AltFS - The Alternative Fileless File System

AltFS provides a virtual file system, over non-file artifacts, to demonstrate hidden storage techniques.

Context

Recently we spent some time investigating the elusive world of fileless operations.
For some time now, we have been noticing that malicious actors are minimizing their use of files on disk.
The need to store persistent data on the target machine is natural and common. However, storing data in files and writing files to disk make the attacker more visible to the threat hunting software and IR investigations.

Therefore, the attacker faces the challenge of bypassing traditional AVs, advanced EDRs and a range of monitoring tools.

Let us break apart the attacker's rivals:

1. Traditional AV engines scan files on the hard disk.
2. More advanced EDRs also examine file operations as they occur, trying to expose malicious use of files.
3. Monitoring tools log file access on the system API level.
4. IR teams will probably use sandboxes to get a summary of a malware's key actions (IoCs).
5. Once a new threat has been identified, security teams and vendors will try to formulate a signature of the malware characteristics in order to prevent future use of it. The signature may take a variety of forms that is as wide as an IoC can be (file path, registry value, TCP connection etc.).

In our latest research, we explore how built-in OS artifacts may be abused to facilitate hidden data storage, while avoiding direct access to files.

In order to help simulating similar techniques, we have built a base framework called **AltFS - The Alternative Fileless File System**.

AltFS is:

- **Pluginable**:
  Anyone can write a new provider, to support new artifacts (e.g. Windows Registry, macOS User Defaults system)
- **Multi Platform**:
  With concrete provider implemetations for Windows and macOS
- **Multi-Layered**:
  Storage model features 2-levels indexing (bucket index -> value index)
- **Binary**:
  Data is serialized using msgpack

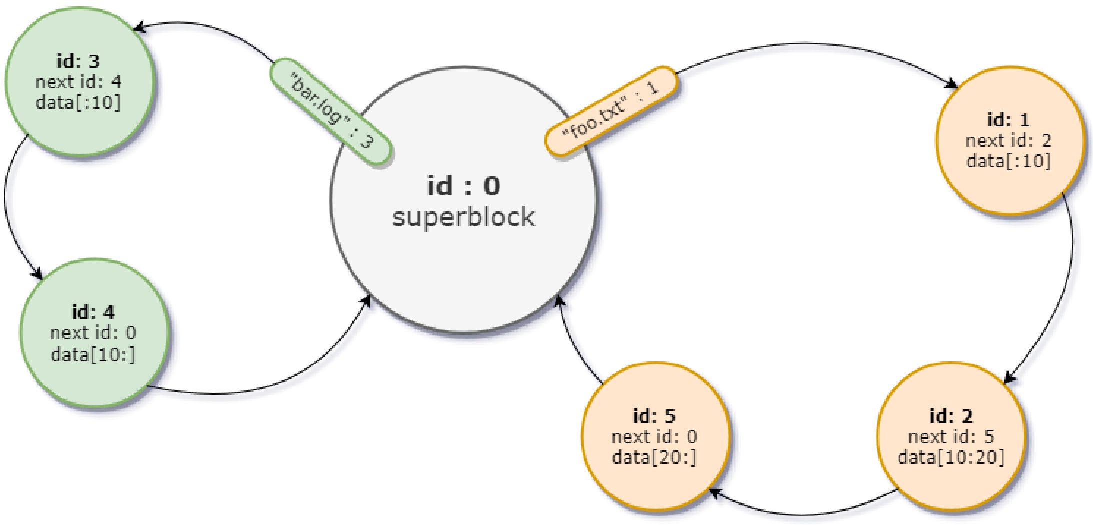
Advantages:

- **Covertness**:
  Storing data in built-in system artifacts instead of files - distances the monitoring eye from the attacker's actions
- **Machine-uniqueness**
  The data storage locations are calculated based on machine unique identification properties, so it is harder to identify and sign the existence of this framework on a machine.
- **Persistency**:
  As storage providers make use of OS artifacts, data remains persistent
- **Ease of use**:
  AltFS interface resembles well-known File APIs, including control over an in-file pointer and FD/handle-like files management

The File System Structure and Models:

- In general, the supported storage structure is a tree structure. But, of course a tree is a generalization of a flat storage structure (by having only one single root node).

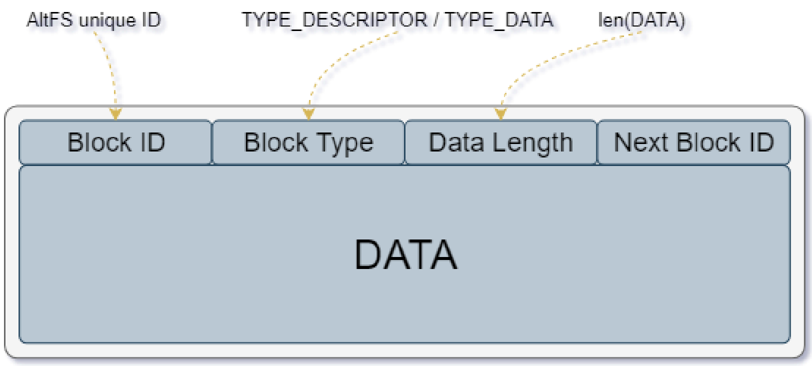
- The virtual structure built on top of the storage is: linked chains of blocks - starting from the superblock and ending at the superblock.
- Each file consists of its own chain, but all files' chains start and terminate at the superblock.
- Block chains are linked using the block headers, that include the ID of the next block. The last data block of the file points to the superblock, that represents the data termination (EOF).
  The following is a diagram of all blocks in a file system that contains 2 files:
  i. "foo.txt" which is 25 bytes in size
  ii. "bar.log" which is 19 bytes in size



Terminology:

- **Bucket:**
  A bucket is a container of values, referenced by the 1st level index. e.g. Registry key
- **Value:**
  A Value is where the block data is actually stored, referenced by the 2nd level index. e.g. Registry value under a key (=inside a bucket)
- **Block:**
  The data structure of the virtual atomic write unit. There are 2 types of blocks: Descriptor and Data. This corresponds to a hard disk sector in the sense that it is the minimal unit for all R/W operations. Unlike a hard disk sector, the Block's size may vary. Te following snippet and drawing show the block structure:

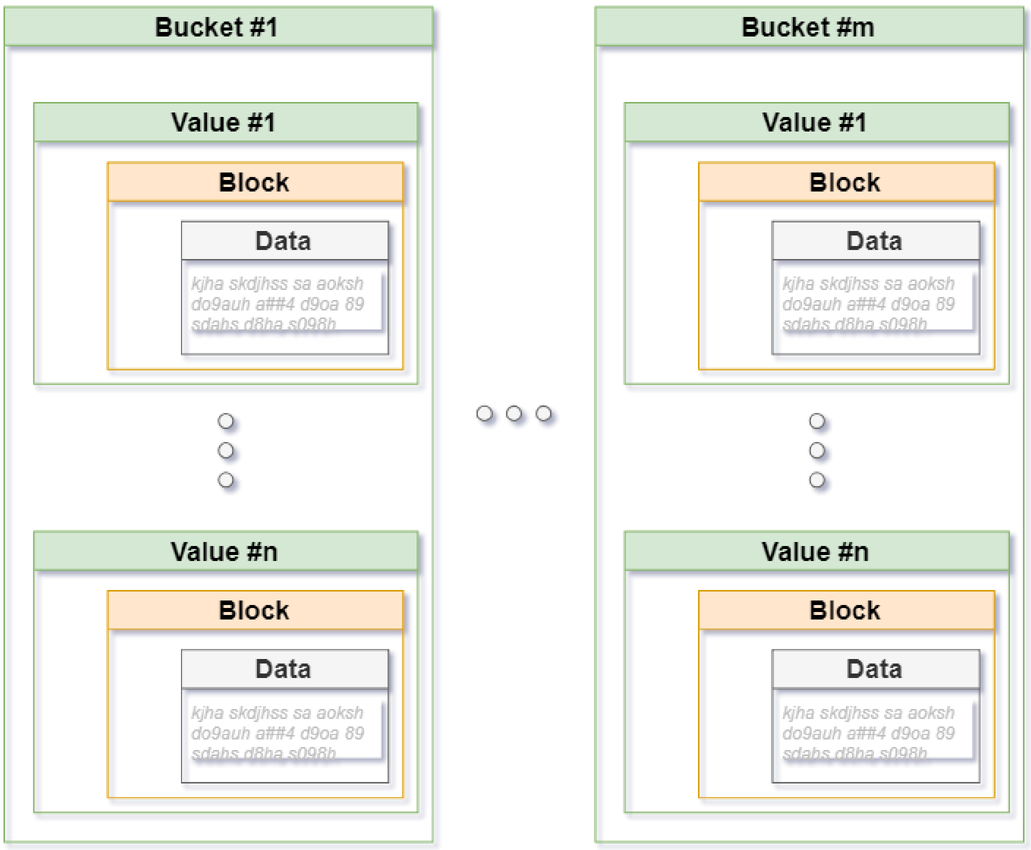
```
{
  block_id : #ID
  block_type : (TYPE_DESCRIPTOR, TYPE_DATA)
  data_length : len(data)
  next_block_id : #NEXT_ID
  data : "BA DC AF FE ..."
}
```



- **Superblock/root block/first block:**
  The first block in the file system. Serves 2 purposes: - Holds the Descriptor object that contains the metadata of the whole file system (think of it as the NTFS table) - It is the termination block of all files' linked chains Its location in the storage is determined in a calculation based on the machine identification string

- **Descriptor:**
  The data structure that holds the FS global mapping of all files' linked chains.
  In practice, it maps a file name to the ID of its first data block (the start of the linked chain).

The following drawing should summarize the relations of some of the terms described above:
green = an actual OS artificat, e.g. Registry key and value
orange = virtual data structure of AltFS


Bucket #1
Value #1
Block
Data
kjha skdjhss sa aoksh do9auh a##4 d9oa 89 sdabs d8ba s098h
Value #n
Block
Data
kjha skdjhss sa aoksh do9auh a##4 d9oa 89 sdabs d8ba s098h
Bucket #m
Value #1
Block
Data
kjha skdjhss sa aoksh do9auh a##4 d9oa 89 sdabs d8ba s098h
Value #n
Block
Data
kjha skdjhss sa aoksh do9auh a##4 d9oa 89 sdabs d8ba s098h

Visual Example Demonstration:

To get a sense of how the FS data is distributed in storage, let us present how an example file system is reflected in Registry.
First let's create a simple Registry based file system, and create 2 files in it:

```
from AltFS import AltFS
afs = AltFS("RegistryStorageProvider", "network_card",
        base_key_full_path=r"HKEY_CURRENT_USER\AppEvents\EventLabels", max_block_size=10)

file_1 = afs.create_file("foo.txt")
file_2 = afs.create_file("bar.log")
file_1.write("ABCDEFGHIJKLMNOPQRSTUVWXY")
file_2.write("ABCDEFGHIJKLMNOPQRS")
```

Now let's examine the Registry dump under the chosen base key.

Legend:

- *"[key]"* suffix = A registry key
- *"[val]"* suffix = A registry value
- The presented JSON objects are the representation of the stored blocks. The raw data as exists in storage is of course serialized and encoded.

- Note that only AltFS applicable values are presented. The original registry values that exist on the machine are left unharmed.
- Note that all blocks are of type 1 (DATA), except for the superblock, which is of type 0 (DESCRIPTOR) and is located under *"HKEY_CURRENT_USER\AppEvents\EventLabels\Notification.SMS\Notification.Looping.Alarm5.0000"*
- Try to follow one of the file chains, starting and ending from and at the superblock.

```
"HKEY_CURRENT_USER\\AppEvents\\EventLabels":
    [key] .Default
    [key] ActivatingDocument
    [key] AppGPFault
    [key] BlockedPopup
    [key] CCSelect
    [key] ChangeTheme
    [key] Close
    [key] CriticalBatteryAlarm
    [key] DeviceConnect
    [key] DeviceDisconnect
    [key] DeviceFail
    [key] DisNumbersSound
    [key] EmptyRecycleBin
    [key] FaxBeep
    [key] FeedDiscovered
    [key] HubOffSound
    [key] HubOnSound
    [key] HubSleepSound
    [key] LowBatteryAlarm
    [key] MailBeep
    [key] Maximize
            [val] Maximize.0000:
                  {
                      "block_id": 5,
                      "block_type": 1,
                      "data": "UVWXY",
                      "data_length": 5,
                      "next_block_id": 6
                  }
            [val] Notification.Default.0001:
                  {
                      "block_id": 6,
                      "block_type": 1,
                      "data": "",
                      "data_length": 0,
                      "next_block_id": 0
                  }
    [key] MenuCommand
    [key] MenuPopup
    [key] MessageNudge
    [key] Minimize
    [key] MisrecoSound
    [key] MoveMenuItem
    [key] Navigating
            [val] DisNumbersSound.0000:
                  {
                      "block_id": 3,
                      "block_type": 1,
                      "data": "ABCDEFGHIJ",
                      "data_length": 10,
                      "next_block_id": 4
                  }
            [val] Notification.Looping.Call7.0001:
                  {
                      "block_id": 7,
                      "block_type": 1,
                      "data": "ABCDEFGHIJ",
                      "data_length": 10,
                      "next_block_id": 8
                  }
    [key] Notification.Default
    [key] Notification.IM
            [val] WindowsUnlock.0000:
                  {
                      "block_id": 2,
                      "block_type": 1,
                      "data": "",
                    "data_length": 0,
                    "next_block_id": 7
                }
        [key] Notification.Looping.Alarm
        [key] Notification.Looping.Alarm10
        [key] Notification.Looping.Alarm2
                [val] WindowsLogon.0000:
                {
                    "block_id": 1,
                    "block_type": 1,
                    "data": "",
                    "data_length": 0,
                    "next_block_id": 3
                }
                [val] VS_BuildSucceeded.0001:
                {
                    "block_id": 8,
                    "block_type": 1,
                    "data": "KLMNOPQRS",
                    "data_length": 9,
                    "next_block_id": 9
                }
                [val] Notification.Looping.Call4.0002:
                {
                    "block_id": 9,
                    "block_type": 1,
                    "data": "",
                    "data_length": 0,
                    "next_block_id": 0
                }
        [key] Notification.Looping.Alarm3
        [key] Notification.Looping.Alarm4
        [key] Notification.Looping.Alarm5
                [val] PrintComplete.0000:
                {
                    "block_id": 4,
                    "block_type": 1,
                    "data": "KLMNOPQRST",
                    "data_length": 10,
                    "next_block_id": 5
                }
        [key] Notification.Looping.Alarm6
        [key] Notification.Looping.Alarm7
        [key] Notification.Looping.Alarm8
        [key] Notification.Looping.Alarm9
        [key] Notification.Looping.Call
        [key] Notification.Looping.Call10
        [key] Notification.Looping.Call2
        [key] Notification.Looping.Call3
        [key] Notification.Looping.Call4
        [key] Notification.Looping.Call5
        [key] Notification.Looping.Call6
        [key] Notification.Looping.Call7
        [key] Notification.Looping.Call8
        [key] Notification.Looping.Call9
        [key] Notification.Mail
        [key] Notification.Proximity
        [key] Notification.Reminder
        [key] Notification.SMS
                [val] Notification.Looping.Alarm5.0000:
                {
                    "block_id": 0,
                    "block_type": 0,
                    "data": {
                        "files_dict": {
                            "bar.log": 2,
                            "foo.txt": 1
                        }
                    },
                    "data_length": 92,
                    "next_block_id": 1
                }
        [key] Open
        [key] PanelSound
        [key] PrintComplete
```

```
    [key] ProximityConnection
    [key] RestoreDown
    [key] RestoreUp
    [key] SecurityBand
    [key] ShowBand
    [key] SystemAsterisk
    [key] SystemExclamation
    [key] SystemExit
    [key] SystemHand
    [key] SystemNotification
    [key] SystemQuestion
    [key] VS_BreakpointHit
    [key] VS_BuildCanceled
    [key] VS_BuildFailed
    [key] VS_BuildSucceeded
    [key] WindowsLogoff
    [key] WindowsLogon
    [key] WindowsUAC
    [key] WindowsUnlock
```

Indexing:

As the FS structure resembles a 2-level index storage - a hashing function is needed to ensure fair distribution of values across buckets. The hashing function chosen for this FS is the simplest modulo operation, called on the bits sum of a previous piece of information.

- **Next Block's Bucket ID Calculation**:
  When writing a series of blocks, the bucket ID in which to put the next block needs to be determined.
  It is calculated using bits sum of the data, that is given to the indexing hashing function, as shown:

  ```
  next_bucket_id = calculate_bits_sum(data) % buckets_count
  ```

- **First Block's Bucket ID Calculation**:
  The superblock is again special.
  In this case there is no previous data buffer to rely on, so the hashing function is called on the bits sum of the machine identification string, as shown:

  ```
  first_bucket_id = calculate_bits_sum(machine_identification_string) % buckets_count
  ```

  This ensures that the superblock location is different across machines.

How to use (example.py):

Note: each of altfs functions might raise an *InternalStorageOperationException*, so for a safer code - always wrap the calls with a try-except clause.
Note: using the "network_card" identification method requires the 3rd party 'getmac' module. In order to install it, run:

```
pip install getmac
```

```
from AltFS import AltFS
DATA = "lorem ipsum"

afs = AltFS("RegistryStorageProvider", "network_card",
            base_key_full_path=r"HKEY_CURRENT_USER\AppEvents\EventLabels", max_block_size=10)
```

```
DEBUG:AltFS:initializing AltFS with storage provider: network_card, machine identification method: RegistryStorageProvider
INFO:AltFS:INIT:number of buckets (=divider): 76
INFO:AltFS:INIT:machine identification string: FF:FF:FF:FF:FF:FF  # (anonymized)
INFO:AltFS:INIT:machine identification checksum: 53
INFO:AltFS:INIT:first bucket ID: 53
DEBUG:AltFS:writing block at (53:0):{'data_length': 38, 'next_block_id': 1, 'block_type': 0, 'block_id': 0, 'data': {'files_dict'
```

```
f = afs.create_file("foo.txt")
print f
```

```
<File: name: foo.txt, status: OPEN, pointer: 0, size: 0>
```

```
afs.get_file_names()
```

```
['foo.txt']
```

```
f.write(DATA)
f.get_pointer()
```

```
11
```

```
f.set_pointer(0)
f.read(len(DATA))
```

```
'lorem ipsum'
```

```
f.close()
print f
```

```
<File: name: foo.txt, status: CLOSED, pointer: 11, size: 11>
```

How to Write a New Provider:

1. Choose the target artifact (e.g. Windows registry). Keep in mind:
   i. Artifact should have a read/write/delete interface
   ii. Arbitrary insertions should not break any OS functionality
   iii. Artifact should be able to contain as much data as possible
   iv. If artifact is organized in a tree structure - make use of the 2-layer indexing of AltFS.
   If not - fixate the first level to a single pseudo index 0.
2. Implement the StorageProvider abstract class

Misc

- WMIStorageProvider requires the WMIClient helper DLL. This was developed to supplement missing functionalities of python's wmi module.
  The project's code sits under /lib, along with the compiled binaries. The provider uses the final DLL artifiact that is located at /providers/common/WMIClient.dll.

Authors

Dor Azouri (@bemikre)

License

BSD 3 - clause "New" or "Revised" License