Paper SAS3601-2019

Using SAS® Viya® to Implement Custom SAS® Analytics in Python: A Cybersecurity Example

Damian Herrick, SAS Institute Inc.

ABSTRACT

This session is designed for data scientists looking to develop custom production analytics solutions backed by SAS® analytic capabilities. First, we will describe an interactive exploration and discovery environment based on the SAS® Viya™/ SAS® Cloud Analytic Services (CAS) architectural foundation. Next, using cybersecurity as an example, we will demonstrate how SAS analytics can be implemented in Python in a production implementation of the environment.

INTRODUCTION

We demonstrate two analytics environments developed for the SAS® Cybersecurity solution that combines SAS and open source tools. The first, an exploration and discovery environment, supports investigation of data and hypothesis testing. Insights gained on static data in this environment can then be implemented in a production environment that supports analytic calculations on streaming data. The results of these calculations feed an alerting system that highlights anomalous events found in the streaming data. Using these environments, data scientists implement SAS analytics in Python. Though designed for cybersecurity, the core environment is extensible to other analytic scenarios.

The environment's foundation is built with SAS® Viya. A custom Python module implements 78 predefined cybersecurity analytics that orchestrate SAS® Cloud Analytic Services (CAS) via the SAS developed, open source python-swat module. Analytics are easily tuned and configured using JSON formatted support files. A data scientist can create custom analytics with the same Python framework that implements the preconfigured analytics.

Alongside this paper, we share a sample version of the python module that readers can use as a base to develop their own analytics that run in CAS. The provided sample builds a Docker container that delivers the python module, SAS python-swat, sample Jupyter notebooks, and a sample dataset of Windows Host Events from the Los Alamos National Laboratory Unified Host and Network datasets.

We chose the language, tools, and architecture presented here specifically for the quality, popularity, and ease of use they provide. SAS Viya provides world class in-memory analytics; Python is popular among data scientists and easily scales to enterprise deployments; and Jupyter is a standard, accepted environment for prototyping data science solutions. Our Python module simplifies interaction with CAS by abstracting connection, management, and analytic details one level higher from the SWAT framework. The abstraction allows the module to operate in a production environment and perform calculations in response to data availability, instead of in response to a user-driven request. The use cases for such an analytic architecture range from instructional delivery in education through production data science solutions. The SAS Viya / Python / Jupyter combination demonstrated here is an exciting option for data scientists looking to develop custom solutions backed by SAS analytics.

DEVELOPMENT ENVIRONMENT CONFIGURATION AND SETUP

To facilitate easier development, a simplified, containerized version of the analytics environment referenced throughout this paper is found in the SAS Global Forum 2019 Github repository located at:

https://github.com/sascommunities/sas-global-forum-2019

All code related to this paper is located in the 3601-2019-Herrick folder:

https://github.com/sascommunities/sas-global-forum-2019/tree/master/3601-2019-Herrick

Follow the steps below to replicate the environment on your local machine. The instructions assume you have access to both a SAS Cloud Analytic Services installation and a terminal emulator on your local computer. All shell scripts provided are written in bash. In MacOS or Linux, this is easily accessible using the terminal shell bundled with the operating system. In Windows, you should use a terminal emulator such as MobaXTerm, and run a bash shell.

1. Install Docker on your local system. The Docker binaries and installation instructions for your specific OS are located at: https://www.docker.com/products/docker-desktop
2. Identify the remote CAS server you plan to connect to from your Docker container. Make a note of the hostname for later use.
3. Create a `.authinfo` file with the host, port (5570), username, and encrypted password. Make sure the file has the proper permissions by running `chmod 600 .authinfo`
4. Place it in the `3601-2019-Herrick/security/` directory.
5. While logged into your CAS server, locate the `cascert.pem` (this should be in the CAS user's home directory), and the certificate located at `/opt/sas/viya/config/etc/SASSecurityCertificateFramework/cacerts/vault-deployTarget-ca.crt`. Copy both of these files from the remote CAS server to the `3601-2019-Herrick/security/` directory on your computer.
6. Once you've copied the security files to right location in the repository, everything is in place to build the two containers. The containers are hierarchical – `centos/jupyter` derives from `centos/base`. They provided this way to avoid a full rebuild and compilation from source of Python and Sqlite each time the Jupyter container is modified.
    a. Navigate to the root of our repository.
    b. Build the base container by running the script `./build_base.sh`. The base container builds from a CentOS 7 base image hosted at the Docker Hub. It uses the `yum` package manager to install and update the packages needed by SAS Cybersecurity. The build process also downloads Python 3.7.2 and Sqlite, then compiles both tools with the required settings for the SAS Cybersecurity solution.
    c. Following completion of the base container build process, build the Jupyter container similarly by running the script ./`build_jupyter.sh`. The Jupyter container installs the specific items needed to run the analytic environments needed for this session. Specifically, it:
        i. installs the security certificates and settings needed to connect to CAS.

ii. Installs all Python packages needed to run a Jupyter server, SWAT, and cybersecurity code.

iii. Creates necessary users and assign privileges.

iv. Starts the Jupyter server.

Running the container requires that you map four additional directories – `/notebooks/`, `/datasets/`, `/logs/`, and `/custom/` – from your local system to the container. These directories mount as Docker volumes. Mounting them this way allows modification of their contents without rebuilding the container, and insures that changes persist after the container is shut down. Sample directories are provided in the Github repository and should be used if you are following along with this paper. Those directories are located in:

- `3601-2019-Herrick/notebooks/`
- `3601-2019-Herrick/data/`
- `3601-2019-Herrick/logs/`
- `3601-2019-Herrick/custom/`

If you choose to use your own notebooks and data directories, make sure they are subfolders of a single common directory.

After the Jupyter container is built, launch the container with the following command:
`./run_container.sh /full/path/to/your/repo/3601-2019-Herrick jupyter`

The `run_container.sh` script automatically mounts the four directories mentioned above:

```
FOLDER=$1
IMAGE=$2

CONTAINER=`docker run -d --rm -p 8888:8888 \
           -v $FOLDER/notebooks/:/home/ds/notebooks \
           -v $FOLDER/data/:/home/ds/datasets \
           -v $FOLDER/logs/:/home/ds/logs \
           -v $FOLDER/custom/:/home/ds/custom \
           centos-ds/$IMAGE`
echo $CONTAINER
```

If the container successfully opens, its identifier prints in the terminal. The identifier is a GUID and should appear similar to:
`8d5e34128ef87decfd8fe315b145fd7f97a9f22626c68512b56fccfbec6822f3`.

Open a new browser tab or window and navigate to: `http://localhost:8888`. The Jupyter server homepage loads and shows all notebooks in the corresponding local directory mapped to the container. Clicking on any of them opens a Jupyter notebook in a new browser tab.

Test the connection to your CAS server using the `00_DisplayActionSets.ipynb` notebook. Enter the name of your server in the statement `conn = swat.CAS("<hostname-for-CAS-server>", 5570)` in the third cell. Execute the cells in order from top to bottom; if all of the security certificates were copied to the container properly, the container will connect to CAS and the remaining cells will display metadata information about your CAS server. Display 1 shows the CASLIBs available in the CAS server used for this paper. Display 2 shows the

actionsets loaded in the cybersecurity CAS server at the start of this demonstration. Additionally, the sample notebook has the following calls for information about the server:

- Server status: `conn.builtins.serverstatus()`
- License information: `conn.builtins.getlicenseinfo()`
- Available nodes: `conn.builtins.listnodes()`
- Current sessions: `conn.listsessions()`

```
In [4]: conn.caslibinfo()
```

Out[4]: **§ CASLibInfo**

	Name	Type	Description	Path	Definition	Subdirs	Local	Active	Personal	Hidden	Transient
0	CASUSER(daherr)	PATH	Personal File System Caslib	/home/daherr/casuser/		1.0	0.0	1.0	1.0	0.0	1.0
1	Formats	PATH	Stores user defined formats.	/opt/sas/viya/config/data/cas/default/formats/		0.0	0.0	0.0	0.0	0.0	0.0
2	ModelPerformanceData	PATH	Library for Model Management performance objects.	/opt/sas/viya/config/data/cas/default/modelMon...		0.0	0.0	0.0	0.0	0.0	0.0
3	Models	PATH	Stores models created by Visual Analytics for ...	/opt/sas/viya/config/data/cas/default/models/		0.0	0.0	0.0	0.0	0.0	0.0
4	Public	PATH	Shared and writeable caslib, accessible to all...	/opt/sas/viya/config/data/cas/default/public/		0.0	0.0	0.0	0.0	0.0	0.0
5	Samples	PATH	Stores sample data, supplied by SAS.	/opt/sas/viya/config/data/cas/default/samples/		0.0	0.0	0.0	0.0	0.0	0.0
6	SystemData	PATH	Stores application generated data, used for ge...	/opt/sas/viya/config/data/cas/default/sysData/		0.0	0.0	0.0	0.0	0.0	0.0
7	WH	PATH	Windows Host Events	/home/datasets/LANL/WH/		0.0	0.0	0.0	0.0	0.0	0.0

elapsed 0.001s · user 0.000963s · mem 0.672MB

Display 1: Available CASLIBS on the cybersecurity research CAS server.

```
In [9]: asinfo = conn.actionsetinfo(all=True)
        asinfoUnloaded = asinfo.setinfo[asinfo.setinfo.loaded == 0].loc[:, 'actionset':'label']
        asinfoLoaded = asinfo.setinfo[asinfo.setinfo.loaded == 1].loc[:, 'actionset':'label']

In [10]: asinfoLoaded
```

Out[10]:

Action set information

	actionset	label
1	accessControl	Access Controls
9	builtins	Builtins
13	configuration	Server Properties
16	dataPreprocess	Data Preprocess
17	dataStep	DATA Step
53	percentile	Percentile
65	search	Search
68	session	Session Methods
69	sessionProp	Session Properties
72	simple	Simple Analytics
78	table	Tables

Display 2: Loaded actionsets available in the CAS server at runtime. Unloaded actionsets can be listed similarly by referencing the asinfoUnloaded variable.

LOS ALAMOS UNIFIED HOST AND NETWORK DATASET

The example data provided for this paper is a small subset of Windows Host Events taken from the Los Alamos National Laboratory Host and Network Dataset (Turcotte, Kent, and Hash). The complete dataset contains approximately 90 days of enterprise network traffic collected from the internal network at the Los Alamos National Laboratory (LANL). Two data sources - Netflow and Windows Host Events - are provided by Los Alamos. We use the Windows Host Events dataset as the source for our examples in this paper.

Detailed data descriptions of all fields in these datasets is beyond the scope of this paper. Refer to Turcotte, Kent, and Hash for full descriptions of the datasets.

Los Alamos provides individual files as bzip2 compressed full-day captures for each of the 90 days. Windows Host Events files decompress into JSON files whose size ranges between 10 and 15GB, each containing approximately 65 million lines of data. Netflow files decompress into CSV formatted files and have fewer columns than the Host Event files. They are smaller than the Host Event files – full day files range between 5 and 8 GB, but capture many more lines of data – approximately 115 million lines.

SAS Cybersecurity expects data in CSV format, and processes files hourly. In order to conform to that standard, the full-day LANL files are pre-processed and output as individual hourly CSV files. The pre-processing workflow is as follows:

1. Download a single day from the remote Host Events or Netflow repository.
2. Decompress into either JSON or CSV.
3. Load the full file into a Pandas dataframe.
4. Extract each hour from the timestamps provided.
5. Write each hour as a CSV file in an "hours" subdirectory.

Due to the size of the individual day files, they are not provided in the Github repository. Instead we provide a single Windows Host Events hourly file in the `/datasets/` directory. The file provides the source data for the examples in this paper.

SAS CYBERSECURITY OVERVIEW

The most recent version of SAS Cybersecurity provides a much more flexible and scalable architecture than earlier releases. Thanks to a modular architecture, individual portions of the solution can be installed and updated without affecting the entire installation. This design introduces a key advantage – the ability to add new or improved analytics without reinstalling the entire stack. Additionally, SAS Cybersecurity is built entirely on the SAS Viya architecture. Both SAS Cloud Analytics Services (CAS) and SAS Event Stream Processing are key components of the analytics and data ingest tiers.

SAS Cybersecurity's analytics tier is diagrammed in Figure 1. This tier represents a **production** installation. In this setting, all analytics are pre-built and calculated when data arrives from an ingest pipeline. Several individual services work together to calculate analytics and deliver results to the client tier. An Analytics Manager monitors the environment, and when new data arrives at the head of the tier, fires a trigger that kicks off an analytic processing job (represented by the DAG section in Figure 1). Analytics calculation takes place in CAS, and results are persisted to disk. An Analytics Publisher transforms and augments the results, then pushes the finalized analytic events to the Client Tier of the solution.

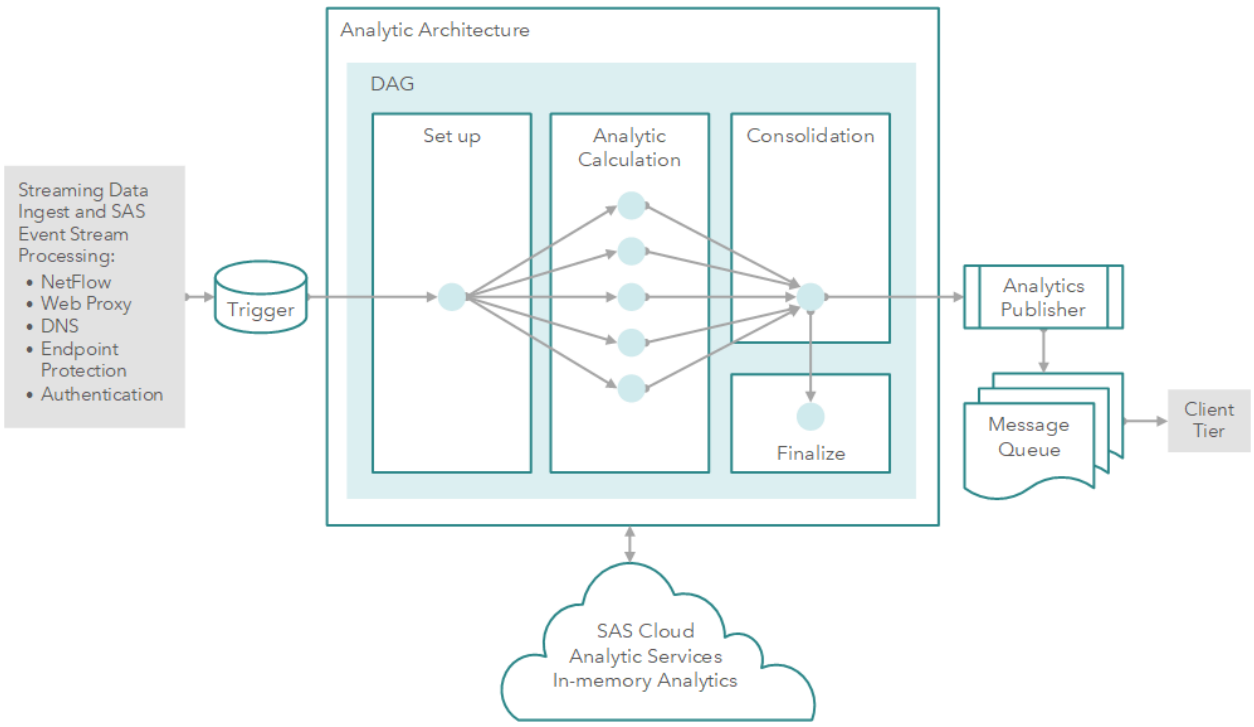

Figure 1: SAS Cybersecurity analytics tier architecture.

To support development of new analytics in a customer setting, a separate **exploration and discovery** environment can be installed. The exploration and discovery environment is diagrammed in Figure 2; the Docker container provided in the associated Github repository is a simplified version of the solution offered to SAS Cybersecurity customers.

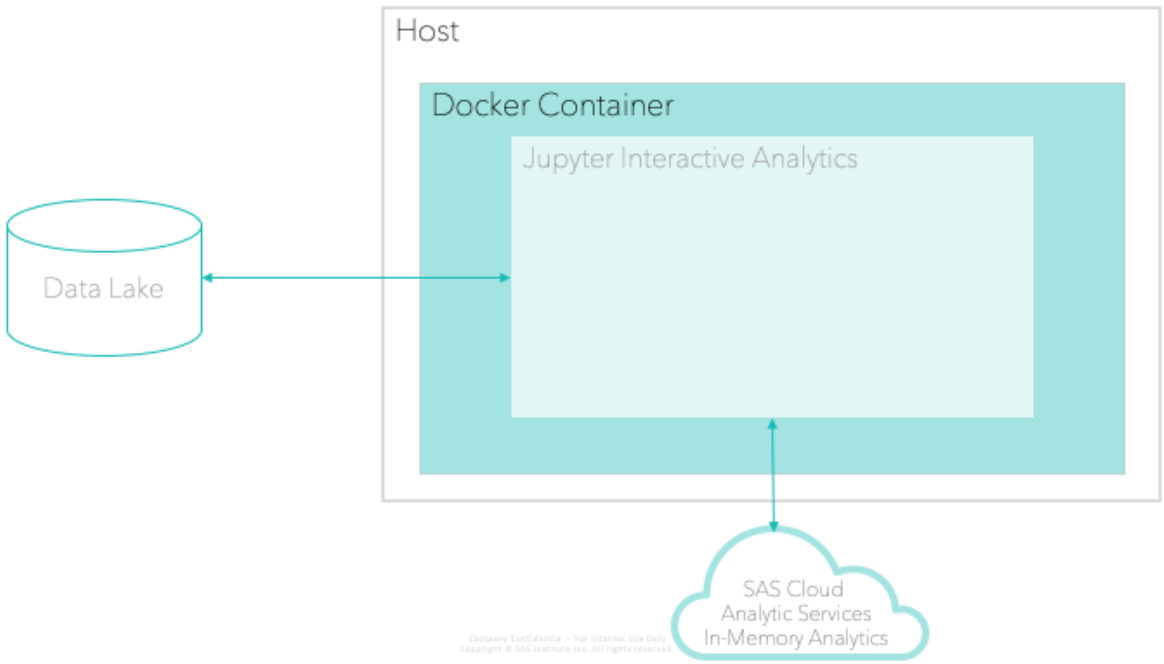

Figure 2: SAS Cybersecurity Analytics Exploration and Discovery architecture.

DESIGN CONSIDERATIONS FOR THE SAS CYBERSECURITY PYTHON CLIENT

The newest version of SAS® Cybersecurity 3.1 incorporates several important advances from earlier versions:

1. Its architecture is designed to be completely modular, facilitating easy updates of any single portion of the solution without a full reinstall.
2. It relies fully on SAS® Viya, using SAS® Cloud Analytic Services (CAS) to calculate cybersecurity analytics, and SAS® Event Stream Processing (ESP) to handle ingest, enrichment, and delivery of streaming data to the analytics tier.
3. The analytics tier (the focus of this paper) was completely redesigned and rewritten in Python.

The design decision to use CAS instead of SAS® LASR™ Analytic Server opened exciting new options for analytics development. Specifically, CAS provides APIs accessible through multiple languages. This functionality allowed the team to factor in a wider range of considerations and requirements when developing the architecture. Feedback from prior proof-of-concept (POC) engagements provided significant input during the design phase. Specifically, even though most target customers were already SAS customers, most security teams (and their data scientists) often were not SAS users. In fact, the practice of cybersecurity data science is rapidly maturing. Most data scientists expressed familiarity with R and Python, and expressed the desire to use one of those languages for model development and testing. Instead of forcing data scientists to learn a new language, Viya's open architecture allows us to easily meet this customer need.

SAS Cybersecurity 3.1 also supports more data sources in a standard install than earlier versions supported. More data sources imply a need for more analytics. Cyber Analytics Research and Development also identified multiple new analytic techniques that are included in this version. Between the additional data sources and new analytic techniques, this solution delivers 78 individual analytics in a standard install – up from 22 in prior versions.

These high-level design considerations led the team to develop a Python architecture that is lightweight, modular, and supports the development of new analytics – both through the standard development process as well as custom, field-developed analytics. Analytics are dynamically loaded at run time, which keeps the memory load of individual processes lighter than if all analytics were loaded.

The analytics tier runs in production with both a dynamic configuration model and a custom designed workflow manager. This architecture allows multiple analytics to run in parallel, meeting the requirement that all analytics complete calculation inside of a 60-minute window.

"Flexible analytics" was a primary design goal of the analytics tier of SAS Cybersecurity. In order to achieve this, the team decided to deliver analytics through a class module that wrapped the behavior of the SAS-delivered python-swat module. Python-swat provides full interaction with CAS, including session management, access to all actionsets licensed to the user, and data transfer and management. While powerful, there is also a learning curve associated with it.

Additionally, SAS Cybersecurity supports pre-built analytics that are ready to run as soon as ingested data begins streaming. Bundling those analytics into a higher-level module facilitates easier installation and troubleshooting.

Providing pre-built analytics and encapsulating python-swat function in a single module met most of the flexible analytics design requirements. In order to fully support flexible

analytics, the module supports custom built analytics – cybersecurity analytics that are developed and deployed by customers or the SAS services team outside the standard development process. Python's support for dynamic class loading at run time allowed us to meet this requirement.

CLASS HIERARCHY

All analytics used by the SAS Cybersecurity Python Analytics Module derive from the CasAnalytic class. This class manages the connection and interaction with the CAS server. Several support classes derive from the CasAnalytic class; they facilitate easier interaction for specific repetitive tasks required by the Analytics tier in production. Finally, general analytic categories derive from the CasAnalytic class – such as the DstAnalytic shown in Figure 3. Specific analytics from that family then derive from this class.

The class hierarchy removes much of the challenge of interacting with CAS – connecting, session management, and actionset loading. It also insures that new data formats are output in the proper fashion for downstream processing by the Analytics Publisher and the Client tier. The extensive utility provided by these classes allows independent developers to build their own custom analytics classes, focus on the individual calculations required by their analytic, and then integrate with the production user interface.

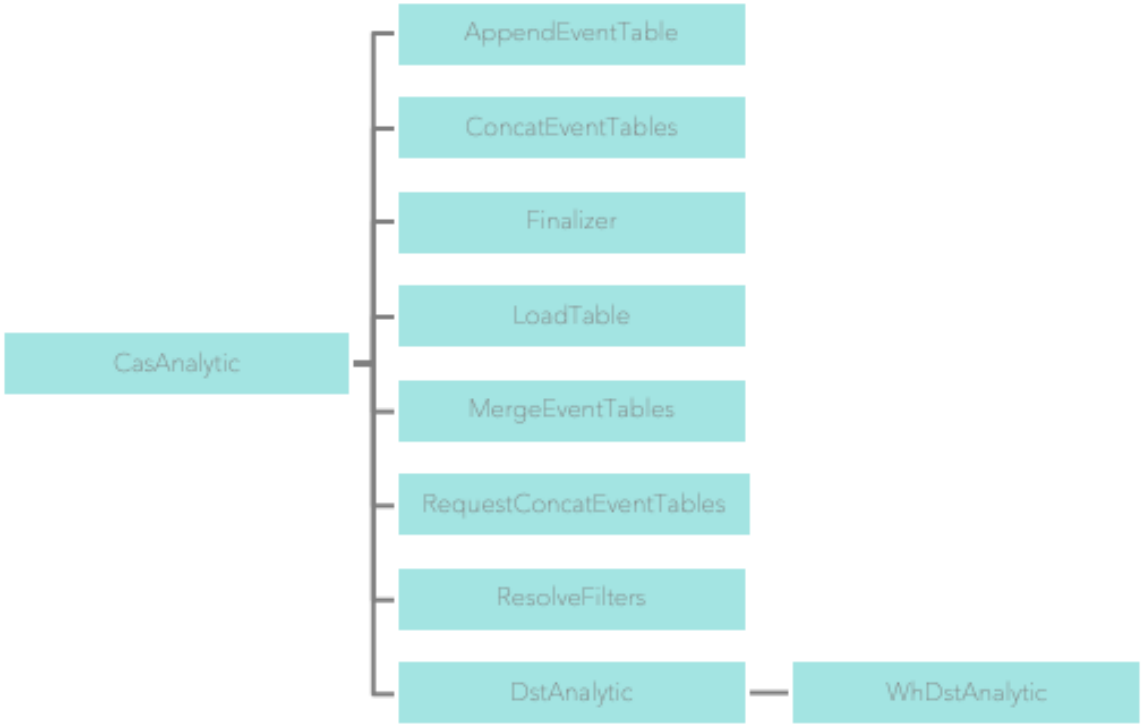

Figure 3: A simplified view of the class hierarchy that supports SAS Cybersecurity analytics.

IDENTIFYING FAILED LOGIN ATTEMPTS BY INDIVIDUAL USERS

To better understand the relationship between the two environments, the remainder of this paper discusses how to test a hypothesis in the exploration and discovery environment, then implement the same analytic as a production analytic under the `sascyber` module.

The provided sample data in the file `/home/ds/datasets/WH/wls-day_02_hr13.csv` consists of one hour of Windows Host Events data. The Jupyter notebook `01_WindowsHostExploratoryAddToCAS.ipynb` describes:

- How to import the data into CAS
- How to conduct simple exploratory analysis
- How to persist the CASTable in the .sashdat format. Taking this step simplifies loading the table into memory during future CAS sessions.

After loading the data into CAS, we determine the number of records in this dataset by simply calling len(hostData). In our case, the provided hour contains 354,758 events.

Table 1 shows a summary of the columns in our data, with the number of distinct values as well as the number of missing values. Note that most fields are missing a significant amount of data, but the EventID field is complete, and only 14 values are missing in the UserName field.

Column Name	Distinct	Missing
AuthenticationPackage	6	232,670
Destination	321	349,487
DomainName	82	14
EventID	18	0
FailureReason	7	353,231
LogHost	9,408	0
LogonID	198,999	49,030
LogonType	11	177,367
LogonTypeDescription	11	177,367
ParentProcessID	4,762	262,043
ParentProcessName	454	262,043
ProcessID	9,219	251,247
ProcessName	928	251,247
ServiceName	1,506	342,347
Source	9,547	237,860
Status	7	306,641
SubjectDomainName	23	349,256
SubjectLogonID	369	349,440
SubjectUserName	1,390	349,256
Time	3,600	0
UserName	17,981	14
occurredTime	3,600	0

Table 1: Distinct and missing values in the Windows Host Event hour 13 dataset.

For the exercise provided here, we calculate a "collection" analytic. In SAS Cybersecurity, a collection analytic examines a group of devices or users, looking for anomalous activity in a predefined category. The Windows Host Event dataset used here is well-suited for such an analytic.

The dataset contains many unique users and devices. Additionally, the EventID field records specific activities that may be of interest to a security investigator. A comprehensive list of Event IDs can be found at the Windows Security Log Events page (Franklin-Smith). An examination of event IDs on that page shows that ID 4625 indicates an account that failed to log on to a device. Excessive login failures in a given hour could indicate potential malicious activity. We will create an analytic to identify which, if any, users are continually triggering this event ID.

The notebook `02_FailedLogins.ipynb` demonstrates how to test the hypothesis that some users attempt (and fail) logins excessively in a 60-minute period. In order to test the hypothesis, we load our full dataset from the SASHDAT created when we imported the CSV into CAS. Once that's loaded, we want to select only the records with an event ID value of 4625. The following code performs that task:

```
clause = "EventID = 4625"
where = clause.format(table="", not_missing="Is Not Missing")
conn.partition(table={"name": "wls_day_02_hr13", "where":where},
               casout={"name": "WLS_DAY02_HR13_FAILED"})
```

The subtable generated by this code contains 1,507 records. This indicates all account failures that took place during hour 13. We can check the number of user names in the data by calling `UserName.distinct()` on the table; this shows that there are 142 users in our new dataset.

Finally, we determine who the biggest offenders are in our dataset by using a fedsql call to group users by the count of failures, as shown in the following code:

```
query = f"CREATE TABLE WLS_DAY02_HR13_FAILED_USERS AS SELECT UserName,
          count(UserName) as Failures
          FROM WLS_DAY02_HR13_FAILED GROUP BY UserName"
conn.fedsql.execdirect(query)
```

Figure 4 shows that three users (Comp065845$, Administrator, and AppService) account for the majority of failures in this hour. This data point warrants further investigation, which indicates that these accounts are system accounts, not associated with a specific person. This fact leads to an inference that the activity is not malicious, but instead is the result of a misconfiguration on the network.

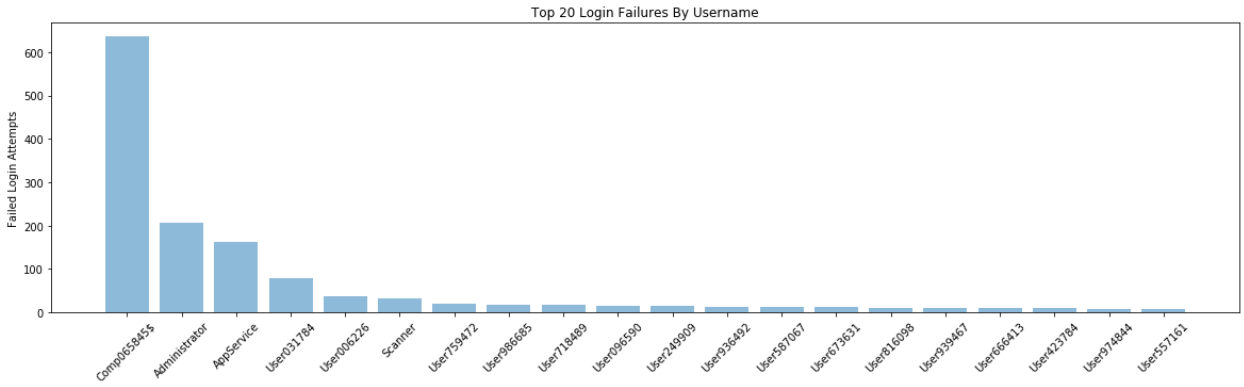

Figure 4: Login failures during the hour 13 dataset. Three system accounts committed the bulk of the login failures during this period.

CONVERTING TO PRODUCTION

After confirming that some users do trigger excessive login failures in an hour, the next step is to translate our findings to the production analytic environment. The translation involves creating a new class for the analytic and a corresponding analytics configuration file with the filter information. The `/custom/` directory in the Github repository contains the source code and configuration files. The class file `whFailedLogins.py` shows that, because we are building a collection analytic, the only additional work needed in code is to specify an analytic field name.

```
from sascyber.analytics.wh.logins.Logins import Logins


class WhFailedLogins(Logins):
    # Default constructor
    def __init__(self):
        super().__init__()
        self.set_analytic_field_name("whFailedLogins")
```

Separately, the `wh_analytics.json` requires more configuration. Due to the length of the file, we refer you to the file itself. The fields that require configuration include:

- `custom_analytics_allowed`
- `custom_analytics_path`
- `input_caslib`
- `import_options.vars`
- `index_vars`
- `aggregation_fields`
- `comparison_fields`
- `filters.EventsFailedLogins`
- `tasks.ResolveEventsFailedLogins`
- `tasks.WindowsHostFailedLoginsAnalytic`

The `03_FailedLoginsProduction.ipynb` notebook demonstrates how to execute a production instance of the Failed Logins analytic. A series of environment variables are set.

These enable the Python module to find all relevant configuration files. After the configuration file is loaded and logging enabled, a DAGServer object is created. This object orchestrates all analytics as defined by the `tasks` section of the `wh_analytics.json` configuration file. Upon completion of the calculations, each user is assigned a score for the calculation based on how anomalous their activity is during the period represented by the dataset.

CONCLUSION

In this paper, we demonstrated how existing SAS® Viya™ solutions, open source Viya connection, and Python language features combine to deliver a flexible and expandable analytics framework. Using Windows Host Events data as an example, we conducted exploratory analysis, tested a hypothesis about failed logins, and used the test results to guide development of a production analytic that will run on streaming data and provide alerts to a security analyst when anomalous activity is detected.

A simplified version of the exploration and discovery analytic environment is provided in an associated Github repository. The repository contains instructions on how to instantiate a Docker container, how to load data into CAS, how to conduct the exploratory analysis, and finally how to test the production analytic. We also provide a simplified version of the SAS Cybersecurity Python Analytic Module.

REFERENCES

Turcotte, M., Kent, A., Hash, C. "Unified Host and Network Data Set". Data Science for Cyber-Security. November 2018:1-22.

Franklin-Smith, Randy. Windows Security Log Events. Available at: https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/default.aspx

ACKNOWLEDGMENTS

The author wishes to thank his colleagues on the Cyber Analytics Research and Development team. In particular, Sean Dyer and Pankaj Telang continually provide insight, feedback, and encouragement that enrich and improve his own work.

RECOMMENDED READING

- *SAS® Viya™: The Python Perspective*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Damian Herrick
SAS Institute, Inc.
Damian.Herrick@sas.com