CERIAS Tech Report 2004-09

METHODS FOR CLUSTER-BASED INCIDENT DETECTION

by Brian D. Carrier, Blake Matheny

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Methods for Cluster-Based Incident Detection*

Brian D. Carrier
carrier@cerias.purdue.edu

Blake Matheny
bmatheny@purdue.edu

Center for Education and Research in
Information Assurance and Security - CERIAS
Purdue University
West Lafayette, IN 47907 USA

Abstract

In this paper, we introduce a statistics-based anomaly detection technique for identifying systems that could have been compromised and had trojan executables installed. Attackers frequently install rootkits and other trojan files onto hosts they compromise so they can easily gain access in the future. Many detection systems use signatures to identify unauthorized files, but signatures for all platforms and patch levels do not exist in large-scale environments, such as government and university networks. Our anomaly detection system organizes hosts into clusters based on their files and uses statistics to identify those that should be examined in more detail.

1. Introduction

When a computer is attacked and compromised, it is common for the attacker to modify the system such that he can easily gain access to it in the future and that he can perform actions that will not be observed by users or administrators. One way to modify the system is to replace system executables with trojan versions. This behavior has been reported to Internet mailing lists [13], has been documented by the Honeynet Group [5], and has been observed by ourselves while performing computer forensics.

The typical method of identifying a trojan file is to compare it to a known and trusted version. This works well on a small scale, but does not scale in an environment with thousands of hosts that have different administrators, have different operating systems, and have different patch levels. The administrators may need a copy of every file from every operating system deployed and every patch released.

* Presented at 2nd IEEE International Workshop on Information Assurance. April 8-9, 2004. Charlotte, NC

System executables, such as `/bin/ls` or `/bin/ps`, are commonly modified by an attacker because many rootkits will replace them with ones that hide data from the user. For example, a trojan `/bin/ls` executable can hide the files and directories that the attacker created. Network servers are also modified so that they grant access to unauthorized users. For example, SSH servers can be modified so that they allow the attacker to login if he provides a magic password. The password is compiled into the server executable and logs are not created when this password is used.

In this work, we propose a statistical-based anomaly detection procedure to identify systems that have trojan files on them. The procedure can be used by large-scale networks where not all hosts are regularly patched, monitored, or administered. Some government and university networks fall into this category. Section 2 details previous work in detecting trojan executables, Section 3 describes our process, Section 4 describes our implementation of the process, and Section 5 gives future work.

2. Related Work

Existing work on detecting trojan executables has been oriented towards small-scale environments. The executables on each host are examined to identify if they have been modified and the central theme is to compare the existing files to a known good or known bad object. Existing technology uses both of these comparisons to identify trojan files.

One common method of detecting a malicious file is by calculating a one-way hash of it and comparing it to values stored in a database. Examples of commonly used hash functions include MD5 [11] and SHA-1 [9]. When any bit in the file changes, the MD5 or SHA-1 value will change as well. Therefore, the hash value can be used as a unique identifier, or fingerprint, for the file.

Hash databases exist for both known good and known bad files. The National Institute for Standards and Technology (NIST) has produced the National Software Reference Library (NSRL) that contains hashes of both known good and known bad files [10]. Currently, the majority of the files are based on the Microsoft Windows operating system, so it is not as useful for detecting modifications to Unix files. Another common hash database is Hashkeeper [4], which also contains both known good and known bad files.

Sun has developed the Solaris Fingerprint Database [14] on its web site that allows users to enter the MD5 hash of files into a form and the web site will identify which Solaris file that the hash corresponds to. This allows you to easily identify if the file is a valid Solaris executable. Similar functionality exists with the `rpm` command [12] in Linux. Both of these databases provide a record of known good files.

Databases of only known bad files also exist. The CyberAbuse Rootk(it)ID Project is a collection of known rootkits [2]. Maintaining a database of known bad files is very difficult because it requires you to constantly update the database when new "bad" things are found. Whereas, a database of known good files only requires updating when you update a host with new "good" files.

Another alternative is to calculate the hashes of files before the system is deployed. You can create the database with the `md5sum` command [3], the `md5deep` command [6], or with monitoring tools [15]. Similarly, an investigator can use hashes from a trusted system that is known to have the same patches applied as the suspect system. Hash databases made by these tools can be processed using a simple text search tool `grep`, the lookup option in `md5deep`, or binary search tools [1].

When available, hash databases are very efficient at identifying trojan executables. The problem is having them available for use. Hashes of system executables are typically not calculated before a system is deployed and there are not enough hash databases with all patches to all systems to effectively rely on. Similarly, new trojan files are created every day and the trojan source code will create many executables and hashes depending on what compiler was used.

Another technique to detect trojan files is to use signature analysis. Some trojan files have certain characteristics that can be identified. For example, executables from a system-level rootkit typically use a configuration file that lists which files or processes to hide from the user. The path to the configuration file may be found by looking at the ASCII strings in the executable. Or, the configuration file maybe found on the host by looking in standard directories.

Similarly, some network services are modified to allow an attacker to login with a predefined password that was compiled into the system. Many of the trojan servers will store the MD5 of the back door password so that the password is not easily found when viewing the ASCII strings of the executable. A trojan server maybe detected by looking for a password or a 128-bit hexadecimal value in the executable strings. The chkrootkit program uses these types of signature techniques to find rootkits that are installed on hosts [8].

Signature detection of trojan files has the same limitations as signature detection for network attacks; it only works for known trojan files. Furthermore, it is trivial to obfuscate the strings in the executable file or store configuration files in non-standard places.

The techniques outlined in this section are useful when examining a couple of hosts, but they do not scale for hundreds or thousands of hosts (unless you have the file hashes calculated before the system is deployed). We will next outline steps to reduce the effort required to identify suspect systems when the original hashes are not known and hash databases do not exist for all types and patch levels of your systems.

3. General Detection Theory

In this section, we present our method for identifying sets of hosts with trojan executables installed on them. Our method does not rely on a database of known good or bad hashes or signatures of trojan files. The general process is to cluster similar hosts together based on system executables and analyze the groups instead of the individual systems.

The hashes from the systems must be collected in a secure fashion. A compromised host could send false data when requested for hash values, so trusted media and, ideally, a trusted kernel should be used. The hash values for specific files are calculated on each host and sent to a central analysis station. The hosts with the same operating system are grouped together and analyzed. Patch levels within the operating system version are not used in this analysis. The details of our collection utility are outlined in the Section 4.

The remainder of this section describes the general theory of host clusters and then two detection techniques are presented.

3.1. Clusters of Hosts

Hosts can be organized into sets based on their system executables. We assume that system executables and libraries in a host are changed by administrators who apply patches or by attackers who install trojan files. All systems start in the same set, the one for unpatched systems. When a system is patched and the system executables and libraries are changed, the system is removed from its current set and placed into a new set. For example, let set C_0 be the set of hosts that have no patches applied and let host

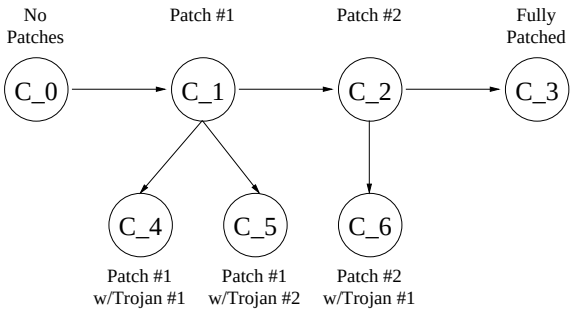

Figure 1. Sets of hosts with four patch sets and three trojan sets

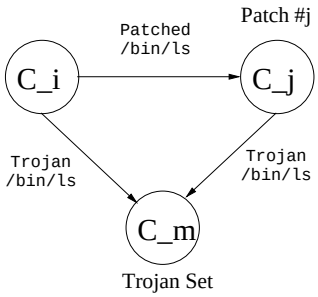

Figure 2. Two patch sets that link to the same trojan set

$h_1 \in C_0$. When a patch is applied to host $h_1 \in C_0$, a transformation removes h_1 from C_0 and adds it to C_1.

$$C_0 = C_0 \setminus h_1$$

$$C_1 = C_1 \cup h_1$$

Sets that contain hosts with only valid executables are **patch sets**. All hosts start in a patch set when they are deployed. A **trojan set** contains hosts that have been removed from a patch set because they had trojan executables installed on them. The installation of trojan files can be thought of as an unauthorized patch because it forces systems to change sets. Figure 1 shows an environment that has four patch sets (C_0, C_1, C_2, and C_3) and three trojan sets (C_4, C_5, and C_6). Hosts in C_4 and C_5 both have the same patch applied, but they have different trojan files and therefore are in different sets.

Note that the clusters can be created in more general terms. Clusters in the above description are created with hosts of the exact same hashes, but they can also be created with hosts that are a given distance, D, from each other. The above description creates clusters with a distance of 0.

3.2. Naive Detection Process

A naive detection method using the sets of hosts is to choose one of the hosts from each set and analyze it. If hash databases or other techniques outlined in Section 2 exist, then they can be utilized. If the host is found to have been compromised, then the entire set is a trojan set and they all have trojan files on them. If trojan files were not found, then the set is likely a patch set. This method allows the administrator to reduce the number of hosts that must be investigated, unless each set has only one host in it. This technique does not help the investigator to focus on specific sets to analyze though.

3.3. Threshold Detection Process

To help the investigator focus on specific systems, we use a threshold. This technique uses the assumption that the trojan sets will contain fewer hosts than the patch sets. The size of each set is examined and if the size of the set is smaller than a threshold T, then the set is identified as a possible trojan set. One host from each of the suspect sets can be examined in more detail to identify if the set is a patch set or a trojan set. An implementation of this could keep a database of confirmed patch sets to prevent future false positives.

Note that this process does not directly rely on the number of files that are updated in each patch. It only relies on the number of hosts that have applied the patch and the number of hosts that have trojan files installed. Although, if a patch modifies the same system executables as a rootkit does, then the trojan set for the two patch sets will be the same. For example, let patch P_j be applied to a subset of the hosts in set C_i. P_j updates only the `/bin/ls` file and it moves the hosts to set C_j. Let there be a rootkit that modifies only the `/bin/ls` file. If the rootkit is installed on hosts in C_i and C_j, then the hosts will both be in the same trojan set, C_m. This example can be seen in Figure 2. This set may not be detected by our algorithm because it is the union of two sets and its size could be larger than the threshold.

The probability of finding the trojan installations increases with the number of types of trojan files that are installed on a patch set. For example, if there are 100 compromised hosts for a given patch set and only two rootkits are installed, then there will be two trojan sets with an average of 50 hosts per set. On the other hand, if there are 10 rootkits among the 100 compromised hosts, then there will be 10 trojan sets and an average of 10 hosts per set. If the threshold is set between 10 and 49, then the algorithm will detect the sets of 10 and not the sets of 50.

4. Implementation

To test the detection methods, we created a sample data set of hashes and implemented the detection process. This section will describe how the data set was created, the methods used for statistical detection, and the detection results.

4.1. Utilities

The first step in the implementation was to develop a collections agent to gather the hashes, a data dumper to format the hash data, and an analysis tool to examine the formated data. Our implementation of these utilities are `sweeper`, `datadumper`, and `stathost`. We briefly discuss these tools below.

Our implementation of the collections agent, `sweeper`, takes a list of hosts to be surveyed and a list of files on the host that should be hashed. `sweeper` used SSH to connect to each host and executed the local version of `md5sum` to calculate the checksums. Once data was "swept" up, the hashes, host names and system types were inserted into a database. The database schema we used is specified in the Appendix.

Our analysis tool, `stathost`, was written in Matlab [7] and therefore the data had to be translated from the database format to a format that Matlab could process. The `datadumper` tool gathered information from the database and put it into a format usable by `stathost`. The details of `stathost` are described in Section 4.3.

4.2. Data Set Creation

Although we wrote `sweeper` to collect hashes from remote systems, we did not have access to an environment large enough to utilize it for testing our process. It would also be difficult to maintain control of a live environment that large so that we knew how many hosts were indeed compromised. Therefore, we created a sample data set of hashes that could be analyzed with our detection process. The data set that was generated was the equivalent to running collection tools on many hosts and storing the sets of hashes in one location. This section outlines how the data set was created.

To generate the data sets, we had to specify the percentage of hosts that are typically compromised, the percentage of compromised hosts that have executables modified, and the number of files that are modified by attackers. We also had to specify how many systems were fully patched, how many were partially patched, and how many had no patches. Unfortunately, we could not find any statistics on these values. Therefore, we estimated these values and varied them to compare the detection algorithm's performance.

The data set had Y types of systems and X systems in each type. For our trials we set $Y = 1$, however this number won't impact the accuracy of the algorithm. Setting $Y > 1$ will simply segregate the data into Y sets, because between systems the duplication of binaries is rare. Also, we expect that the collection tool will sort the data by operating system. Each system had Z hashes collected from it.

To create multiple patch sets, we simulated the process of applying patches. Let A be the percentage of hosts in a type that have no patches applied, B be the percentage of hosts in a type that have all patches applied, and the remainder of the hosts, C percent, be those that have some of the patches applied.

For example, an environment would have $Y = 4$ if Solaris 5.7, Solaris 5.8, RedHat Linux 8.1, and Microsoft Windows XP were deployed; $X = 200$ if there were 200 hosts of each platform; and $Z = 75$ if there were 75 files on each host to examine. For the patch values, an environment would have $A = 30$ if 30 percent of the systems had no patches applied, $B = 30$ if 30 percent of the systems were fully patched, and $C = 40$ if 40 percent of the systems were partially patched.

The data set was generated by calculating Z random hash values for each type of system and populating each host with them. Therefore, all hosts in the operating system type had the same starting hash values. The A percent of hosts that had no patches applied kept the original hash values and formed a patch set. The B percent of hosts that are fully patched had 35 percent of the original hash values changed to new random values and added to a new patch set. All hosts that were fully patched had the same set of hashes. In other words, we assumed that a fully patched system had 35 percent of its system executables modified from the initial installation.

To create the patch sets for hosts with a partial number of patches applied, some of the changes that were applied to the fully patched hosts were applied to the remaining hosts (C percent of the total). The first 20 percent of the partially patched hosts had 15 percent of the changes that were applied to the fully patched systems, the second 20 percent had 30 percent of the changes, the third 20 percent had 45 percent of the changes, the fourth 20 percent had 60 percent of the changes, and the final 20 percent had 75 percent of the changes. These modifications were used to simulate an environment where many hosts had only some subset of available patches applied, as would be the case for many real environments.

Q percent of the hosts that were not fully patched were chosen at random to become compromised systems. 2 percent of the hashes for the selected host were changed. The assumption was that fully patched systems would not be compromised and that only a small fraction of the actual system executables would be modified. For example, a typ-

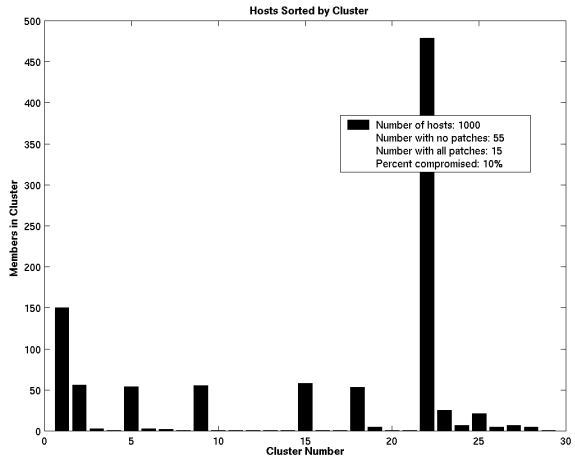

Figure 3. The size of host clusters in a sample data set

ical Unix rootkit only modifies five or six of the executables in the `/bin/` directory.

The modifications to the compromised Q percent of the hosts were subject to alteration by specifying a number of rootkits, R. For all our simulations we specified $R = 5$. The rootkits were equally distributed among the compromised hosts and a random rootkit was created for every R^{th} compromised host. This method was used to simulate some standard rootkits being used across the data set, but non-standard ones being used occasionally as well. One thing that was not considered, for the sake of simplicity, was rootkits that only impact one type of system (e.g. a specific patch level). In reality this type of localized clustering would help the algorithm, because a stronger correlation would be seen.

As an example of an environment in our data set, Figure 3 shows the results from clustering the hosts in one of the data sets based on their file signatures. The x-axis is the cluster number and the y-axis is the number of hosts in a cluster. We can see the 1000 hosts have been broken down to 29 clusters. If the threshold were set to 250, then all clusters except for cluster 22 would be identified as suspect. If the threshold were set to 40, then all clusters besides 1, 2, 5, 9, 15, 18, and 22 would be identified.

4.3. Detection Procedure

The `stathost` application was written in Matlab to process the data sets and detect the compromised systems using the threshold detection technique. The implementation includes a distance metric for determining the relative closeness of hosts, a clustering method for grouping hosts, and an extension for more localized clustering. While the procedure was originally described using set theory, we implemented it using matrices. Before moving onto the theories, we define some commonly terms used.

We begin by populating a matrix A, of size $Z \times (X * Y)$, where Z is the number of hashes and $X * Y$ is the total number of systems being surveyed. A row in A contains the hashes for one file on different systems and a column in A contains the file hashes for one system. We use the notation $|A|$ and $||A||$ to denote the number of rows and columns, respectively, in a matrix or vector A. $A_{:,n}$ denotes all rows of column n of matrix A. $A_{m,n}$ denotes the cell at row m, column n of matrix A. Also, the terms "system" and "host" may be used synonymously.

We define the distance between two hosts H_1 and H_2 represented by column vectors of the same size as follows:

$$d(H_1, H_2) = |H_1| - \Sigma(H_1 == H_2)$$

$H_1 == H_2$ returns a column vector consisting of a zero where rows in H_1 and H_2 were not equal and a one where rows in H_1 and H_2 were equal. $\Sigma(H_1 == H_2)$ returns a sum of the binary values in the new column vector returned by $H_1 == H_2$. Therefore the distance between H_1 and H_2 is 0 when every row is equal, and the distance between H_1 and H_2 is $|H_1|$ when there are no rows of equal value. We say that two hosts are equal with respect to their column vectors when their distance is 0.

Our process takes a matrix A, the desired distance D and a threshold T. The return value is a new matrix, σ, whose nonzero columns correspond to hosts in a cluster with distance D and a size less than or equal to T. The return matrix was calculated using the following method:

$$\sigma(A, D, T) = [x|(\sum_{i=1}^{||A||} \Sigma(\sum_{j=i}^{||A||} d(A_{:,i}, A_{:,j}) = D) \leq T; \\ \forall A_i \in A, A_i \in x)]$$

This method creates σ by iterating over all columns of the matrix A, finding the distance between $A_{:,i}$ and the remaining columns. If the total number of these columns is $\leq T$ then we copy the columns to the same location in σ. This process does not iterate over columns that have already been included in a cluster. The number of members in a cluster can be determined easily by the sum of all nonzero columns of distance D in σ.

While matrix σ and A will both be of the same size, it should be noted that, in general, $\sigma \subset A$ for the nonzero rows of σ. The only case where

$$\forall i, j \;\; \sigma_{i,j} = A_{i,j}$$

is true, is the case where there are no clusters in A whose size is less than or equal to T.

When $D = 0$, a host can fall into only one cluster and this property is important for analysis purposes because hosts in more than one cluster represent duplicate work that may need to be done. Duplicate hosts may also contribute unnecessarily to a cluster's size.

After this procedure, a list of clusters and hosts is returned to the user and a single host from each cluster can be sampled to determine if it is compromised. Discovering a compromised system that is part of a cluster indicates that every host in the cluster is compromised.

The procedure discussed is a simple one. For environments where the majority of hosts are not compromised, the large clusters will contain hosts that are not compromised and some of the smaller clusters will contain hosts that are compromised. For environments where the majority of the hosts are compromised, some of the large clusters may contain hosts that are compromised. For the second scenario, it is necessary to invert the clustering function such that hosts *above* the threshold are detected.

4.4. Detection Results

The simulated data sets were analyzed with the detection process and the results can be found in Table 1. We used 18 different test cases and varied the number of hosts, the percentage of patched hosts, and the percentage of compromised hosts. The threshold, T, was kept constant at five percent of the total hosts so that we could "blind" the results on the simulated data. In a real world situation the threshold would likely not be kept constant because the number of compromised systems would not be known ahead of time. Our detection procedure used a distance, D, of 0 when creating the host clusters.

The number of false positives is the number of clusters that were selected as being compromised, but did not contain compromised hosts. Individual hosts were not counted towards false positives, because if one host in a cluster was a false positive, all hosts in the cluster were false positives and only one would need to be investigated. The accuracy of our procedure can be assessed using the percentage of compromised hosts that were successfully detected and the number of false positives.

The results show that our process detects a high percentage of the compromised systems and has a low number of false positives. When 1 percent of the hosts were compromised, we detected all of them and had no false positives. When 10 percent of the hosts were compromised, we detected between 90 and 100 percent of them and only had one scenario with false positives. When 50 percent of the hosts were compromised, we detected between 40 and 77 percent of them and 4 of the 6 scenarios had false positives. The number of false positives was higher when we had a smaller number of hosts.

To improve the accuracy of the detection method, the threshold can be modified. Generally, to increase the number of systems that are successfully detected, then the threshold should be increased, and to decrease the number of false positives then the threshold should be decreased. Our data shows that even an educated guess for the threshold can yield useful results.

Table 1 supports our hypothesis for the synthesized data set. One should note that for each trial (represented by a row of the table), a different data set was generated with random hash values. Our data shows that the accuracy of detection decreases and the number of false positives increases as the percentage of compromised hosts increases and the threshold is held constant. This is because the size of the clusters with no compromised hosts (patch sets) fall below the threshold and the size of the clusters with compromised hosts (trojan sets) rise above the threshold as the number of compromised hosts increase. However, our tests were done using a static value for the threshold and the accuracy could have been increased by modifying this appropriately. Using a larger threshold will identify the compromised hosts, but it will likely also increase the number of false positives.

This process relies on trends in systems. If no trends can be found, no clustering can occur and the number of systems that need to be surveyed does not decrease. The optimal trend is for all systems to be uniform because detecting anomalies in this case is trivial. The worst case scenario is for every system to be different. Our results show that even in the case when the systems vary (many different clusters), the rate of detection is still very high and the number of false positives remains low. For example, Figure 4 shows a graph representing clusters of hosts from one of the simulations where 50 percent of the hosts are compromised. There were 25 different clusters for only 100 hosts, however there was a 62 percent detection rate and only one cluster with false positives. For this specific scenario, a higher threshold would have improved the results. We used a static threshold of 5 percent of hosts, which is 5 in this scenario, and there are two clusters containing 6 compromised hosts each and one cluster containing 7 compromised hosts. Increasing the threshold by a small amount in this scenario would have found all the compromised hosts and not increased the number of false positives.

Statistical analysis on our data sets consisted of a three part process in which a distance metric determined the host clusters, a threshold identified suspect clusters, and a new data set was produced to determine the accuracy of the specified distance and threshold. If further refinement of a data set is needed, this process allows us to easily change the distance variable, D, and the threshold, T. For our analysis, we did not vary these values and kept $D = 0$ and T as five percent of the the total hosts.

Hosts (X)	Patch Levels			Compromised Hosts (Q)	Threshold (T)	Detected	False Positives (in clusters)
	None (A)	**All** (B)	**Partial** (C)				
100	30%	15%	55%	1 (1%)	5	1/1 (100%)	0
100	15%	30%	55%	1 (1%)	5	1/1 (100%)	0
100	55%	15%	30%	1 (1%)	5	1/1 (100%)	0
100	30%	15%	55%	10 (10%)	5	9/10 (90%)	0
100	15%	30%	55%	10 (10%)	5	10/10 (100%)	0
100	55%	15%	30%	10 (10%)	5	10/10 (100%)	4
100	30%	15%	55%	50 (50%)	5	31/50 (62%)	1
100	15%	30%	55%	50 (50%)	5	29/50 (58%)	2
100	55%	15%	30%	50 (50%)	5	20/50 (40%)	5
1000	30%	15%	55%	10 (1%)	50	10/10 (100%)	0
1000	15%	30%	55%	10 (1%)	50	10/10 (100%)	0
1000	55%	15%	30%	10 (1%)	50	10/10 (100%)	0
1000	30%	15%	55%	100 (10%)	50	94/100 (94%)	0
1000	15%	30%	55%	100 (10%)	50	92/100 (92%)	0
1000	55%	15%	30%	100 (10%)	50	95/100 (95%)	0
1000	30%	15%	55%	500 (50%)	50	386/500 (77%)	0
1000	15%	30%	55%	500 (50%)	50	335/500 (67%)	0
1000	55%	15%	30%	500 (50%)	50	205/500 (41%)	5

Table 1. Detection results using the simulated data sets

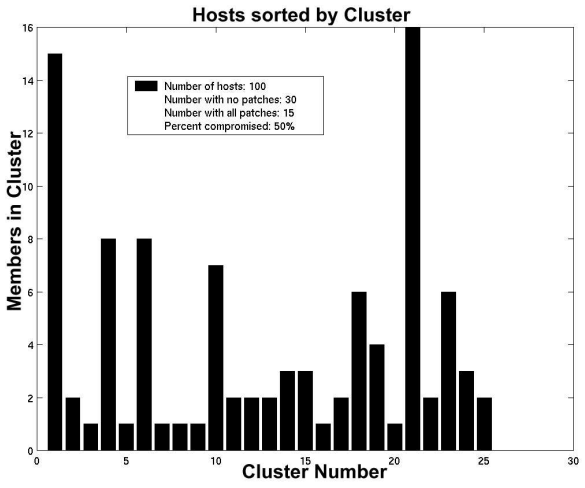

Figure 4. Size of clusters in data set with 50% compromised hosts

5. Future Work

For a practical implementation, several changes to the collection tool, `sweeper`, would need to take place. First, the use of SSH may be inappropriate in some cases because the system may not be running it or the administrator running the collections agent may not have access to the system. An alternative to gathering the needed hashes would be to distribute CDs with `sweeper` on them. System administrators can run the CD on their hosts and send the collected data back for analysis.

Testing these theories on a large scale network would be useful. The lack of statistics for the numbers of compromised systems and patch frequency made it difficult to judge if our results properly reflect reality.

Hash values are useful because they provide unique values for a file, but they are limited because they do not show how different two files are. Measurements other than hash values should be investigated for this type of analysis. Static analysis techniques on executables, such as flow graphs and system call lists, could yield better clustering results.

Another possible expansion upon our existing work would be that of multiple thresholds. Several thresholds would be specified to the clustering function and clusters are returned with a priority. This would allow administrators to prioritize the order in which they investigate the hosts.

6. Conclusion

In this paper, we presented techniques using clusters of hosts to detect the modification of files. These techniques are most useful in large-scale environments where there are a minority number of compromised hosts. However the methods presented can be modified via input parameters, to

work well in other situations. This procedure does not need signatures of every operating system and patch level.

Overall our results showed that the detection rate was high when the threshold was set at an appropriate level and a minority of the systems were compromised. The threshold being used for our trials was inappropriate when the number of compromised hosts was large, which caused a higher number of false positives and a lower detection rate. Despite this, the overall clustering technique was still successful in that it reduced the number of systems needed for a representative sample.

Appendix

Below is a listing of our schema (implemented using MySQL) for sweeper. It should be noted that this schema is a variable one, in that the exact structure for a table is not known until *sweeper* is run, and the applications wanting to be surveyed are known.

```
CREATE TABLE tbl_name (
  hostname TEXT NOT NULL,
  app1 VARCHAR(64),
  ...
  appN VARCHAR(64)
);
```

Where *tbl_name* is the result of running the command `uname -msr` (or something similar for machine without the uname binary) and hostname is the host specified in the hosts list. app1-appN have a 1-1 correspondence with applications listed in the applications list, and the value is the md5 hash.

References

[1] B. Carrier. *The Sleuth Kit*, 2003. Available at: http://www.sleuthkit.org/sleuthkit/.

[2] Cyber Abuse. *Rootk(it)ID Project*, 2003. Available at: http://rk.cyberabuse.org/.

[3] GNU. *Textutils*, 2003. Available at: http://www.gnu.org/directory/GNU/textutils.html.

[4] HashKeeper. *HashKeeper Databases*, 2003. Available at: http://www.hashkeeper.org.

[5] The Honeynet Group. *Know Your Enemy*, 2003. Available at: http://www.honeynet.org.

[6] J. Kornblum. *MD5 Deep*, 2003. Available at: http://md5deep.sourceforge.net/.

[7] The MathWorks. *Matlab*, 2003. Available at: http://www.mathworks.com/.

[8] N. Murilo and K. Steding-Jessen. *chkrootkit*, 2003. Available at: http://www.chkrootkit.org.

[9] National Institute of Standards and Technology (NIST). *Secure Hash Standard - FIPS PUB 180*, May 1993.

[10] National Institute of Standards and Technology (NIST). *National Software Reference Libarary*, 2003. Available at: http://www.nsrl.nist.gov.

[11] R. Rivest. *The MD5 Message-Digest Algorithm*, April 1992. Available at: http://www.ietf.org/rfc/rfc1321.txt.

[12] RPM. *RPM Package Manager*, 2003. Available at: http://www.rpm.org/.

[13] Security Focus. *Incidents Mailing List*, 2003. Available at: http://www.securityfocus.com.

[14] Sun Microsystems. *Solaris Fingerprint Database*, 2003. Available at: http://sunsolve.sun.com/pub-cgi/fileFingerprints.pl.

[15] Tripwire Inc. *Tripwire*, 2003. Available at: http://www.tripwire.com.