SERVER LOG ANALYSIS

August 1, 2018

Scott McCoy
Dr. Todd Wilson

Introduction

New computer systems are created and used daily that affect businesses and institutions both large and small. As the prevalence of enterprise computer systems grows, so too does the profitability of attacking such systems. With the quickly changing culture of software development security holes can easily be introduced as new APIs and frameworks become available but these security risks can go months or years without being patched. This means that often times it is not a question of if a user account will be compromised but when a user will be compromised. When a user of one of these systems is compromised a myriad different types of information is at risk of being stolen by attackers. Libraries risk losing their databases of research papers, businesses risk databases of customer information, and governments risk their national security.

The purpose of this project was to combat such attackers by attempting to identify compromised users and alert relevant information security personel before damage can be done. This was achieved through analyzing the log files of a server in real time in order to identify and respond to any users that exhibit suspicious behavior. The initial motivation for this project came from a university library that was having trouble with compromised user accounts that would download as many academic papers and journals as possible for resale. This would result in the affected databases being locked to the entire university until administrators were able to prove to the database provider that the attacker had been dealt with.

Libraries and Resources

Inspirations

There were several programs that this project drew inspiration from. The first piece of software that was studied was Splunk. Splunk is security information and event management program with a variety of features includeing data collection, searching, indexing, and analysis. The principle feature that drawn from Splunk was the real time analysis of log files with the intent of finding strange data patterns. Splunk many several advanced features such as data visualization and searching functionality but the goal of this project was to be much more light weight than something like Splunk in terms of size and overhead. Another program that inspired this project was IBM QRadar. The feature that we were most interested in replicating with this project was the detection of insider threats. Often times the most common threat in library systems the authenticated so it was important that our program would be able to detect threats from authenticated users. The final program that we drew inspiration from was Graylog. The inspiration from Graylog was taken primarily from its alerting mechanisms. Greylog is capable of sending an email or Slack message, spawning a new server to balance load, and blocking IP ranges in a firewall when a threat is detected.

Libraries and Tools

There were several external libraries and tools that were used in the creation of this project. The first major tool that was used was the Unix tail command. The tail command outputs the last 10 lines of a file to standard output. When used with the -f option, tail continuously outputs any line that is appended to the file. By using tail -f on the log file and then piping the output into the program using the Unix pipe operator. Another library that was used was the Boost library. One of the specific files that were included were the Boost lexical cast file ($< boost/lexical\_cast.hpp >$) which was used for catching errors during the processing of a raw line provided by the tail command. The other was the Boost string file ($< boost/algorithm/string.hpp >$) which was used in the division of the raw line.

DATA STRUCTURES

Userdata and Free List

The Userdata class provided the abstraction for each line of data that was parsed. When a line of data was obtained it was immediately sent to the free list via the getNode function for parsing. This function extracted the needed data from the line (username and IP address), put it into a node from the freelist, and then returned the node. Once this node was obtained by the main loop it was time stamped with the current time added with the appropriate lookahead distances.

Priority Queue

The priority queue was the data structure responsible for managing when data should be expired from each of the map data structures. The implementation of the priority queue that was used was taken from the standard library with the exception of a custom function ($pair\_greater$) to be used as the comparator for the priority queue. The reason for the custom comperator was to allow for the ordering of the priority queue to go from least to greatest rather than from greatest to least and to allow for the usage of a std::pair as the storage value. The values stored in the pair are a ($time\_t$) timestamp that represents the expiration date for that record and a pointer to the record itself. A point to the record was used to allow any modification of record data to affect all other instances of the same record that may be present in the priority queue. This was important in determining when to return Userdata objects to the free list for reuse. The change in the ordering of values in the priority queue was necessary due to how the ($time\_t$) type is represented numerically.

Flood Map

The flood map was responsible for detecting attacks that would cause a flood of requests to appear in the log file over a short period of time such as DoS, spidering, or any type of mass download script. This is achieved by mapping each user to a number of requests recieved from that user over a set period of time (the period of time and the number of requests are determined by values read in from the config file). If this threshold is crossed then an alert is sent to the administrator for further action. The flood lookahead distance will typically be relatively short as the purpose of this data structure is to detect bursts of activity.

IP Map

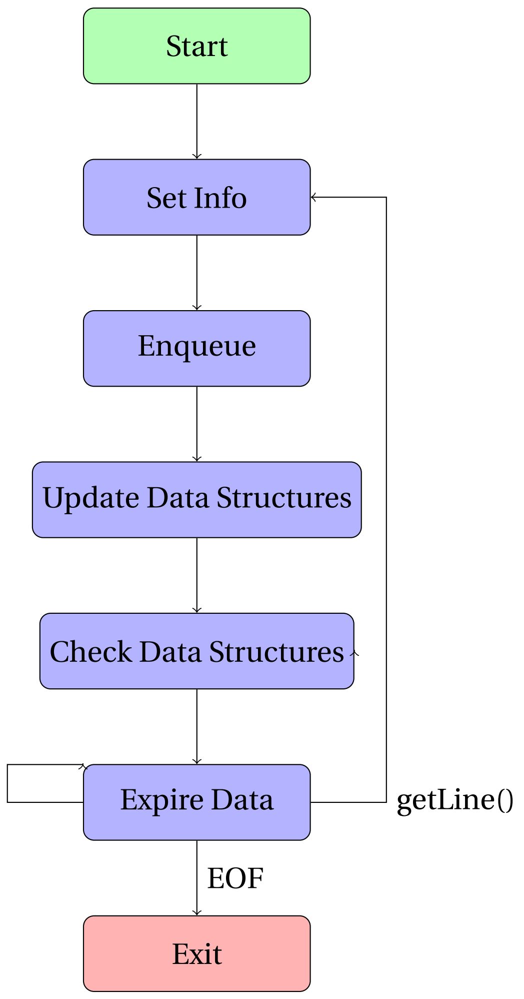
The IP map is a map from an IP address to a map from a username to a timestamp. The purpose of this data structure is to keep track of which and how many usernames are associated with each IP address. This allows for the detection of attackers who have compromised multiple accounts and are switching between them in an effort to avoid detection to potentially be detected. There is also a timestamp associated with each username which allows the association between the IP address and that particular user account to be broken after a period of time determined in the config file. The number of users associated with a given IP before flagging the administrator can also be set via the config file.

User Map

The user map is a map from a username to a map from an IP adress to a timestamp. The purpose of this data structure was to do the opposite of the IP map by tracking which IP addresses are associated with a given username. Just as in the IP map, the number of occurrances of each IP address was tracked for each user in order to determine when an IP address should no longer be associated with a username. As with the IP map, the data structure can be tuned via the config file to meet client needs. The purpose of this data structure was to watch for users that are rapidly switching IP addresses using the same user account.

ALGORITHMS

The main algorithm that was used for this program can be broken down into five distinct repeating steps and two auxilary steps. This steps are given by the following flow chart.



The first step in the algorithm calls the start function which basically serves as the main function for the program. At the beginning of the start function all of the data structures for the program are declared and the main stdin reading loop is started. From this point forward control of the program resides with this read loop until termination. The first thing that happens when a line is read into the program is that a Userdata object is created for the data in the line. This includes creating timestamps with the corresponding lookahead distances for the new record. It should be noted that the initial timestamp is taken from the current time according to the machine that this program is running on; not the timestamp contained within the line read from the log file. As long as the program is running on the same machine that the log is being kept on this should not pose a problem. However, if the log data is being sent over a network or a situation where some sort of buffer is in use then this could pose a problem. Once all of the data for the record is set the program then moves to push the data into the priority queue. This is done by pushing each timestamp as a weight and a pointer to the record as the value. Once the data is in the priority queue it is ready to be processed by each of the security data structures. This step largely deals with changing data in the maps to reset expiration times or modify counters. Once the data structures are properly up to date the alert checks are called for each one. This checks to see if an alert needs to be triggered based on the values contained within the maps. After any alerts have been sent old data is removed from the data structures. The logic for this step is that while there are data points that have a timestamp that is younger than the current time then the data point represented by that timestamp should be expired. This continues until the front of the priority queue "catches up" to the current time. Once all of the expirations are done then the program loops back around to the beginning of the file read function to recieve more input from the log file or to terminate if there is no input remaining.

PARADIGMS

Online Processing vs Batch Processing

For this project it was determined that online processing would be the proper way to approach this problem rather than batch processing. This is because the goal of this program was to deliver real time feedback on the status of the user accounts that were being tracked by the system. Using batch processing would therefore be less effective due to the need to accumulate data before the system could provide any response.

FUTURE WORK

Although this project was successful at a base level, there are still many features that would be useful and would increase the effectiveness of this program. The following are several ideas that may prove to be useful and interesting work.

Generic Log Reading

One thing that would add a lot of convenience to the program would be something that is able to read any log file and determine the form that the information takes. This would be useful because if there is any change in the logging software or if a system administrator changes what information is being recorded in the log there is a risk of the program failing.

Generic/Dynamic Tracking Patterns

In this vein, it would also be extremely useful to come up with some basic kinds of data forms that one might want to track in the log and then adapting them to work generically. For example, the user map and the IP map are basically the same thing. There are several other situations given different types of information that might be useful to track in this same way. For example, if location of the connection was included in the log file then it would be very useful to map this data to a username and then be able to expire this information in the same way that the IP/username associations are expired.

Extended Reporting

It may be helpful to the administrators to include more information in any activity report that is sent. For example, rather than just including the line number that set the off the alert it may be useful to include the line range for the entire traffic pattern that lead to the alert. It may also be helpful to administrators to assign alerts priority levels based on how often an alert actually ends in being a confirmed threat. This would allow high priority alerts to be sent immediately (perhaps even over SMS rather than SMTP) while low priority alerts may have some kind of time delay where they are allowed to build up before notifying the administrator.

Full Automation

Due to the enormity of the data that is being handled and processed by this program, this project may be a great candidate for machine learning. It may be possible to train a machine learning algorithm to determine when a user is comprimised and then perform containment procedures without the direct supervision of an administrator. This would increase both the accuracy and the response time of any retalitory action that would normally come from the system administrator.

ADDITIONS

As of 1 August 2018, this program is capable of reading from a pregenerated file rather than just from a constant stream of input. This feature was added to make testing and configuring the program much faster. In order to implement this there were some major changes to the way that timestamps were created and processed that were necessary. In the previous version, a record was created from a line and then, to save time, this record was just stamped with the current system time. The problem with this was that when you went to test the program this forced you to run your tests in real time. This made configuring the program extremely difficult because you had to wait in real time for the log to finish processing before you could observe the results of the tests and make any config adjustments. By pulling the time from the timestamp included in the line from the log file, we are able to avoid this problem at the cost of more processing time per record. Another benefit of doing the timestamping in this way is that if there is a major flood of requests and our data pipeline gets backed up the timestamping will will not be affected.