Starting the System

We spin up the entire system at once, using a `docker-compose` command. Every sensor is built to retry connections to the sensing infrastructure, waiting for services to be available. Because of this, the early log messages *look* like errors, but are expected behavior as the sensors on the target system wait for the Sensing API to be ready.

We can bring everything up with:

```Bash
> docker-compose up
Starting savior_kafka_1 ...
Starting savior_cfssl_1 ...
Starting savior_cfssl_1 ... done
Starting savior_api_1 ... done
Starting savior_target_1_1 ... done
Attaching to savior_dropper_callback_1, savior_kafka_1, savior_cfssl_1, savior_api_1, s
cfssl_1             | 2018/01/08 15:44:00 [INFO] Initializing signer
...
```

There are a ton of log messages. When each sensor comes up we see a message that looks like:

```Bash
target_1_1          | Starting ps-sensor(version=1.20171117)
target_1_1          | Sensor Identification
target_1_1          |   sensor_id  == 419fe31d-c393-4219-bdd9-ff22c2e98304
target_1_1          |   virtue_id  == 6900f86f-ddd3-4edb-9a5c-25973100d98a
target_1_1          |   username   == root
target_1_1          |   hostname   == 828ad1fa6a57
target_1_1          | Sensing API
target_1_1          |   hostname   == api
target_1_1          |   http port  == 17141
target_1_1          |   https port == 17504
target_1_1          |   version    == v1
target_1_1          | Sensor Interface
target_1_1          |   hostname  == 828ad1fa6a57
target_1_1          |   port      == 11010
target_1_1          |   @ Waiting for Sensing API
```

How do we know everything started properly? We'll see particular log messages for each of the infrastructure components.

We know the Sensing API is ready when it starts serving the Root CA public key to clients:

```Bash
api_1               | [info] GET /api/v1/ca/root/public
api_1               | [debug] Processing with ApiServer.ConfigureController.ca_root_cer
api_1               |   Parameters: %{}
api_1               |   Pipelines: [:insecure]
api_1               | [info] Sent 200 in 437µs
```

Each sensor has multiple messages we could check to see that they are ready and progressing through the registration process. For instance, when the sensor starts the key registration and challenge process:

```Bash
target_1_1          |   + got private key response from Sensing API
target_1_1          |   % private key fingerprint(dc:21:73:5d:d7:6c:73:b7:8f:97:6e:4b:1
target_1_1          |   % CA http-savior challenge url(http://828ad1fa6a57:11030/path/t
target_1_1          |   @ Requesting Signed Public Key from the Sensing API
```

or when the sensor recieves its *public key*:

```Bash
target_1_1          |   + got a signed public key from the Sensing API
```

The clearest message is when each sensor emits the message:

```Bash
target_1_1          | Registered sensor with Sensing API
```

What Sensors are Running?

When everything is up and running, we can quickly look up which sensors are running in which Virtue's to sense our target user (in this case `root`):

```Bash
>  ./bin/dockerized-inspect.sh
[dockerized-run]
Getting Client Certificate
Running virtue-security
{
    "timestamp": "2018-01-08 15:35:22.151224Z",
    "targeting_scope": "user",
    "targeting": {
        "username": "root"
```

```
    },
    "sensors": [
        {
            "virtue": "57512854-834b-4848-af1c-e11bbd2897c9",
            "username": "root",
            "timestamp": "2018-01-08 15:35:01.528776Z",
            "sensor_name": "ps-sensor-1.20171117",
            "sensor": "a69f9df7-416e-4de3-8c7a-27ac09f1868f",
            "public_key": "...",
            "port": null,
            "kafka_topic": null,
            "address": "828ad1fa6a57"
        },
        {
            "virtue": "31d8414e-3825-4dbb-9db1-6516b40355f4",
            "username": "root",
            "timestamp": "2018-01-08 15:35:01.737116Z",
            "sensor_name": "lsof-sensor-1.20171117",
            "sensor": "0b43f335-9aef-47e8-a6b6-d13a5f7c5565",
            "public_key": "...",
            "port": null,
            "kafka_topic": null,
            "address": "828ad1fa6a57"
        },
        {
            "virtue": "9b5a2294-990d-4a52-9864-a2e5cac6f08e",
            "username": "root",
            "timestamp": "2018-01-08 15:35:00.888712Z",
            "sensor_name": "kernel-ps-sensor-1.20171117",
            "sensor": "248a406c-aabd-43c8-835c-a7cd76b11800",
            "public_key": "...",
            "port": null,
            "kafka_topic": null,
            "address": "828ad1fa6a57"
        }
    ],
    "error": false
}
```


Even with the public RSA keys of the sensors elided, this is a lot of data. What we want to see, in particular, is that three sensors are running:

- `sensor-ps`
- `sensor-kernel-ps`

- `sensor-lsof`

Each sensor is identifiable by both a unique **sensor_id** and their unique **public_key**.

Notes:

- The **port** and **kafka_topic** keys appear as `null` values in this data, but are not actually null in the database. These two values are specially controlled to reduce the possibility of unauthorized systems or bad actors accessing the log stream of a sensor or connecting to its Command and Control port.

Highlighting Missing PS Data

With both PS sensors and the LSOF sensor (`sensor-ps` , `sensor-kernel-ps` , `sensor-lsof`) running, we should be able to find discrepencies between the `ps` command put in place by the malware to hide itself (`sensor-ps` is exploited by this), and either the original system level `ps` (stashed in `/tmp` by the malware) or through kernel level pseudo- `ps` data placed into the kernel log.

We can see the discrepency easily; two of the three sensors will properly report that the `dropper.sh` malware is running, while the other won't:

Bash

```
> ./bin/dockerized-stream.sh --grep "dropper.sh" --pretty-print
[dockerized-run]
Getting Client Certificate
Running virtue-security
{
    "timestamp": "2018-01-09T16:28:35.851598",
    "sensor_name": "kernel-ps-sensor-1.20171117",
    "sensor_id": "983f955a-2a3e-4076-91b1-02e7aaf5dd3d",
    "message": {
        "user": "root",
        "process": "/bin/bash /tmp/dropper.sh",
        "pid": "11"
    },
    "level": "info"
}
{
    "timestamp": "2018-01-09T16:28:38.346031",
    "sensor_name": "lsof-sensor-1.20171117",
    "sensor_id": "3d9f4db9-bf52-4450-aab5-75bb90fd2100",
    "message": "bash      11     root  255r   REG                8,1      176 3480843 /
    "level": "debug"
}
```

distilled down to remove extraneous data:

```Bash
"message": "bash     10     root ... /tmp/dropper.sh"

"message": {
    "user":"root",
    "process":"/bin/bash /tmp/dropper.sh",
    "pid":"10"
}
```

So the `lsof` and pseudo-`ps` commands pick up on the malicious `dropper.sh` script, while the exploited `ps` misses it.

What does the authentication cycle look like?

The first thing any sensor or client does when communicating with the Sensing API is walk through the certificate issuance process. Once the sensor/client has a public/private key pair, all further communcations with *any infrastructure* use mutual TLS authentication.

When such a client/sensor connects to the Sensing API, the client certificates go through an authentication and validation process. The API logs this transaction, and the API request will only progress if all authentication and validation conditions are met:

```Bash
api_1              | [info] PUT /api/v1/sensor/419fe31d-c393-4219-bdd9-ff22c2e98304/re
api_1              | % authenticating request
api_1              |   ? calculating datetime boundaries
api_1              |     @ not_before(180108153900Z)
api_1              |     @ not_after(190108153900Z)
api_1              |     @ certificate not self-signed
api_1              |     @ trust chain validates
api_1              |     @ certificate key size(4096) large enough
api_1              |     + certificate constraints met
api_1              |   + authenticated CN(828ad1fa6a57) with certificate(...)
```

Note Bene:

One of the possible authentication mechanisms that is missing from the validation process at this time is checking the Certificate Authority *Certificate Revocation List* to ensure that the certificate provided by the client is still valid and hasn't otherwise been revoked. The primary reason for this is the efficiency (or lack thereof) in the existing tooling in both the core CA (`cfssl` in our case) and the language tooling for reviewing CRLs.

This is not a problem with Elixir/Erlang in particular, but a wider problem with the format of CRLs during distribution, and tooling fast enough to deal with a contantly rotating set of revocations on the order of hundreds of thousands to millions of revocations. Current revocation distrubutions are done as lists of certificates, and libraries do lookups typically by walking the revocation list, which is far too slow to do during the HTTP(s) request cycle. A faster and more efficient CRL processing mechanism that is capable of doing full CRL validation on the order of < 1ms is part of the future work for Two Six Labs.

What does the authentication/registration cycle look like?

Each sensor walks through the full authentication cycle with the Sensing API, during which it recieves:

- the **Certificate Authority Public Key** which is used during certificate validation.
- An **RSA 4096 bit Private Key** and **Certificate Signing Request** unique to the sensor
- An **RSA 4096 bit Public Key**

After the authentication cycle, the full sensor registration cycle begins, during which the Sensing API verifies that the sensor is reporting and responding on a specific port, to complete the cycle.

These two cycles are separated out as a measure of *security in depth*. In particular, a sensor does not recieve its unique Kafka log topic until the registration cycle, which is conducted completely over mutually authenticated TLS. The unique Kafka topic is where logs for the sensor will be streamed, and the topic names are randomized UUID strings, making the topics difficult, if not impossible to guess or walk.

Looking at just the logs from a single sensor during this double cycle shows the process unfolding:

```
    + Sensing API is ready
  @ Retrieving CA Root public key
  + got PEM encoded certificate
  < PEM written to [/opt/sensors/lsof/certs/ca.pem]
  @ Requesting Private Key and Certificate Signing Request from the Sensing API
  + got PEM encoded certificate
  < PEM written to [/opt/sensors/lsof/certs/ca.pem]
  % private key fingerprint(d1:64:bc:b2:66:8c:12:88:7e:19:56:cc:fb:6b:8c:cb:5d:64:bd:9
  % CA http-savior challenge url(http://d174dc3b8ddf:11030/path/to/challenge) and toke
  @ Requesting Signed Public Key from the Sensing API
 -> connection from  ('172.19.0.5', 51225)
[info] got [GET] request path [/path/to/challenge]
[http_01_savior_challenge] > handling request
  | sending HTTP-01/SAVIOR certificate challenge response
 <- connection closed
  + got a signed public key from the Sensing API
  @ waiting for registration cycle
registering with [https://api:17504/api/v1/sensor/fbe4cee0-4535-41c5-b9fc-5976326e31b5
  client certificate public(/opt/sensors/lsof/certs/rsa_key.pub), private(/opt/sensors
 -> connection from  ('172.19.0.5', 49575)
[info] got [GET] request path [/sensor/fbe4cee0-4535-41c5-b9fc-5976326e31b5/registered]
[route_registration_ping] > handling request
  | sensor ID is a match
 <- connection closed
Registered sensor with Sensing API
```

Bash

What's actually happening in those inspect and stream calls?

For convenience, we've wrapped the `stream` and `inspect` command lines in simple Bash scripts. Unpacking them requires an up front explanation of *why* the `virtue-security` tool is wrapped in a Docker container, though.

To the Sensing API, the `virtue-security` command line tool is just another client, like a sensor. Despite being a short lived interaction with the API (unlike the repeated long term interactions of a sensor), the `virtue-security` command still needs to provide a Client Certificate during API calls. The same script used by the Sensing API to retrieve its own certificates during infrastructure startup is used by the `virtue-security` container to generate certificates. Making sure this command is properly run, in the right order, with the right flags, and with certificates placed in a standard place, is much easier done within a controllable Docker container with a built in run script. So, `virtue-security` sits within a container, and a

utility script called `dockerized-run.sh` passes commands into a new container each time it's invoked. The `dockerized-run.sh` command can be used as a proxy for the `virtue-security` command, with all of the command parameters passed intact to the `virtue-security` script within the container.

dockerized-stream.sh

The streaming command uses **targeting** selectors to desribe who, or what, we want to stream sensor data about. In this case, we'll use the following parameters:

- **target** the user `root`
- show all messages of `debug` or higher level
- include all messages since `100 minutes ago`
- keep the stream open and `follow` messages as they arrive

which looks like:

```Bash
./bin/dockerized-run.sh stream --username root --filter-log-level debug --follow --sinc
```

dockerized-inspect.sh

Inspecting the running system also uses a `username` **targeting** selector, returning all of the sensors, across any Virtues, observing the targeted user:

```Bash
./bin/dockerized-run.sh inspect --username root
```

Other commands

monitor

The Sensing API publishes its high level actions (and a heartbeat) to a Command and Control channel. The Kafka topic for this channel can be retrieved with the API call `/control/c2/channel`, which requires full authentication to retrieve.

Monitoring the C2 channel looks like:

```Bash
./bin/dockerized-run.sh monitor
[dockerized-run]
Getting Client Certificate
Running virtue-security
%% monitoring Sensing API C2 on Kafka
    topic=api-server-control
    bootstraps=[kafka:9455]
💗 - (timestamp=2018-01-08 20:41:00.004138Z)
💗 - (timestamp=2018-01-08 20:42:00.001936Z)
💗 - (timestamp=2018-01-08 20:43:00.006825Z)
💗 - (timestamp=2018-01-08 20:44:00.002473Z)
💗 - (timestamp=2018-01-08 20:45:00.001345Z)
- DEREGISTRATION - (timestamp=2018-01-08 20:45:00.009343Z, sensor_id=854f8298-4a51-457
- DEREGISTRATION - (timestamp=2018-01-08 20:45:00.009897Z, sensor_id=42067c8e-5cef-450
💗 - (timestamp=2018-01-08 20:46:00.003055Z)
💗 - (timestamp=2018-01-08 20:47:00.001463Z)
💗 - (timestamp=2018-01-08 20:48:00.001588Z)
💗 - (timestamp=2018-01-08 20:49:00.001838Z)
💗 - (timestamp=2018-01-08 20:50:00.001995Z)
💗 - (timestamp=2018-01-08 20:51:00.001699Z)
+ REGISTRATION - (timestamp=2018-01-08 20:51:51.724592Z, sensor_id=b07c0910-dc35-412d-b
+ REGISTRATION - (timestamp=2018-01-08 20:51:52.593709Z, sensor_id=b7a35817-c18b-4199-9
+ REGISTRATION - (timestamp=2018-01-08 20:51:53.699897Z, sensor_id=27a4d5ba-8260-4341-8
💗 - (timestamp=2018-01-08 20:52:00.002028Z)
```

In this C2 sample we see a regular heartbeat message from the API, as well as **deregistration** and **registration** messages. Sensors make every attempt to gracefully deregister themselves from the API, but aren't always successful. In those cases, any sensor that hasn't synchronized with the API within a reasonable time frame is force deregistered.

While the `virtue-security` tool is providing a readable summary of each C2 message, the messages that are transmitted on C2 are all well-formed JSON documents.

version

Built into `virtue-security` is a simple action that serves as a test for a live API as much as it is a dump of the version of the local `virtue-security` script and the version of the remote API. Running the `version` action looks like:

```
> ./bin/dockerized-run.sh version
[dockerized-run]
Getting Client Certificate
Running virtue-security
virtue-security(version=1)
sensing-api(version=2017.11.1)
```
Bash

add-target

One of the `bin` command available adds a new *target* instance to the running Virtue environment. This command takes one argument, which is the container name for the new instance:

```
 ./bin/add-target.sh target_3
86efc1f19573dcf58ca81aa0c53c05a6c5c10747e8962aae0d7e2a23dc52675e
```
Bash

The output of this command isn't terribly interesting by itself. You can watch the process of the new instance interacting with the infrastructure through the `monitor` or `stream` calls, or by watching the logs of the infrastructure. In the later case, you'll only see the infrastructure side (Sensing API, Kafka, CA) of the authentication and registration cycles - the logs for this new instance aren't automatically added to the logging output of the original infrastructure `docker-compose` call.

Sensors

So what goes into a sensor? How are sensors incorporated into a target instance?

Writing a Sensor

Sensors so far are wrapped in a simple Python script and accompanying library that abstract away the asynchronous aspects of authentication, registration, and communications. With this construct Python is used to parse the output or logs of a sensor, which can be any type of application, logger, driver, or system component. The sensor does the heavy lifting of data acquisition, and Python does what it's good at: parsing semi-structured data and shipping it off over the network.

The sensors implemented so far each use the shared support library, with each individual sensor needing a custom Python script of less than 100 lines. A thinned out example of monitoring the system `ps` command looks like:

```python
async def ps(message_stub, config, message_queue):

    repeat_delay = config.get("repeat-interval", 15)
    ps_path = wrapper.opts.ps_path
    ps_args = ["-auxxx"]

    print(" ::starting ps (%s) (pid=%d)" % (ps_path, self_pid))
    print("    $ repeat-interval = %d" % (repeat_delay,))

    full_ps_command = [ps_path] + ps_args

    while True:

        # run the command and get our output
        p = subprocess.Popen(full_ps_command, stdout=subprocess.PIPE)
        got_header = False

        async for line in p.stdout:

            # ... parse the PS command output

            # report
            logmsg = {"timestamp": datetime.datetime.now().isoformat(),"level": "info"
            logmsg.update(message_stub)

            await message_queue.put(json.dumps(logmsg))

        # sleep
        await sleep(repeat_delay)

if __name__ == "__main__":
    wrapper = SensorWrapper("ps-sensor", ps)

    wrapper.argparser.add_argument("--ps-path", dest="ps_path", default="ps", help="Pat
    wrapper.start()
```

The custom portion of the sensor script handles parsing the log messages from `ps`, adding them to a message queue that is automatically handled by the support library.

Building Targets

For the Docker containers used for development of the sensing infrastructure, sensors and targets are defined with simple JSON documents describing what comprises each sensor, and which sensors should be included

in each target.

Each sensor defines a `sensor.js` file that our tooling can detect. The `sensor.js` definition file for the `ps` sensor looks like:

```JavaScript
{
  "name": "ps-sensor",
  "version": "1.0",
  "target_folder": "ps",
  "files": [
    {
      "source": "sensor_ps.py",
      "dest": "sensor_ps.py"
    }
  ],
  "startup_script": "run_ps_sensor.sh",
  "requirements_files": ["ps_requirements.txt"],
  "required_sub_directories": ["certs"],
  "apt-get": []
}
```

And defines which files, Python libraries, directories, and OS level packages are required for the sensor. Each sensor also defines its own run time script.

Each target defines a `target.json` file that describes how to stage all of the files for selected sensors, such that a properly crafted *Dockerfile* can locate and install all of the sensors. Our `target.json` file for the **demo-target** looks like:

```JavaScript
{
  "name": "site-visit-demo-target",
  "library_directory": "./sensor_libraries",
  "sensors_directory": "./sensors",
  "requirements_directory": "./requirements",
  "startup_scripts_directory": "./sensor_startup",
  "sensors": [
    "lsof-sensor",
    "ps-sensor",
    "kernel-ps-sensor"
  ]
}
```

The majority of the fields define where to place certain sensor files during staging and installation. The last key

of this example, `sensors`, is a list of what sensors should be staged for this target.

A global build script is used to find all of the sensors and targets, validate their definitions (and verify all of their files and settings), and move sensor files into place so that targets can be built with Docker:

```Bash
 ./bin/install_sensors.py
Running install_sensors
  wrapper(./sensors/wrapper)
  sensors(./sensors)
  targets(./targets)

Finding Sensors
  + ps-sensor (version 1.0)
  + lsof-sensor (version 1.0)
  + kernel-ps-sensor (version 1.0)

Finding Targets
  + site-visit-demo-target

Installing 3 sensors in target [site-visit-demo-target]
  ~ Preparing directories
    - removing [sensors_directory] (./sensors)
    + creating [sensors_directory] (./sensors)
    - removing [requirements_directory] (./requirements)
    + creating [requirements_directory] (./requirements)
    - removing [startup_scripts_directory] (./sensor_startup)
    + creating [startup_scripts_directory] (./sensor_startup)
    - removing [library_directory] (./sensor_libraries)
    + creating [library_directory] (./sensor_libraries)
  + installing Sensor Wrapper library
    + library files
    + requirements.txt file
  + installing lsof-sensor (version 1.0)
    + run time files
    + run time directories
    + startup script
    + requirments.txt files
  + installing ps-sensor (version 1.0)
    + run time files
    + run time directories
    + startup script
    + requirments.txt files
  + installing kernel-ps-sensor (version 1.0)
    + run time files
```

```
    + run time directories
    + startup script
    + requirments.txt files
  + Creating requirements_master.txt to consolidate required libraries
  + Building support library install script
    % Scanning for pip install targets
    + Writing install script
  + Building sensor startup master script
    + 3 startup scripts added
  + Finding apt-get requirements for installed sensors
    = Found 1 required libraries
```

With the sensors moved into place, a standard Dockerfile can be used to build and run the entire target, including all sensors.