AN ALGORITHM FOR HTTP BOT DETECTION

Daryl Ashley
Senior Network Security Analyst
University of Texas at Austin - Information Security Office
ashley@infosec.utexas.edu
January 12, 2011

Introduction

In the paper "Botnet Command and Control Mechanisms", botnets are described as "collections of compromised computers (Bots) which are remotely controlled by its originator (BotMaster) under a common Commond-and-Control (C&C) infrastructure". The paper then describes the popularity and advantages of the IRC protocol as the C&C infrastructure as well as the emergence of HTTP as a C&C mechanism (Zeidanloo 2009).

Both IRC and HTTP can be used to control a number of infected systems, but there are some fundamental differences between the two infrastructures. Within an IRC botnet, infected hosts join an IRC channel and await commands from a botmaster that has also joined the channel (Zeidanloo 2009). In this way, commands can be "pushed" to infected clients at any time. The HTTP mechanism requires the infected client to send an HTTP GET request to a C&C website. Hosts that are infected within this type of infrastructure use a "pull" mechanism and can only receive commands after they have sent a request to the C&C site (Gu 2008).

An infected computer that is controlled by either infrastructure can be instructed to perform malicious activities. For example, they may be instructed to send SPAM or scan computers for weak SSH passwords. The malicious activity may be detected due to the anomalous network activity or because a system administrator from another network reports the malicious activity. But, if the infected computer can be identified based on C&C activity alone, the computer can be remediated before it has a chance to start performing any malicious activity. This can reduce the damage caused by a computer controlled in this way.

This paper describes an algorithm for detecting potential HTTP C&C activity based on repeated HTTP connections to a C&C website. If a website is identified by the algorithm, an analyst can perform additional research to determine whether or not the website is involved in C&C activity. If the website is malicious, Intrusion Detection System (IDS) signatures can be developed to detect the activity.

HTTP C&C Polling Activity

A partial packet capture showing network activity to a C&C website is shown in Figure 1. A security analyst can develop an IDS signature by looking for strings within the URI portion of the HTTP GET request. The analyst would focus on strings that appear to be part of the protocol employed by the C&C infrastructure, but that probably will not occur in normal HTTP traffic. In this case, the rule could look for the presence of the strings *.php?* and *x4x4x*. A large number of signatures on the Emerging Threats website examine the URI portion of the HTTP GET request (Emerging Threats 2010).

```
GET /babynot/hxncsi.php?ncsxd=72<1=07x644420x4x4x4x0x HTTP/1.1
Host: cdn.cbtclick.biz
Cache-Control: no-cache
```

Figure 1: Packet capture showing HTTP C&C activity

A host infected with this type of malware does not receive instructions directly from a C&C website because a TCP connection is not maintained with the C&C site (Lee 2008). Instead, the infected host must initiate a TCP session and send an HTTP GET request to the website to request a command. The C&C site will then send commands to the infected host as a response to the HTTP GET request. One advantage of this approach is that the C&C website does not need to keep track of which computers are infected. Instead, it waits for infected hosts to contact the website for instructions. A disadvantage is that the website cannot maintain control of an infected host unless the infected host sends HTTP GET requests to the website on a recurring basis. This paper will refer to the repeated HTTP GET requests as *polling*.

A simple mechanism that will cause an infected host to repeatedly poll for commands is for the malicious code to make calls to *sleep()* within a *while* loop. Figure 2 shows example C code that may be used to accomplish this type of polling activity.

If the malicious code uses the code snippet in Figure 2, the requests to the C&C website should occur at 60 second intervals. Figure 3 shows timestamps for HTTP sessions with

```
while (1) {
        get_and_process_cnc_commands();
        sleep(60);
}
...
void get_and_process_cnc_commands()
{
        // Insert code here to connect to C&C website, poll for a command
        // and process the command
}
```

Figure 2: Sample code to illustrate sleep mechanism

a C&C website by an infected host. The figure also shows the time interval between each HTTP session. The infected host in Figure 3 appears to be polling the C&C website once every hour. The consistent nature of the data values in Figure 3 has been referred to as "Periodic Repeatability" (Lee 2008).

Date/Time	Time Interval (seconds)
2010-05-01 21:47:46	-
2010-05-01 22:47:50	3604
2010-05-01 23:47:53	3603
2010-05-02 00:47:55	3602
2010-05-02 01:47:58	3603
2010-05-02 02:48:01	3603
2010-05-02 03:48:04	3603
2010-05-02 04:48:07	3603
2010-05-02 05:48:10	3603
2010-05-02 06:48:13	3603
2010-05-02 07:48:16	3603
2010-05-02 08:48:19	3603
2010-05-02 09:48:22	3603
mean = 3603 std dev = 0.43	

Figure 3: HTTP Polling Data

Use of Standard Deviation to Detect "Periodic Repeatability"

In order to create automation to detect this type of polling activity, the standard deviation of the time intervals was used to measure how consistent the time intervals are. For example, if the malicious code is using a sleep interval of 3603 seconds and the mean and standard deviation of the observed polling activity is computed, the mean should be about 3603 seconds and the standard deviation should be small. Since the sleep interval may differ depending on various malware variants, the computed mean may not be of much value. However, a small standard deviation provides some evidence that the polling activity is being controlled by a call to the *sleep()* function. The standard deviation for the data in Figure 3 is very small (less than 1 second) and may be an indication of automated activity.

Removing Anomalous Data

There are some problems with this approach. Consider the data in Figure 4(a). By visually inspecting the data, it appears that the infected host is polling the C&C website at approximately 1 hour intervals. However, there are several large time intervals mixed in with the regular polling activity. These large time intervals may be present because the infected computer was either turned off or disconnected from the network. For example, this may be a laptop that was taken home at the end of the work day. When the standard deviation is computed using the data in Figure 4(a), the value is no longer small.

If the large time interval values are ignored, the standard deviation would be very small and automation could detect the polling activity. A tool that can be used to isolate the values associated with the polling activity is the k-means algorithm. The k-means algorithm can be used to partition data sets into a number of clusters. If the data in Figure 4(a) can be partitioned so that all the values that are approximately 3600 seconds are in one cluster, and the remaining values are assigned to other clusters, one of the clusters will contain data values that have a mean close to 3600 seconds and a small standard deviation.

To use the k-means algorithm, the following must be specified:

1. The number of clusters to partition the data into
2. The initial mean of each cluster

The algorithm was run using 2 means, with the initial means calculated as follows: mean 1 was assigned the lowest data value and mean 2 was assigned the largest data value. Figure 4(b) shows the resulting partitions after using the algorithm.

The data belonging to the first cluster has a mean of 3603.6 and a standard deviation of 1.8. The standard deviation is small and provides evidence of automated polling activity.

The first cluster has an additional characteristic: most of the data values (11 out of 13) are located in the first cluster. The large data values that were likely due to the computer being powered off or disconnected from the network have been partitioned into a different cluster.

Date/Time	Time Interval (seconds)
2010-04-30 12:51:52	-
2010-04-30 13:52:02	3610
2010-04-30 14:52:12	3610
2010-04-30 15:52:15	3603
2010-04-30 16:52:18	3603
2010-05-01 05:00:56	43718
2010-05-01 06:01:00	3604
2010-05-01 07:01:03	3603
2010-05-01 08:01:06	3603
2010-05-01 09:01:09	3603
2010-05-01 10:01:12	3603
2010-05-01 11:01:16	3604
2010-05-01 21:47:46	38790
2010-05-01 22:47:50	3604
mean = 9396.8 std dev = 14174.4	

(a) Time Intervals

	cluster 1	cluster 2
Initial Mean	3602	43718
Final Mean	3605	41254
Data	3610	38790
	3610	43718
	3603	
	3603	
	3604	
	3603	
	3603	
	3603	
	3603	
	3604	
	3604	

(b) Data Clusters

Figure 4: More Complete HTTP Polling Data

Summary of Algorithm

Based on the information in the preceding section, the following algorithm can be used for detecting HTTP polling activity. Given a set of time intervals:

1. Determine if there are "enough" time interval values to perform the analysis. For example, there are 13 data values in Figure 4. If there were only 3 data values, there might not be enough evidence to confirm that polling is occurring.
2. If there are "enough values", use the k-means algorithm to partition the data
3. Determine if a single cluster contains "most" of the data values
4. If a cluster contains "most" of the data values, compute the standard deviation of the cluster.

5. If the computed standard deviation is "small enough", assume that automated polling activity is occurring.

In order to use this algorithm, the number of nodes used in the k-means algorithm and an algorithm for computing the initial means must be specified. Also, three configuration parameters must be specified:

1. An integer value for step 1 that will determine if enough data values are present to perform the analysis.
2. A percentage for step 3 that will determine if enough of the data values are present in a single cluster.
3. A maximum standard deviation amount for step 5.

An implementation of the algorithm in Perl is included as an appendix. The implementation will partition an array of data into 2 clusters. The minimum and maximum data values in the array are used as the initial means.

Number of Initial Clusters

One problem with using the k-means algorithm to detect polling activity is that using a different number of clusters may produce different results. The data in Figure 5 is almost identical to the data in Figure 4(a). There is one exception - a 7206 second time interval on 5-1 at 9:01. This time interval may have been caused by the C&C website being unavailable for some reason when the infected host tried to poll for a command at 8:01.

Figure 6(a) shows the results of the k-means algorithm using the same initial means that were used to partition the data in Figure 4. The data belonging to the first cluster has a mean of 3965 and a standard deviation of 1080. Once again, the standard deviation is large. However, if the data is partitioned into 3 clusters, and 3602, 7206, and 43718 are used as the initial means, a cluster will be produced with a low standard deviation. The results are shown in Figure 6(b).

Calculation of Initial Means

A second problem with the K-means algorithm is that using different initial means can yield different results. The results shown in Figure 6(b) were produced when the initial means were set to 3602, 7206, and 43718. If the initial means are instead set to 3602, 38790, 43718, the data will be partitioned as shown in Figure 7. Once again, the cluster with the large number of elements has a large standard deviation.

Date/Time	Time Interval (seconds)
2010-04-30 12:51:52	-
2010-04-30 13:52:02	3610
2010-04-30 14:52:12	3610
2010-04-30 15:52:15	3603
2010-04-30 16:52:18	3603
2010-05-01 05:00:56	43718
2010-05-01 06:01:00	3604
2010-05-01 07:01:03	3603
2010-05-01 09:01:09	7206
2010-05-01 10:01:12	3603
2010-05-01 11:01:16	3604
2010-05-01 21:47:46	38790
2010-05-01 22:47:50	3604
mean = 9396.8 std dev = 14174.4	

Figure 5: More Complete HTTP Polling Data

	cluster 1	cluster 2
Initial Mean	3602	43718
Final Mean	3965	41254
Data	3610	38790
	3610	43718
	3603	
	3603	
	3604	
	3603	
	7206	
	3603	
	3604	
	3604	
Cluster 1 mean = 3965 std dev = 1080		

(a) Two Clusters

	cluster 1	cluster 2	cluster3
Initial Mean	3602	7206	43718
Final Mean	3605	7206	41254
Data	3610	7206	38790
	3610		43718
	3603		
	3603		
	3604		
	3603		
	3603		
	3604		
	3604		
Cluster 1 mean = 3605 std dev = 2.8			

(b) Three Clusters

Figure 6: Using Different Sized ClustersTo Partition Data

	cluster 1	cluster 2	cluster3
Initial Mean	3602	38790	43718
Final Mean	3965	38790	43718
Data	3610 3610 3603 3603 3604 3603 7206 3603 3604 3604	38790	43718
Cluster 1 mean = 3965 std dev = 1080			

Figure 7: Data Partitioned Into 3 Clusters

Alternative Method For Poll Detection

The problem of finding a large cluster of data values with a small standard deviation can be restated as follows:

> Let S be a set containing n integers. Does there exist a subset T of S that contains at least r elements which has a standard deviation less than some value σ?

If such a subset is found, the number and/or values of the initial means used by the k-means algorithm may be modified to detect the subset.

If no subsets are detected by the k-means algorithm, a brute force approach could be used to verify that no such subset exists. This would require computing the standard deviation for all subsets of S containing at least r elements. If none of the subsets have a standard deviation $< \sigma$, then the set S will not have any subsets meeting this criteria. This approach would require computation of the standard deviation for the following number of

subsets:

$$\sum_{i=r}^{n} \binom{n}{i}$$

For large n, this approach would become impractical. However, the following can be used to reduce the number of computations (proofs are provided in Appendix 1):

1. The standard deviation of the elements in a subset T will be greater than $(M-m)/\sqrt{2n}$, where M is the maximum value in T and m is the minimum value in T. Note that n is the number of elements in S, not T.
2. Any subset of S that contains T will have a standard deviation that is also greater than this value.

The following example outlines how this can be used. Consider the sorted set of values shown in Figure 8. The set of data contains 12 elements, so in this case, $n = 12$. Suppose that a "large" cluster is defined as a subset that contains at least 9 elements, so $r = 9$. Also, a "small" standard deviation is defined as any value less than 30 seconds, so $\sigma = 30$. Note that x_1, x_2, x_3, and x_4 are the smallest elements in the set of values. At least one of these elements must be present in any subset containing 9 elements. If not, the subset would contain at most 8 elements. Suppose x_1 is the element that is in the subset. Next, note that elements x_9, x_{10}, x_{11}, and x_{12} are the largest elements in the set of values, and that at least one must be present in the subset of 9 elements. If this not the case, once again, the subset would contain at most 8 elements. Since x_9 is the smallest of the 4 values, the value of $M - m$ for any subset containing x_1 that has at least 9 elements will be at least $x_9 - x_1$. So, when $(M-m)/\sqrt{2n}$ is computed for any subset that contains x_1, the lower bound for the standard deviation will be $(x_9 - x_1)/\sqrt{2n}$.

Similar arguments can be used to show that if x_2 is the smallest element in a subset containing at least 9 elements, the value for $M-m$ must be at least $x_{10}-x_2$. If the smallest element of the subset is x_3, then $x_{11}-x_3$ should be computed. Finally, if x_4 is the smallest element in the subset, $x_{12} - x_4$ should be computed. Since any subset containing at least 9 elements will have a minimum value of x_1, x_2, x_3, or x_4, all subsets containing at least 9 elements will be accounted for by these calculations.

If $(M-m)/\sqrt{2n} \geq 30$ for each computation, then the standard deviation will be ≥ 30 for any subset of S containing at least 9 elements. Figure 9 shows the results of the computations when this approach is applied to the data in Figure 8. Since none of the computed values is less than 30, there is no subset containing at least 9 elements with a standard deviation less than 30. Therefore, the k-means algorithm cannot be used to partition the data so that a cluster of at least 9 elements with a standard deviation less

Item No.	Time Interval (seconds)
x_1	0
x_2	3603
x_3	3603
x_4	3603
x_5	3604
x_6	3604
x_7	3604
x_8	3610
x_9	3610
x_{10}	7206
x_{11}	38790
x_{12}	43718

Figure 8: Sorted HTTP Polling Data

Smallest Element	M	m	$(M-m)/\sqrt{2n}$
x_1	3610	0	737
x_2	7206	3603	737
x_3	38790	3603	7183
x_4	43718	3603	8188

Figure 9: Minimum Computed Standard Deviations

than 30 is found, regardless of the number of means used or how the initial values of the means are chosen.

Note that for an arbitrary set with n values, if a "large" subset must contain at least r elements, only $n-r$ computations would be required. If, however, $(M-m)/\sqrt{2n} < 30$ for any of the computations, there may be a subset indicative of polling activity. If, in this situation, it is assumed that a subset indicative of polling exists, this approach can be used as an alternative method for detecting polling activity. A perl implementation of the alternative approach is provided in Appendix 3.

Potential False Positives

Since the alternative method computes a lower bound for the standard deviation, there is a possibility that the actual standard deviation will be larger than σ even though the lower bound is less than σ. So, if the computed lower bound is less than σ, the standard deviation of subsets containing the r elements would need to be computed to determine if any of

the subsets have a standard deviation less than σ. Consider the data in Figure 10 and the computations shown in Figure 11. The first computation in Figure 11 shows a lower bound less than 30. But, the actual standard deviation for the 9 smallest elements in Figure 10 is 53. So, it is possible for the alternative approach to produce false positives. However, the alternative approach will never *exclude* any set that contains a subset indicative of polling activity.

Item No.	Time Interval (seconds)
x_1	3459
x_2	3470
x_3	3491
x_4	3500
x_5	3521
x_6	3544
x_7	3572
x_8	3598
x_9	3603
x_{10}	7206
x_{11}	38790
x_{12}	43718

Figure 10: Sorted HTTP Polling Data

Smallest Element	M	m	$(M-m)/\sqrt{2n}$
x_1	3603	3459	29
x_2	7206	3470	763
x_3	38790	3491	7205
x_4	43718	3500	8209

Figure 11: Minimum Computed Standard Deviations

A Side Note On Standard Deviations

Consider the set $S = \{1, 10\}$. The standard deviation for the elements in this set is 6.36. Next, consider the set $S' = \{1, 10, 11\}$. The standard deviation for the elements in S' is 5.51. Notice how the standard deviation for the elements in S' is smaller than standard deviation for the elements in S even though $S \subset S'$. For this reason, the alternative

approach cannot use the standard deviation in place of $(M - m)/\sqrt{2n}$ when determining the lower bound for the standard deviation of any subset of S that contains T.

Runtime Performance

As long as the number of iterations performed by the k-means algorithm is reasonably bounded, its runtime performance for a one-dimensional data set is $O(n)$ (Tan 2006). Once the data has been partitioned, at most n computations are required to count the number of values in each partition. If there are k clusters, at most, k comparisons to r will be required to determine if one of the clusters has enough data values. Therefore, identification of a single large cluster is $O(n)$. It's straightforward to show that computation of the standard deviation for the large cluster is also $O(n)$. So, the total runtime for this algorithm is $O(n)$.

Since the alternative approach requires the elements in the set to be sorted, the runtime of the sorting operation will be at best $O(n\log\, n)$. Since there are at most $n-r$ computations required to compute the lower bounds for the standard deviations, this portion of the algorithm is $O(n)$. Therefore the runtime of the alternative approach is $O(n\log\, n)$. Both approaches are much faster than the brute force method, which has a runtime of $O(n!)$.

IDS Signatures

Repeated HTTP sessions must occur between an infected client and a C&C website before the algorithms described in this paper will detect the activity. An IDS signature, however, will detect malicious activity as soon as it begins. The data in Figure 12 shows an infected client contacting a C&C website approximately once every two minutes. If the poll detection algorithm is configured to require 6 time interval values before analyzing for polling activity, the algorithm will not identify this activity until about 12 minutes after the first HTTP session. If the malware used a one hour polling interval, the infected host would remain undetected for 6 hours.

As mentioned earlier, an IDS signature can be written to detect this activity by looking for strings that may be unique to this malware's protocol. For example, the signature can inspect a URL for the presence of the following strings: *bb.php?v=*, *&id=*, *&b=*, and *&tm=*. This would allow an IDS to detect the malicious activity in Figure 12 during the first HTTP session. Since an IDS can identify malicious activity so much faster than the poll detection algorithms, infected hosts can be remediated much more quickly when using an IDS. However, if a signature has not been developed for a specific malware variant, the IDS will not detect hosts that are infected by the variant. The algorithms in this paper can be used to identify such variants. If there was no IDS signature that would detect the

malware responsible for the activity in Figure 12, the activity could be detected by one of the polling algorithms. Once the activity was identified, the information in the URLs, packet captures, or other sources could be used to write a signature.

Date/Time	URL
2010-12-10 12:06:18	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=0
2010-12-10 12:08:18	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=2
2010-12-10 12:10:19	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=4
2010-12-10 12:12:19	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=6
2010-12-10 12:14:20	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=8
2010-12-10 12:16:21	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=10
2010-12-10 12:18:22	http://doughaa.com/full/bb.php?v=200&id=501087576&b=loadeale&tm=12

Figure 12: HTTP Bot Session Data

Conclusion

The algorithm described in this paper can be used to detect automated polling activity to a website. However, non-malicious polling activity, such as web traffic generated by stock tickers, weather updates, and torrent sites may also be identified by these algorithms. Whitelisting of specific URLs can be used to improve the accuracy of these approaches.

Preprocessing of network data may also improve the accuracy of the algorithms. For example, the HTTP bot communication is composed of HTTP GET requests followed by a command from the C&C website. If the C&C website generates commands that are short in length, a small number of bytes may be exchanged between the infected client and C&C website during an HTTP session. Instead of analyzing all HTTP sessions, these algorithms can be used to analyze HTTP sessions that have small amounts of data transferred during the session. If the algorithms are supplemented by whitelisting websites and preprocessed network data, they may be effective in identifying malicious websites.

However, even if these algorithms can be improved to a point where no false positives are generated, they require a certain amount of data before any analysis can be performed. This means that an infected host will remain undetected for a period that is proportional to its sleep interval. Therefore, it is recommended that these algorithms be used to supplement the signatures used by an IDS to detect HTTP C&C activity.

References

Emerging Threats (2010). Rulesets retrieved November 17, 2010 from Emerging Threats Web site: http://www.emergingthreats.net/

Gu, G, Zhang, J, & Lee, W (2008). BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. Retrieved December 21, 2010 from Texas A&M University Computer Science and Engineering Department Web site: http://faculty.cs.tamu.edu/guofei/paper/Gu_NDSS08_botSniffer.pdf

Lee, J, Jeong, H, Park, J, Kim, M, & Noh, B (2008). The Activity Analysis of Malicious HTTP-based Botnets using Degree of Periodic Repeatability*. Retrieved December 21, 2010 from IEEE Xplore Digital Library Web site: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4725350&tag=1

Tan, P, Steinbach, M, & Kumar, V (2006). Introduction To Data Mining. Boston, Ma: Pearson Education, Inc.

Zeidanloo, H & Manaf A (2009). Botnet Command and Control Mechanisms. Retrieved December 6, 2010 from IEEE Xplore Digital Library Web site: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05380180

Appendix 1 - Proofs

Theorem 1: Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n real numbers. Let m be the minimum value in S and M be the maximum value in S. Let σ be the standard deviation of the n values. Then:

$$\sigma \geq \frac{M - m}{\sqrt{2n}}$$

Proof: Let μ be the mean of the n values. From the definition of variance:

$$\begin{aligned} \sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \\ &= \frac{1}{n}\left[(x_1 - \mu)^2 + (x_2 - \mu)^2 + ... + (x_n - \mu)^2\right] \\ &\geq \frac{1}{n}\left[(m - \mu)^2 + (M - \mu)^2\right] \end{aligned}$$

So, for a given m and M, we have established a lower bound for the variance. The value of the lower bound will depend on μ. For a fixed m and M, define the following function:

$$\sigma(\mu) = \frac{1}{n}\left[(m - \mu)^2 + (M - \mu)^2\right]$$

Since μ is the mean of the n values, we know that $m \leq \mu \leq M$. This means that $\mu \in [m, M]$. We can then use the derivative to determine the value of μ for which $\sigma(\mu)$ becomes minimized within this interval. This will allow us to compute a lower bound for the variance:

$$\begin{aligned} \sigma'(\mu) &= \frac{1}{n}\left[-2(m - \mu) - 2(M - \mu)\right] \\ &= -\frac{2}{n}\left[m + M - 2\mu\right] \end{aligned}$$

Setting the derivative equal to 0:

$$\begin{aligned} 0 &= -\frac{2}{n}\left[m + M - 2\mu\right] \\ 0 &= m + M - 2\mu \\ 2\mu &= m + M \\ \mu &= \frac{m + M}{2} \end{aligned}$$

Taking the second derivative:

$$\sigma''(\mu) = \frac{4}{n} > 0$$

So, $\mu = \frac{m+M}{2}$ is a minimum. Using this value in our first set of equations, we have:

$$\begin{aligned}\sigma^2 &\geq \frac{1}{n}\left[(m-\mu)^2 + (M-\mu)^2\right] \\ &\geq \frac{1}{n}\left[\left(m - \frac{m+M}{2}\right)^2 + \left(M - \frac{m+M}{2}\right)^2\right] \\ &= \frac{(M-m)^2}{2n}\end{aligned}$$

Therefore:

$$\sigma \geq \frac{M-m}{\sqrt{2n}}$$

Note that if the set S contains m, M, and the remaining $n-2$ values are equal to $\frac{m+M}{2}$, then:

$$\begin{aligned}\mu &= \frac{1}{n}\left[m + (n-2)\frac{M+m}{2} + M\right] \\ &= \frac{1}{n}\left[n\left(\frac{M+m}{2}\right)\right] \\ &= \frac{M+m}{2}\end{aligned}$$

For this set of values, we have:

$$\sigma = \frac{M-m}{\sqrt{2n}}$$

This shows that this value is also the greatest lower bound for the standard deviation.

Corollary 1: Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n real numbers and let $T = \{y_1, y_2, ..., y_r\}$ be a subset of S. Let m be the smallest element of T, M be the largest element in T, and σ be the standard deviation of the elements in T. Then

$$\sigma \geq \frac{M-m}{\sqrt{2n}}$$

Proof: By Theorem 1:

$$\sigma \geq \frac{M-m}{\sqrt{2r}}$$

Since T is a subset of S, $r \leq n$, so

$$\frac{M-m}{\sqrt{2r}} \geq \frac{M-m}{\sqrt{2n}}$$

therefore,

$$\sigma \geq \frac{M-m}{\sqrt{2n}}$$

Corollary 2: Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n real numbers and let $T = \{y_1, y_2, ..., y_r\}$ be a subset of S. Let m be the smallest element of T and M be the largest element in T. Let U be a subset of S that contains T, and σ_U be the standard deviation of the elements in U. Then,

$$\sigma_U \geq \frac{M-m}{\sqrt{2n}}$$

Proof: Let U be a subset of S that contains T. Let M_U be the largest element in U, m_U be the smallest element in U, and let σ_U be the standard deviation of the elements in U. By Corollary 1,

$$\sigma_U \geq \frac{M_U - m_U}{\sqrt{2n}}$$

Since U contains T, $M_U \geq M$ and $m_U \leq m$, so

$$\frac{M_U - m_U}{\sqrt{2n}} \geq \frac{M-m}{\sqrt{2n}}$$

therefore,

$$\sigma_U \geq \frac{M-m}{\sqrt{2n}}$$

Appendix 2 - Perl Implementation of HTTP C&C Polling Detection Algorithm

```
#!/usr/bin/perl

# Our parameters to determine if a cluster has a large enough proportion of our
# initial data values and if the computed standard deviation is small enough
our $max_standard_error = 30;
our $cluster_percent_threshold = .75;
our $min_data_values = 10;

# Paranoid - used to prevent infinite loop within k-means subroutine
our $max_iterations = 10;

# Whoever uses this script will need to implement get_array_values() and
# do_something()

my @arr = get_array_values();
if (scalar(@arr) >= $min_data_values) {
    if (ispolling(\@arr)) {
        # Polling activity detected - do something
        do_something();
    }
}

exit 0;

sub ispolling ()
{
        my @arr = @{$_[0]};

        # Use k-means algorithm to partition data into 5 clusters
        my @centroid;
        $centroid[0] = min(\@arr);
        $centroid[1] = max(\@arr);
```

```
        my @results = kmeans(\@arr, \@centroid);

        my $pct = scalar(@results) / scalar(@arr);
        if ($pct >= $cluster_percent_threshold) {
                my $mean = avg(\@results);
                if ($mean >= $min_time_interval) {
                        my $stderr = stderr(\@results, $mean);
                        my $stderr_threshold = $mean / $stderr_factor;
                        if ($stderr_threshold < $min_standard_error) {
                                $stderr_threshold = $min_standard_error;
                        }
                        if ($stderr_threshold > $max_standard_error) {
                                $stderr_threshold = $max_standard_error;
                        }
                        if ($stderr < $stderr_threshold) {
                                return 1;
                        }
                }
        }

        return 0;
}

sub kmeans ()
{
        my @data = @{$_[0]};
        my @centroid = @{($_[1])};

        my @cluster;
        for (my $i = 0; $i < scalar(@centroid); $i++) {
                $cluster[$i] = [];
        }

        my @tmp;
```

```
for (my $i = 0; $i < $max_iterations; $i++) {
        # Clear out the arrays that hold the time interval data
        for (my $j = 0; $j < scalar(@centroid); $j++) {
                @{$cluster[$j]} = ();
        }

        # Assign each time interval value to the appropriate array
        for (my $j = 0; $j < scalar(@data); $j++) {
                my $idx = 0;
                my $distance = abs($data[$j] - $centroid[0]);
                for (my $k = 1; $k < scalar(@centroid); $k++) {
                        my $newdistance = abs($data[$j] - $centroid[$k]);
                        if ($newdistance < $distance) {
                                $distance = $newdistance;
                                $idx = $k;
                        }
                }
                push(@{$cluster[$idx]}, $data[$j]);
        }

        # Calculate new centroid values
        for (my $j = 0; $j < scalar(@centroid); $j++) {
                if (scalar(@{$cluster[$j]}) > 0) {
                        $tmp[$j] = avg($cluster[$j]);
                } else {
                        $tmp[$j] = $centroid[$j];
                }
        }

        # Look to see if previous centroid values match the old
        # centroid values.  If they do, break out of the loop.
        if (scalar(@tmp) == scalar(@centroid)) {
                my $mismatch = 0;
                for (my $j = 0; $j < scalar(@centroid); $j++) {
                        if ($centroid[$j] != $tmp[$j]) {
```

```
                                        $mismatch = 1;
                                        $centroid[$j] = $tmp[$j];
                                }
                        }
                        if (!$mismatch) {
                                last;
                        }
                }
        }

        my $large = 0;
        my $largeval = scalar(@{$cluster[0]});
        for (my $i = 1; $i < scalar(@{$cluster[$i]}); $i++) {
                if (scalar(@{$cluster[$i]}) > $largeval) {
                        $large = $i;
                        $largeval = scalar(@{$cluster[$i]});
                }
        }

        return @{$cluster[$large]};
}

sub max ()
{
        my @arr = @{$_[0]};

        my $max = $arr[0];
        for (my $i = 1; $i < scalar(@arr); $i++) {
                if ($arr[$i] > $max) {
                        $max = $arr[$i];
                }
        }

        return $max;
}
```

```
sub min ()
{
        my @arr = @{$_[0]};

        my $min = $arr[0];
        for (my $i = 1; $i < scalar(@arr); $i++) {
                if ($arr[$i] < $min) {
                        $min = $arr[$i];
                }
        }

        return $min;
}

sub avg ()
{
        my @arr = @{$_[0]};

        # Don't divide by zero if our array is empty
        if (scalar(@arr) == 0) {
                return 0;
        }

        my $total = 0;
        for (my $i = 0; $i < scalar(@arr); $i++) {
                $total += $arr[$i];
        }

        return $total / scalar(@arr);
}

sub stderr ()
{
        my @arr = @{$_[0]};
```

```
        my $mean = $_[1];

        if (scalar(@arr) <= 1) {
                return 0;
        }

        my $total = 0;
        for (my $i = 0; $i < scalar(@arr); $i++) {
                $total += ($arr[$i] - $mean) * ($arr[$i] - $mean);
        }

        $total /= (scalar(@arr) - 1);

        return sqrt($total);
}
```

Appendix 3 - Perl Implementation of Alternative Algorithm

```
#!/usr/bin/perl

# Our parameters to determine if a cluster has a large enough proportion of our
# initial data values and if the computed standard deviation is small enough
our $max_standard_error = 30;
our $cluster_percent_threshold = .75;
our $min_data_values = 10;

my @arr = get_array_values();
if (scalar(@arr) >= $min_data_values) {
    @arr = sort {$a <=> $b} @arr;
    if (ispolling(\@arr)) {
        # Polling activity detected - do something
        do_something();
    }
}

exit 0;

sub ispolling ()
        my @data = @{$_[0]};

        my $spread = ceil(scalar(@data) * $cluster_percent_threshold);
        for (my $i = 0; ($i + $spread) <= scalar(@data); $i++) {
                my $tmp = ($data[$i + $spread - 1] - $data[$i]) / (sqrt(2*scalar(@data))
                if ($tmp < $max_standard_error) {
                        return 1;
                }
        }
        return 0;
}
```