Finding Anomalies in Windows Event Logs Using Standard Deviation

John Dwyer
Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099, USA
dwyerj1@nku.edu

Traian Marius Truta
Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099, USA
trutat1@nku.edu

Abstract— **Security is one of the biggest concerns of any company that has an IT infrastructure. Windows event logs are a very useful source of data for security information, but sometimes can be nearly impossible to use due to the complexity of log data or the number of events generated per minute. For this reason, event log data must be automatically processed so that an administrator is given a list of events that actually need the administrator's attention. This has been standard in intrusion detection systems for many years to find anomalies in network traffic, but has not been common in event log processing. This paper will adapt these intrusion detection techniques for Windows event log data sets to find anomalies in these log data sets.**

***Keywords*-Windows Event Logs, Standard Deviation, Anomaly Detection.**

I. INTRODUCTION

Security is one of the biggest concerns of any company that has an IT infrastructure. It is very important for an administrator to always know the security posture of the network and servers that they manage. One way to always know the state of an environment is through logs [1]. While there are hundreds or thousands of devices that create logs on a network, most logs are hardly ever read due to the complexity and volume of the log data. This creates a problem for the administrator as logs must be reviewed, but if the whole day is spent reviewing logs (and this will not be enough time given the size and the complexity of these logs), there is never any time left over to react to the problems found in the logs.

Windows event logs are one of the best tools that can be used to find and remedy problems and vulnerabilities in Windows operating systems [2]. While Windows event logs are a very important source of information, they can be difficult to review as the default "Event Viewer" in Windows only gives options for basic filtering of events and doesn't give any options for correlation or other useful tools that could help an administrator find a problem quickly and efficiently [2, 3]. Another problem with trying to review Windows event logs is the speed at which they are created. If it takes an administrator 1 minute to review a log entry and logs are coming in at a rate of 50 per-minute, it becomes impossible for an administrator to review the logs. For this reason, the logs must be reviewed by a third-party software solution to remove the events that wouldn't concern the administrator and only show the administrator events that could help find problems and vulnerabilities.

This paper will introduce a novel approach to identify anomalies in Windows event log data using standard deviation. With a set of event logs, it is possible to use SQL queries to average the average number of events of a specific type at any time of the day for any server or user in the dataset. With this, the average number of events of a specific type can be determined and the standard deviation of those events can be determined. This allows alerting for times that go outside of the standard deviation. For example, if a specific server usually sees 150 login attempts at 8:30 AM on Monday and it receives 1000 login attempts at that time, an alert can be created to show that there is a possible breach. With these functions, it is also possible to alert on events that are not based on security problems. For example, if a large number of I/O errors are written to the event log by a failing hard drive, an administrator would be alerted due to the anomaly created by the influx of events. This proposed anomaly detection in Windows event logs is implemented with the help of SQL queries and Transact-SQL.

The remaining of the paper is structured as follows. Section II describes the data set used in this paper. Section III discusses the techniques used to de-identify the data set. Section IV documents the implementation of the anomaly detection techniques used in this research. Section V presents the preliminary findings that were gathered with the anomaly detection. The paper ends with future work directions and conclusions.

II. DATA DESCRIPTION

The data used in this paper was gathered from the event logs of approximately 30 production servers over the span of 6 months. This amounts to approximately 23GB of log data. The servers included a Citrix farm, Domain Controllers, Exchange servers, web servers, application servers, and database servers [4, 5]. All servers had auditing enabled for successful and failed logon attempts to track the number of logons at any specific time of day [6]. The log data was collected by converting the event logs into a syslog format and sending the logs to a central data store. This was implemented using Snare for Windows and a Snare server [7, 8]. The event logs were then imported into a SQL database using the Transact-SQL *bulk import* statement [9]. Once in the database, all identifiable information was transformed using a multitude hashing algorithms to protect the identity of the entity that provided the dataset [11]. These algorithms are described in Section III.

The standard Windows event log format contains many fields such as the server the log was generated from, the time and date the log was generated, the process or program that generated the log, a description of the event, the account that the event occurred under, and many other fields [11]. In a standard implementation, many of these fields may be used to give the administrator more information such as a description of the event to lower the amount of research that must be done to trace down a problem. An example of the data used can be seen in Fig.1. In this paper, only the following fields will be used:

- **Event Log** – Contains the name of the event log in which the log originated [10]. Only the *Application*, *Security*, and *System* event logs are used in this paper.
- **Event Source** – The program or process that generated the event. Many event sources are used in this paper [10]. For example, all login and logoff events are from the event source Security. In this instance both the event log and the event source have the same name but are separate fields, this is not the case with all events.
- **Event ID** – The unique ID of the event based on which Source generated the event. Event IDs are not unique between sources but are always unique within their own source. An Event ID is not specific to each event just each event of a specific type [10]. For example, all Windows account lockout events are placed in the *Security* log with a source of *Security* and an event ID of 644 [6].
- **Event Type** – This field describes the type of event that occurred and can be useful for determining what type of activity generated the event [10]. In the example events above, all events are of the type *Success Audit* which shows the events were created by successful login attempts.
- **Event Category** – This categorizes events into specific groups based on the type of event [10]. For example, the category Logon/Logoff events contains multiple event IDs which relate to the category.
- **Time/Date** – The time and date of the event is used to calculate the number of a specific event at any point in time throughout the day on any specific day of the week. The day of week is used due to the fact that you may have more login requests at a specific time on a Wednesday that you would on a Sunday. The Time/Date field of the event is split up into multiple columns using a Transact-SQL substring command. The columns that are used in the paper are *EvtHour*, *EvtMinute*, *EvtDayOfWeek*, *EvtDayOfYear*, *EvtYear*. Hours and minutes were intentionally separated to allow for easier computation of per hour and per-minute results.
- **Server ID** – This is a unique identifier for each server in the dataset. This is useful to help an administrator link back an alert to a specific server to identify where the problem occurred.
- **User ID** – This is a unique identifier for each user in the dataset. There are over 400 unique users identified in the dataset. This is useful to help an administrator link back an alert to a specific user to identify which user account may be linked to the problem or vulnerability.

All other fields were removed because they were not necessary for the processing of the events.

ServID	UserID	EvtLog	EvtSrc	EvtType	EvtCat	EvtID	EvtDayOf Week	EvtDayOf Year	EvtHour	EvtMin	EvtYear
17	247	Sec	Sec	Success Audit	Logon/Logoff	538	Tue	2-Oct	12	4	2012
17	247	Sec	Sec	Success Audit	Logon/Logoff	538	Tue	2-Oct	12	4	2012
17	247	Sec	Sec	Success Audit	Account Logon	680	Tue	2-Oct	12	4	2012
17	377	Sec	Sec	Success Audit	Logon/Logoff	552	Tue	2-Oct	12	4	2012
17	247	Sec	Sec	Success Audit	Logon/Logoff	528	Tue	2-Oct	12	4	2012

Figure 1. Sample log data.

III. Data Preprocessing

In this paper both known and custom hashing methods are used to remove identifiable information from the dataset while still keeping the integrity of the data. To be sure that the same username or server name is always transformed to the same value, a hashing table must be stored in the database [10]. This means that the stored hashes must also be directly related to the username and server name without actually storing the data in plain text. For this reason, a substring was taken of the value and the substring of the value was encrypted with a MD5 hash followed by a SHA1 hash [12]. Each encrypted substring was given a unique ID number which was used in presenting the data to allow it to be readable versus displaying an encrypted value. For instance, an event may show that user 415 logged into server 10. This was implemented to protect the privacy of the entity that provided the log data for analysis. In a real world scenario, it is likely that an administrator would not need to mask server names and user IDs as the data would be used fully inside the company and not provided to outside resources. The hashing tables were created with the SQL queries shown below in Fig. 2.

IV. Implementation

The processing of data and alerting is broken down into a 5 step process as can be seen in Fig. 3. Section III showed the process for de-identifying the data. This section will go through the steps of processing the data, counting the events, calculating the average number of events and generating alerts based on the average number of events and current count of events. These methods were implemented using Microsoft SQL Server 2008 R2 and Windows Server 2008 R2. The system used for testing had 8 processor cores, 16 GB of memory and 300 GB of solid state storage.

```
SELECT Row_Number() over(order by EncryptedServerName desc) as ServerID,
   EncryptedServerName INTO ServerHash
   FROM (
      SELECT distinct Hashbytes('SHA1',Hashbytes('MD5',
         Substring(Server,3,5))) as EncryptedServerName
      FROM RawLogs
   ) as ServerNames;

SELECT Row_Number() over(order by EncryptedUserName desc) as UserID,
   EncryptedUserName INTO UserHash
   FROM (
      SELECT distinct Hashbytes('SHA1',Hashbytes('MD5',
         Substring(User,3,5))) as EncryptedUserName
      FROM RawLogs
   ) as UserNames;
```

Figure 2. Source code for server and user hashing functions.

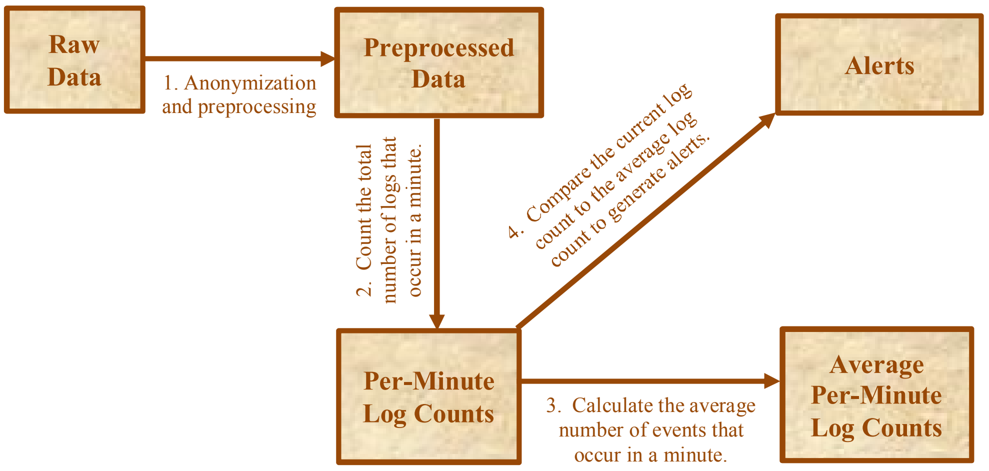

Raw Data
1. Anonymization and preprocessing
Preprocessed Data
2. Count the total number of logs that occur in a minute.
Alerts
4. Compare the current log count to the average log count to generate alerts.
Per-Minute Log Counts
3. Calculate the average number of events that occur in a minute.
Average Per-Minute Log Counts

Figure 3. Data processing and alerting process.

Once the events are de-identified they can now be processed to obtain usable information. The first step is to count the number of events that happened at a specific time of a specific day for specific servers and specific users. This can be accomplished by a simple select statement with a count and a group by clause as shown in Fig. 4.

This statement outputs the results into a new table for further analysis and calculation. This will allow lowering the granularity by grouping again and removing fields that are not needed and outputting the data into another table. This is useful in determining the number of events across all servers and users for a specific time of day or a specific day.

Now that the events have been counted, the next step is to compute the average number of events and standard deviation for a specific day of the week. The day of week is used as it is very important to event logs as a company that is only open 5 days a week will have much fewer events on Saturday and Sunday than the rest of the week. These statistics are calculated via the T-SQL `sum()` and `stddev()` methods. The complete SQL query can be seen in Fig. 5 below.

This SQL statement (see Fig. 5) once again outputs the data into another table so the averages and standard deviation will be available for alerting. Unlike the counts previously, this must be computed separately for each combination of attributes as the averages and standard deviation cannot be simply added together. This means that a separate table will be used for each level of granularity. Once this data is gathered, alerts can begin to be generated based on a comparison between the count table and the statistics table(s). The formula used to generate alerts is shown below:

$$Threshold = \frac{SUM(EvtCount)}{COUNT(*)} + k \cdot STDDEV(EvtCount)$$

Alerts are generated simply by comparing the number of events at one point in time on a specific day of the week to the average number of events at that time on that day of the week. The sum, count, and standard deviation were all determined in the last function and this function simply adds the values together and applies a multiplier. If the number of events is greater than the average plus k standard deviations, an alert is generated (where k is a constant that multiplies the value of the standard deviation). In this implementation, multiple k values were tested to reduce the risk of false negative results. High k values will cause higher chance of false negatives while low k values will cause a higher number of false positive results. This is a common trade-off in security related tasks as false negatives are much worse than false positives. A large number of false positives is also not a good thing to have as it causes an administrator to waste time researching alerts that do not relate to a problem. This alerting statement is shown in Fig. 6.

```
SELECT ServerID, UserID, EvtLog, EvtSrc, EvtID, EvtDayOfWeek,
   EvtDayOfYear, EvtHour, EvtMinute, EvtYear, COUNT(*) as EvtCount
INTO LogCountsPerUser
FROM LogBuffer
GROUP BY EvtYear, EvtDayOfYear, EvtDayOfWeek, EvtHour, EvtMinute,
   EvtLog, EvtSrc, EvtID, ServerID, UserID;
```

Figure 4. Source code for counting events.

```
SELECT ServerID, UserID, EvtLog, EvtSrc, EvtID, EvtDayOfWeek, EvtHour,
   EvtMinute, COUNT(*) as NumberOfDays, SUM(EvtCount) as EvtTotal,
   SUM(EvtCount)/COUNT(*) as EvtAverage, STDEV(EvtCount) as EvtStdDev
INTO EventsPerUser
FROM LogCountsPerUser
GROUP BY EvtDayOfWeek, EvtHour, EvtMinute, EvtLog, EvtSrc, EvtID,
   ServerID, UserID;
```

Figure 5. Source code for determining event averages.

```
SELECT c.ServerID, c.UserID, c.EvtLog, c.EvtSrc, c.EvtID, c.EvtHour,
   c.EvtMinute, c.EvtDayOfYear, c.EvtYear, c.EvtCount,
   e.EvtAverage + (e.EvtStdDev * 3.3) as Threshold
INTO PerUserAlerts
FROM LogCountsPerUser c join EventsPerUser e on(
   c.ServerID = e.ServerID and c.UserID = e.UserID
   and c.EvtLog = e.EvtLog and c.EvtSrc = e.EvtSrc
   and c.EvtID = e.EvtID and c.EvtHour = e.EvtHour
   and c.EvtMinute = e.EvtMinute and c.EvtDayOfWeek = e.EvtDayOfWeek)
WHERE c.evtCount > (e.EvtAverage +  (e.EvtStdDev * 3.3))
```

Figure 6. Source code for alerting function.

This SQL statement (see Fig. 6) will output all alerts into an alert table. In a real-world environment, this table would likely have more values such as a description of the event to help the administrator know more about the alert to help identify the problem. This alert table could have triggers implemented against the table that generate emails to notify administrators as soon as an alert is generated. If only daily reporting is needed, SQL jobs could scrape the table daily and send a daily report showing the events from the previous day.

The per-user alerting functions are very useful for finding anomalies for a specific user on a specific server but they are unable to alert across multiple users and multiple servers. For this reason, another set of functions must be created for this ability. This will show attacks that are happening across many user accounts and/or multiple servers. Attacks across multiple user accounts are common with brute-force attacks to try to crack passwords to obtain network access. Attacks across multiple servers are much more prevalent than attacks against one server as it gives an attacker more possible entry points into the network. The per-user functions would not detect these types of attacks as the alerting is centered around a specific user on a specific server and thus it would not catch the anomaly. This requires counting the number of events without specifying *ServerID* and *UserID* along with a slight reconfiguration of the functions for determining event averages and alerting. The statement for counting logs can be seen in Fig. 7 and the updated average and alerting statements can be seen in Fig. 8 and Fig. 9 respectively.

In testing, these statements produce much better results due to the fact that the data can show a problem across the whole environment versus just a problem with one user on one server fixing the problem with the high number of false positive results that can be seen in the per-user data. These statements (see Figs. 7 – 9) can be taken even a step farther to alert based on intervals of minutes or hours. This is accomplished through the use of the Transact-SQL *floor* function. It is used to round the value of the *EvtMinute or EvtHour* field to the nearest *x* minute. For example, if we wanted to do alerting based on 5 minute intervals, we would simply round everything to the previous 5 minute mark (08:23 would round down to 08:20). This method allows the average and alerting per-minute functions to still work with a slightly modified event counting function. The modified statement is shown in Fig. 10.

The per-minute queries are very useful to find attacks across multiple servers and multiple users, but they still fall a bit short in giving the administrator all of the information needed to track a possible breach. The alert will show that an event happened on the network but it is not traced back to a specific server or group of servers. This can be accomplished by simply adding the per-user and per-minute functions together. If an alert is generated by any of the per-minute functions, an administrator could look for events of the same type at the same time on the same date in the per-user alert data. This will help the administrator trace back the problem to specific hosts or specific user accounts at that point in time.

```
SELECT EvtLog, EvtSrc, EvtID, EvtDayOfWeek, EvtDayOfYear,EvtHour,
   EvtMinute, EvtYear, Sum(EvtCount) as EvtCount
INTO LogCountsPerMinute
FROM LogCountsPerUser
GROUP BY EvtYear, EvtDayOfYear, EvtDayOfWeek, EvtHour, EvtMinute,
   EvtLog, EvtSrc, EvtID;
```

Figure 7. Source code for counting events per-minute.

```
SELECT EvtLog, EvtSrc, EvtID, EvtDayOfWeek, EvtHour, EvtMinute,
   COUNT(*) as NumberOfDays,SUM(EvtCount) as EvtTotal,
   SUM(EvtCount)/COUNT(*) as  EvtAverage,
   STDEV(EvtCount) as EvtStdDev
INTO EventsPerMinute
FROM LogCountsPerMinute
GROUP BY EvtDayOfWeek, EvtHour, EvtMinute, EvtLog, EvtSrc, EvtID;
```

Figure 8. Source code for determining per-minute averages.

```
SELECT c.EvtLog, c.EvtSrc, c.EvtID, c.EvtHour, c.EvtMinute,
   c.EvtDayOfYear,  c.EvtYear, c.EvtCount,
   e.EvtAverage + (e.EvtStdDev * 3.3) as Threshold
INTO PerMinuteAlerts
FROM LogCountsPerMinute c join EventsPerMinute e on(
   c.EvtLog = e.EvtLog and c.EvtSrc = e.EvtSrc
   and c.EvtID = e.EvtID and c.EvtHour = e.EvtHour
   and c.EvtMinute = e.EvtMinute and c.EvtDayOfWeek = e.EvtDayOfWeek)
WHERE c.evtCount > (e.EvtAverage + (e.EvtStdDev * 3.3));
```

Figure 9. Source code for per-minute alerting function.

```
SELECT EvtLog, EvtSrc, EvtID,  EvtDayOfWeek, EvtDayOfYear, EvtYear,
   EvtHour, Floor(EvtMinute/5)*5 as EvtMinute, Sum(EvtCount) as EvtCount
INTO LogCountsFiveMinute
FROM LogCountsPerUser
GROUP BY EvtYear, EvtDayOfYear, EvtDayOfWeek, EvtHour,
   Floor(EvtMinute/5)*5, EvtLog, EvtSrc, EvtID;
```

Figure 10. Source code for minute interval counting function.

V. Results

The purpose of the methods implemented in this paper is to make it easier for an administrator to find problems in Windows event logs by narrowing down the number of logs that an administrator must view to only logs that are of some concern to the administrator. The first part of this process is to calculate the best *k* value to use for alerting based on the number of false positive results and false negative results generated by the alerting function. The optimal *k* value would produce no false negative results while producing very few false positive results. It was determined that the optimal *k* value was 3.3 based on the number of alerts and lack of false negative results. The number of alerts per *k* value can be seen in Fig. 11. This graph shows that as the value of *k* grows, the number of alerts decreases.

It is also apparent through this graph that the per-user function creates substantially more alerts than the per-minute function. It was determined that the per-user function was not useful for determining what events are anomalies on its own and only became more useful as it was applied to the per-minute functions to narrow the results further to specific servers at the time of the alert.

There were over 23 million log entries in the data set used in this paper. The per-user alerting function generated approximately 10,000 alerts while the per-minute alerting function generated only 554 alerts. The large difference in results is mostly caused by the differences in the way that the two functions work. In general, the per-user function is more likely to give more false positives as the number of events is much smaller and it is more difficult to get good values for the averages and standard deviation; thus, the results are not as good. If the data set was log data for 2 or 3 years, the results may be a bit better, but in this case, the per user function is not a good fit for the dataset. Sample results of the per-user function can be seen in Fig. 12.

The per-minute results are far more accurate due to the number of events that occur across all servers and all users in a minute. In comparison to the per-user results, the per-minute results show over 10,000 events in some instances while the per-user results tend to stay in the 5-20 event range throughout a minute. The results shown in Fig. 13 show some of the anomalies caught by the per-minute alerting. These results are far better than that of the per-user results in the case that some results are over 1000 events above the threshold while the per-user results were usually around 1 event above the threshold amount. The fifth result in Fig. 13 shows that in a period of 1 minute, over 14000 successful logins were made on the network and this was over 1600 logins above the threshold. This is something that an administrator would want to know to determine why there was nearly a 10% increase in the number of logins at that time. Through manual review it was possible to verify that this was an anomaly in the data set and was caused by a new system being added to the environment.

The functions based on intervals of minutes also showed many of the same alerts that the per-minute function produced. These functions could be useful to an administrator for attacks that carry on for longer periods of time and wouldn't generate an alert with only per-minute alerting enabled. The number of alerts created by these functions compared to the alerts created by the per-minute function can be seen in Fig. 14.

Due to the size of the data set, it is important that the functions complete quickly to produce results. In an implementation where the events are streaming through the functions as they are collected from servers, it must take less than one minute to process one minute of logs so that the functions do not bottleneck the processing of data. In testing it took approximately 15 minutes to compute alerts for 6 months or 23 GB of data. Based on this information, the functions are able to process approximately 1.5 GB of data per minute on the test system used. Based on these results, one is able to conclude that the functions perform well and are able to handle large environments that produce large amounts of log data.

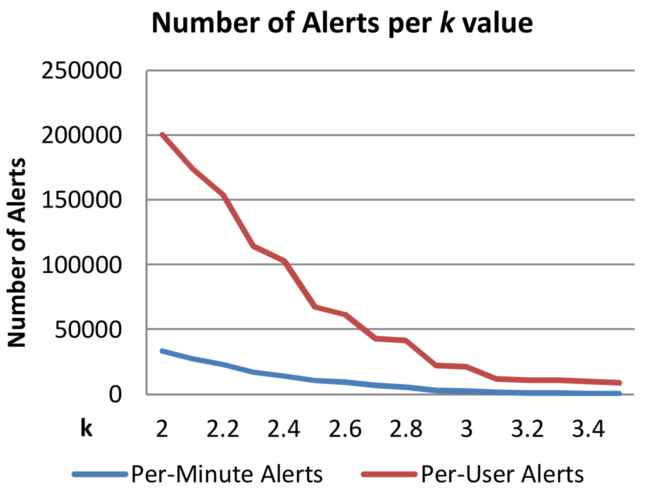


Figure 11. Number of alerts per *k* value.

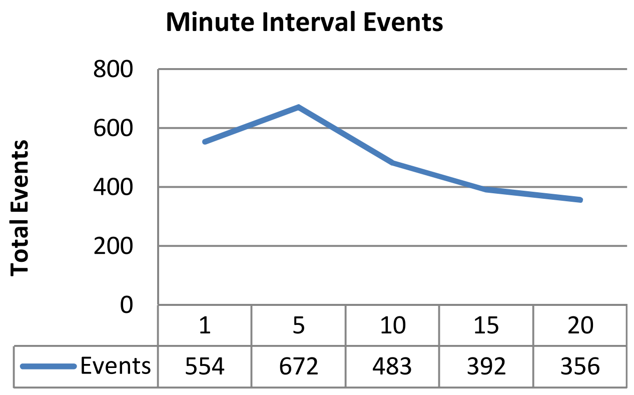
ServID	UserID	EvtLog	EvtSrc	EvtID	EvtHour	EvtMin	EvtDayOf Year	EvtYear	EvtCount	Threshold
1	247	Sec	Sec	680	6	29	7-Jun	2012	5	4.9830532
1	247	Sec	Sec	680	10	32	12-Jul	2012	4	3.9830532
1	297	Sec	Sec	538	6	40	10-Sep	2012	2	1.9949874
1	377	Sec	Sec	552	1	14	7-Sep	2012	4	3.8894636
1	377	Sec	Sec	552	5	51	5-Jul	2012	3	2.9830532

Figure 12. Sample per-user alert data.

EvtLog	EvtSrc	EvtID	EvtHour	EvtMin	EvtDayOf Year	EvtYear	EvtCount	Threshold
Sec	Sec	538	1	2	10-Sep	2012	11573	10933.816
Sec	Sec	538	2	24	13-Jul	2012	4766	4672.9863
Sec	Sec	538	8	0	7-Sep	2012	2	5031.649
Sec	Sec	538	8	42	10-Sep	2012	5124	3918.3936
Sec	Sec	538	8	43	10-Sep	2012	14372	12702.781

Figure 13. Sample per-minute alert data.

Due to the size of this data set and price of software that can perform similar tasks, it was not possible to test these results against any other software to test the validity of the results. Most event log analysis software packages are not free to use, thus they could not be used in testing for this paper. Data mining software that has built-in anomaly detection was also not able to be used due to the size of the data set.



	1	5	10	15	20
Events	554	672	483	392	356

Figure 14. Number of alerts for per-minute intervals.

VI. Conclusions and Future Work

This paper introduced a novel approach to find anomalies in Windows event log data through the use of standard deviation. This was accomplished through the use of SQL queries and Transact-SQL. Two possible methods for generating alerts were tested and it was determined that it is best to generate alerts across all servers and all users versus generating alerts for specific users on specific servers. The results show some use to an administrator in the fact that it lowers the amount of logs that must be reviewed to an amount that is feasible to review. Through manual review, it was determined that many of the results generated by the per-minute functions were actual anomalies in the data set and needed further review from an administrator.

There are two paths to explore in the future with this research. First, other measures could be used rather than standard deviation and other tasks could be performed to help with the speed of processing the data in larger systems. Second, other software could be purchased to verify the results rather than reviewing the results manually to check the validity of alerts. In the future, it would also be useful to implement methods to allow the data to automatically generate alerts versus having to manually run each step of the process as was done in this paper.

References

[1] R.M. Magalhaes, "Understanding Windows Logging," *WindowsSecurity.com*, Jan. 2013. [Online]. Available: http://www.windowsecurity.com/articles-tutorials/windows_os_security/Understanding_Windows_Logging.html.

[2] K. Dashora, D.S. Tomar, J.L. Rana,, "A Practical Approach for Evidence Gathering in Windows Environment". *International Journal of Computer Applications*, Vol. 5, Number 10, pp. 21–27, August 2010.

[3] "Event Viewer," *Wikipedia*, Mar. 2013. [Online]. Available: http://en.wikipedia.org/wiki/Event_Viewer.

[4] Citrix XenApp Official Site, Available: http://www.citrix.com/downloads/xenapp.html.

[5] "Citrix XenApp," *Wikipedia*, Mar. 2013. [Online]. Available: http://en.wikipedia.org/wiki/Citrix_XenApp.

[6] "Auditing Logon Events," *Microsoft Technet*, Jan. 2005. [Online]. Available: http://technet.microsoft.com/en-us/library/cc787567%28v=ws.10%29.aspx.

[7] Snare Official Site, Available: http://www.intersectalliance.com/projects/SnareWindows/.

[8] "Snare (software)," *Wikipedia*, Dec. 2012. [Online]. Available: http://en.wikipedia.org/wiki/Snare_%28software%29.

[9] "Transact-SQL," *Wikipedia*, Feb. 2013. [Online]. Available: http://en.wikipedia.org/wiki/Transact_SQL.

[10] R.F. Smith, "Event-Log Fields," *Windows It Pro*, Apr. 2004. [Online]. Available: http://windowsitpro.com/systems-management/event-log-fields.

[11] "Hash Function," *Wikipedia*, Mar. 2013. [Online]. Available: http://en.wikipedia.org/wiki/Hashing_algorithm.

[12] B. Mulvey, "Evaluation of SHA-1 for Hash Tables", 2007, *Hash Functions*, Available: http://home.comcast.net/~bretm/hash/9.html.