you decide it best to research and catalogue already known and well-documented methods.

Instead of documenting your research in a form that you might forget, you decide that you’ll incorporate your findings into a tool instead. Therefore, your task is to create your first forensic analysis tool, in Python. Not only will you learn something new, but you’ll also create an automated tool that can be used in all your future forensic investigations.

Exercise 1: Research

Your first task is to read the provided **Common Malware Persistence Mechanisms** blog published by INFOSEC Institute. You can find a copy of the blog at the following location within the `Lab 3 - Registry Forensics.7z` lab file bundle:

```
PATH: ./blog/Common Malware Persistence Mechanisms.htm
```

Exercise 2: Development

Your next task is to implement a registry persistence analysis tool that will enumerate all the known Windows Registry keys that you have learnt about in the blog. The tool is to be written in Python.

For your convenience, someone has already started developing your tool for you. Open and begin extending the Python program located at the following location within the `Lab 3 - Registry Forensics.7z` lab file bundle:

```
PATH: ./findbadness.py
```

To help you complete this task, numerous resources have been made available to you:

```
PATH: ./doc/python-2.7.16-docs-html/index.html
NOTE: The full Python 2.7 offline documentation
```

```
PATH: ./Registry/
NOTE: A Python package that includes low level Windows Registry
support
```

```
PATH: ./doc/reg-doc/
NOTE: Additional Windows Registry documentation
```

```
PATH: ./reference_data/
NOTE: A collection of NIST reference Windows Registry hives
```

```
PATH: ./regview.py
NOTE: A graphical Windows Registry hive viewer
```

Your completed tool will come in very handy for future investigations (*hint hint*), so put it aside somewhere safe for future use.

▾ **Hint**

You can open the provided code in your VM using `VS Code` (make sure you're in the correct directory first):

```
analyst@forensics~$ code findbadness.py
```

▾ **Hint**

It's probably easier to copy the registry hives to a working directory:

```
analyst@forensics~$ cp findbadness.py ~/Documents/
analyst@forensics~$ cp reference_data/Win10_10586_IE11+Edge
/p1/Windows/System32/config/SYSTEM ~/Documents/
analyst@forensics~$ cp reference_data/Win10_10586_IE11+Edge
/p1/Windows/System32/config/SOFTWARE ~/Documents/
analyst@forensics~$ cp reference_data/Win10_10586_IE11+Edge
/p1/Users/Forensics/NTUSER.DAT ~/Documents/
analyst@forensics~$ cd ~/Documents
analyst@forensics~$ ls -lah
total 86M
drwxr-xr-x  2 analyst analyst 4.0K Mar  5 00:54 .
drwxr-xr-x 13 analyst analyst 4.0K Mar  5 00:45 ..
-rw-r--r--  1 analyst analyst 2.8K Mar  5 00:54 findbadness.py
-rw-r--r--  1 analyst analyst 768K Mar  5 00:54 NTUSER.DAT
-rw-r--r--  1 analyst analyst  74M Mar  5 00:54 SOFTWARE
-rw-r--r--  1 analyst analyst  11M Mar  5 00:54 SYSTEM
```

▾ **Hint**

Run the script to get an understanding of what it does:

```
analyst@forensics~$ python3 findbadness.py
usage: findbadness.py [-h] -f FILEPATH
findbadness.py: error: argument -f is required
```

```
analyst@forensics~$ python3 findbadness.py -f SYSTEM
[*] Attempting to open registry hive at location [/home/analyst
/Scratch/SYSTEM]
[*] Attempting to acquire root key of hive
[*] Enumerating children of the root key [ROOT]

[ROOT]
 +-- [ActivationBroker]
 +-- [ControlSet001]
 +-- [DriverDatabase]
 +-- [HardwareConfig]
 +-- [Keyboard Layout]
 +-- [Maps]
 +-- [MountedDevices]
 +-- [ResourceManager]
 +-- [ResourcePolicyStore]
 +-- [RNG]
 +-- [Select]
 +-- [Setup]
 +-- [Software]
 +-- [WPA]

[*] Fin
```

▼ **Solution**

The `findbadness.py` script contains functions to enumerate the registry paths and keys. Your task is to add functionality to retrieve keys, values, and types additionally. To achieve that, it's helpful to first add to the script a function to convert the hivepaths:

```
def convert_to_hive_path(key_path):
    hive_path = ''

    result =
re.match(r'(?i)HKEY_CURRENT_USER|HKCU|HKEY_LOCAL_MACHINE|HKLM',
key_path)

    if result is not None:
        root_key_name = result.group(0).upper()

        if root_key_name in ['HKEY_LOCAL_MACHINE','HKLM']:
            subkey = key_path[len(result.group(0))+1:]

            result = re.match(r'(?i)SOFTWARE|SYSTEM', subkey)

            if result is not None:
                hive_path += subkey[len(result.group(0))+1:]

            else:
                raise UnsupportedNameKeyPathException

        elif root_key_name in ['HKEY_CURRENT_USER', 'HKCU']:
            hive_path += key_path[len(result.group(0))+1:]

    else:
        raise UnsupportedNameKeyPathException

    return hive_path
```

Then, extend the `main` function to return data about the keys, values, and types:

```
def main():
...
            # convert to hive key naming convention
            hive_path = convert_to_hive_path(name_key_path)

            # find and print name key values and data
            print('[*] Attempting to open name key
[{}]'.format(name_key_path))
            name_key = reg.open(hive_path)

            print('\nName key name: {}'.format(name_key_path))
            print('Name key last update datetime:
{}'.format(name_key.timestamp()))

            value_key_count =name_key.values_number()
            print('Name key value key count:
{}\n'.format(value_key_count))

            print('[*] Attempting to enumerate value keys')
            if value_key_count > 0:
                for value_key in name_key.values():
                    print('\n+-- Value key name:
{}'.format(value_key.name()))
                    print('+-- Value key value type:
{}'.format(value_key.value_type_str()))

                    if value_key.value_type() ==
Registry.RegBin:
                        print('+-- Value key data:
{}'.format([byte for byte in value_key.value()]))
                    else:
                        print('+-- Value key data:
{}'.format(value_key.value()))
            else:
                print('\n+-- Name key contains no value keys')
...
    return

if __name__ == "__main__":
    main()
```

You should modify it to include the additional registry key/value pairs, etc. listed in the blog you read at the top of the lab.

Download a sample solution

Check the sample solution file and associated walkthrough video for further information.

Exercise 3: Registry Last Update Timeline

As you discovered during your first investigation, MFT timeline analysis is a very powerful technique. Wouldn't it be handy if you could create a similar timeline using the Windows Registry, using a registry key's last update time?

Your next task is to write a tool (or extend the tool you wrote in the previous exercise) that parses a registry hive and outputs all the keys and their related value data key pairs in chronological order, using the key's **last update time**.

▾ **Hint**

You can use the following function to sort the parsed registry keys:

```
# sort name keys by datetime
print('[*] Sorting name keys in registry hive by datetime')
name_keys.sort(key=lambda x: x[0])
```

▾ **Hint**

Use the following function to enumerate the key names, paths, and timestamps:

```
def enumerate_all_children_name_keys(root_name_key):
    name_keys = []

    def recursively_hunt_namekeys(root_name_key, depth=0,
maxdepth=0):
        if depth <= maxdepth:
            path = root_name_key.path()[5:]
            timestamp = root_name_key.timestamp()
            name_keys.append((timestamp, path))
            if root_name_key.subkeys_number() > 0:
                for name_key in root_name_key.subkeys():
                    recursively_hunt_namekeys(name_key,
depth=depth+1, maxdepth=maxdepth)

    recursively_hunt_namekeys(root_name_key, maxdepth=15)

    return name_keys
```

▾ **Hint**

Use this function to print your desired output:

```
    # print sorted summarized name keys and name values
    print('[*] Commencing printing of name key summaries')

print('datetime,name_key_path,value_key_name,value_key_type,value_key_data')
    for entry in name_keys:
        timestamp, name_key_path = entry
        name_key = reg.open(name_key_path)
        if name_key.values_number() > 0:
            for value_key in name_key.values():
                try:
                    # check for type Regin and present data as
byte list
                    if value_key.value_type() ==
Registry.RegBin:
                        print('{},{},{},
{},{}'.format(timestamp, name_key_path,value_key.name(),
value_key.value_type_str(), [byte for byte in
value_key.value()]))
                    else:
                        print('{},{},{},
{},{}'.format(timestamp, name_key_path,value_key.name(),
value_key.value_type_str(), value_key.value()))

                except (UnicodeDecodeError, UnicodeEncodeError,
Registry.RegistryParse.UnknownTypeException):
                    # pass registry name keys that have
unsupported types and unicode encoding/decoding issues
                    pass
        else:
            print('{},{},,'.format(timestamp, name_key_path))


    print('\n[*] Fin')

    return
```

▾ **Solution**

Download a sample solution

Check the sample solution file and associated walkthrough video for further information.

Exercise 4: Enumerating Deleted Keys (Bonus)

If you've made it this far and are wanting a slightly more difficult challenge, there's yet another tool that can be added to your bag of tricks: a tool that allows you to enumerate or search deleted registry keys.

It is likely that you'll have to do some research. The required reading has been made available to you at the following location within the `Lab 3 - Registry Forensics.7z` lab file bundle:

```
PATH: ./doc/reg-doc/TheWindowsNTRegistryFileFormat.pdf
```

▾ **Solution**

Not this time :D (It's supposed to be a challenge!)