Core Log

This page is under construction

Overview

This section introduces and explores attributes of our log files that can be used for further feature engineering

There are numerous methods to extract features from log data and files. These methods were applied to our files to understand and determine the best features to extract. The data was taken from the already parsed log files using Drain - a log parsing algorithm created by the researchers at The Chinese University of Hong Kong. Although not perfect, the algorithm separated similar log messages into constant and variable expressions.

Correlation Id Analysis

Approximately 60% of log messages contain correlation id's. Each id specifies a unique transaction that occurs in the log files. Correlation id's can be used to analyze and group log events based on similar transactions. However, UEM developers typically do not trust correlation id's because of its poor implementation and grouping of unrelated events . Additionally, there are approximately 10 000 unique correlation id's, each with 3-5 events. Nevertheless, correlation id's are still necessary for tracking non-device commands.

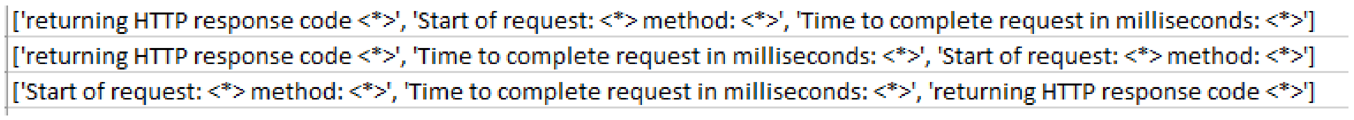
Example: Correlation-Id,84394633-4a5b-48f2-b414-50e6fac790f4

The code for grouping by correlation id is found here in dataset_generator.py with the data in dataset_cleaned.csv

Parameter Analysis

The DeepLog paper for Anomaly Detection proposed a feature extraction method that uses a Long Short Term Memory (LSTM) model that looks at the sequence of parameters over time as well as the time differences throughout the logs. So far, a sequential approach can be applied in two different ways

1. Parameter analysis based on the sequence of events as they appear in the log files. The issue with this approach is that certain log events do not always occur and are often optional. Hence, the sequence of log events are not the same between files even though they preform similar tasks. A filtering method to parse out the optional messages can be applied, however requires extensive searching in the UEM source code.
2. Parameter analysis based on correlation id (transactions). However, due to the nature of correlation id's, the id values are not similar between logs. An id in one file is not the same as another even though they execute the same transaction. Additionally, events within a specific correlation id are not sequential not in any particular order.



```
['returning HTTP response code <*>', 'Start of request: <*> method: <*>', 'Time to complete request in milliseconds: <*>']
['returning HTTP response code <*>', 'Time to complete request in milliseconds: <*>', 'Start of request: <*> method: <*>']
['Start of request: <*> method: <*>', 'Time to complete request in milliseconds: <*>', 'returning HTTP response code <*>']
```

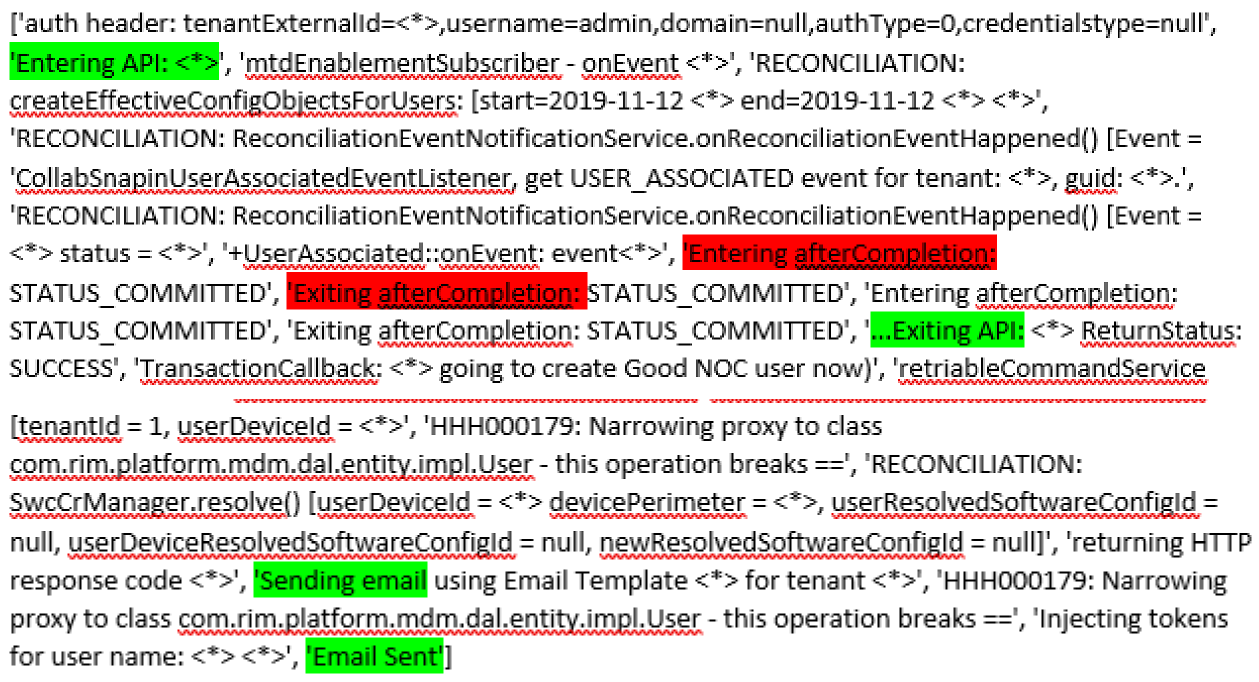
The image above displays events within 3 different correlation id's. Although the sequences are different, this is still considered as non-anomalous. Hence, a sequence approach to parameter analysis may present some roadblocks.

Therefore, a method of clustering parameters based on log key (eventTemplate) was taken with the idea that anomalous parameters occur rarely and is noticeably different than "normal" parameters. The log key based approach was taken because there are approximately 9600 different correlation id's, each with only 3-5 parameters. To group the parameters, log specific values were removed. These values included specific dates and times, tenant id's and specific run times. This resulted in approximately 440 events that contains parameters. Although significantly less than grouping by correlation id, grouping by log key only detects abnormal parameters and not if the parameter fits the transaction.

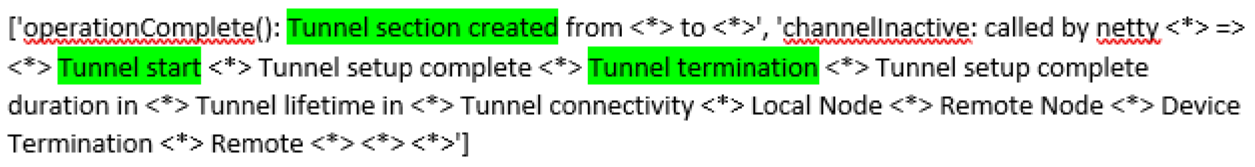
The code for the parameter grouping and cleaning is found in https://gitlab.rim.net/bb-optic/loganomalydetector/tree/parameter_list_cleaning_hv in files Parameters_Structured_to_Template.py and Parameter Cleaning.py with filtered data in UOS_CLOUD_20191112-205133_20191112-215131.log_templates_cleaned.csv

State Ratio Analysis

The paper for Detecting Large-Scale System Problems by Mining Console Logs " by the EECS Department at University of California and Intel Labs Berkeley explored an anomaly detection method using a state ratio vector. Typically, log lines contain various states that generalize the executed action. For example, words like *commiting* or *aborting* display a state in the log line. The research paper suggests creating a ratio vector of the states can be used to flag when abnormal ratios occur. However, the systems they used are the Darkstar online game server and Hadoop File System. These systems contain a lot of log files with states, with 28% and 50% of log messages containing states respectively. In terms of our files, there are no explicit states in the log messages. Additionally, the log messages contains start and finish type messages. Based on this, a simple check if a log event contains a start and finish message was implemented. These messages were determined through scanning the log events and finding grouped identifiers.



['auth header: tenantExternalId=<*>,username=admin,domain=null,authType=0,credentialstype=null', 'Entering API: <*>', 'mtdEnablementSubscriber - onEvent <*>', 'RECONCILIATION: createEffectiveConfigObjectsForUsers: [start=2019-11-12 <*> end=2019-11-12 <*> <*>', 'RECONCILIATION: ReconciliationEventNotificationService.onReconciliationEventHappened() [Event = 'CollabSnapinUserAssociatedEventListener, get USER_ASSOCIATED event for tenant: <*>, guid: <*>.', 'RECONCILIATION: ReconciliationEventNotificationService.onReconciliationEventHappened() [Event = <*> status = <*>', '+UserAssociated::onEvent: event<*>', 'Entering afterCompletion: STATUS_COMMITTED', 'Exiting afterCompletion: STATUS_COMMITTED', 'Entering afterCompletion: STATUS_COMMITTED', 'Exiting afterCompletion: STATUS_COMMITTED', '...Exiting API: <*> ReturnStatus: SUCCESS', 'TransactionCallback: <*> going to create Good NOC user now)', 'retriableCommandService [tenantId = 1, userDeviceId = <*>', 'HHH000179: Narrowing proxy to class com.rim.platform.mdm.dal.entity.impl.User - this operation breaks ==', 'RECONCILIATION: SwcCrManager.resolve() [userDeviceId = <*> devicePerimeter = <*>, userResolvedSoftwareConfigId = null, userDeviceResolvedSoftwareConfigId = null, newResolvedSoftwareConfigId = null]', 'returning HTTP response code <*>', 'Sending email using Email Template <*> for tenant <*>', 'HHH000179: Narrowing proxy to class com.rim.platform.mdm.dal.entity.impl.User - this operation breaks ==', 'Injecting tokens for user name: <*> <*>', 'Email Sent']



['operationComplete(): Tunnel section created from <*> to <*>', 'channelInactive: called by netty <*> => <*> Tunnel start <*> Tunnel setup complete <*> Tunnel termination <*> Tunnel setup complete duration in <*> Tunnel lifetime in <*> Tunnel connectivity <*> Local Node <*> Remote Node <*> Device Termination <*> Remote <*> <*> <*>']

The above images are the identified 'states'.

The code for state anomaly is found here in the file state_ratio_script.py

Perimeter Id Analysis

Perimeter Id's are often used by UEM developers to trace device enrollment. Perimeter id is a globally unique device identifier. Every event associated with the device contains the unique perimeter id, allowing to find all log lines for a device regardless of the server the command is sent to. A majority of UEM user enrollment errors involve the user's device, making the use of perimeter id particularly important. An example perimeter id in the logs is , "perimeterUuid":"5923b68f-5f50-47d8-8aa5-e2d4fc1aa0f7". When filtering and grouping events based on perimeter id, there were a total of 5 different perimeter id's/devices. Each id contained approximately 3000 events each. Also keep in mind that this data is from user enrollment performance tests where numerous enrollments were applied on a single device. In an analysis of more "normal" data, there were approximately 29 000 different perimeter id's with 3-5 events each, which is still less than the number of different correlation id's.

The code for perimeter grouping is found here in the file group_by_perimeter_id.py and the filtered data in UOS_CLOUD_20191112-205133_20191112-215131.perimeter.csv and UOS_CLOUD_20191112-205133_20191112-215131.perimeter_grouped.csv.

Thread Id Analysis

Thread id's indicate when an activity is passed to another thread, allowing for another way to track the sequence of events. However, when a thread execution is complete, it returns into the pool to be reused for future activities. A thread may span across multiple events depending on how they are programmed. Based on current domain knowledge, it is currently unknown how threads are selected from the pool. In grouping by thread id's it is seen that certain threads run once, while others run 3000 times. Again, due to lack of domain knowledge, the reasoning behind this is unknown, but can be determined by discussing with a domain expert and an analysis of the threading functions in the source code.

The code is found here in group_by_thread_id.py and the data is in UOS_CLOUD_20191112-205133_20191112-215131.grouped_by_threadId.csv