VOL. 10, NO. 3, FEBRUARY 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

www.arpnjournals.com

NOVEL IMPLEMENTATION OF A WORM DETECTION SYSTEM USING PROTOCOL GRAPHS

M. R. Muralidharan and Srinivasan Bhargav
Super Computer Education, Bangalore, India
Department of Manipal Institute of Technology, Indian Institute of Science, Manipal, India
Email: murali@serc.iisc.in

ABSTRACT

Computer worms are self-propagating malicious entities that spread throughout a network or the entire internet, causing irreparable damage. More sophisticated worms emerged and a continuous race between attackers and defenders is ongoing. In order to detect the effects caused by these worms on a network, we have implemented an efficient algorithm that uses the Protocol Graph method for the detection and prevention of worm propagation. The system is implemented using C++ and a Perl wrapper, with a frontend. The system will be able to distinguish malicious traffic in real time based on effective statistical methods. Our algorithm is very efficient and we have included a survey of possible implementation methods and the reason as to why our method proves to be unique and efficient.

Keywords: Intrusion detection computer security, computer network management, graph theory, algorithm design and analysis, IP networks, data structures.

INTRODUCTION

Significance of Computer Worms

Worms can cause disruptions such as a Denial of Service attack, and can lead to malicious activities such as information theft or abuse (N.Kawaguchi and et.al., 2006) . In the area of virus and worm modelling, Kephart, White and Chess of IBM performed a series of studies from 1991 to 1993 on viral infection based on epidemiology models (J. O. Kephart and S. R. White. 1991), (J. O. Kephart and S. R. White. 1993). Staniford et al. used the classical epidemic model to model the spread of Code Red right after the Code Red incident on July 19th, 2001. The Code Red worm managed to successfully compromise hundreds of thousands of Microsoft Windows IIS servers. Chen et al. presented a discrete-time version of a worm model that considered the patching and cleaning effect during a worms propagation (Chen, S., & Tang, Y.,2007). In this paper we have described a method of implementation for detecting the presence of HitList worms in an enterprise network. The basic idea for this implementation is derived from Mr. Reiter's article about using Protocol Graphs for HitList worm detection. (Collins, M. P., & Reiter, M. K., 2007).

Implementation of a worm detection system

In this implementation of a worm IDS, we use the Protocol Graph (Collins, M. P., & Reiter, M. K., 2007) method for analysis of flow records. The method describes that we should sample the records for a small duration, *dur*, say every 60 seconds. The number of vertices, V and the size of the largest connected component C are recorded. These records are then analysed to see if they satisfy certain conditions (6) and (7). Those that satisfy the conditions are considered NORMAL traffic and others are possibly malicious. This method offers a very low false positive rate, since it can be calibrated by fine tuning the threshold value t. In our system, we use our own graph search algorithm in order to detect V and C, for a mentioned *dur*. We verify our system, by feeding malware PCAP files from the Netresec group [13], and also the Wireshark Book website. After testing, suitable conditions can be set by modifying the parameters in equations (6), (7) and (9), and the system can be deployed for use real time. The system can be part of a NAT box, or on any system where the flow records can be collected and processed.

Figure-1. Our system in action on a Linux box.

Epidemiological model for worm propagation

In order to understand the nuances of worm propagation, we need to look at literature relating to the earliest models of worm propagation. Monitoring and Early Detection for Internet Worms" by Cliff Zou et al. (Zou, Cliff C.; Gong, Weibo; Towsley, Don; and Gao, Lixin, 2004). gives us an accurate description about the different models used for worm propagation. The paper introduces an idea of a non-threshold based detection system, which relies on trend detection rather than threshold detection. Various approaches have been taken as counter measures to worm propagation, such as the Epidemiological model. This model attempts to predict worm behaviour using differential equations. Assume $I(t)$

©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

denotes the number of infected hosts and N is the total number of suspicious (vulnerable) hosts at a time when worms break out. And Ω denotes the address space in the Internet (2^{32}) and worms scan rate is denoted as η. Now, the number of hosts that are infected at t is expressed as follows. The rate of infection,

$$dI(t)/dt = \eta\Omega I(t)[NI(t)] \quad (1)$$

Solving this equation we get,

$$I(t) = I(0)N/[I(0) + [N - I(0)]e^{-(\eta/\Omega)Nt}] \quad (2)$$

Where $I(0)$ is the number of initially infected hosts = 0.

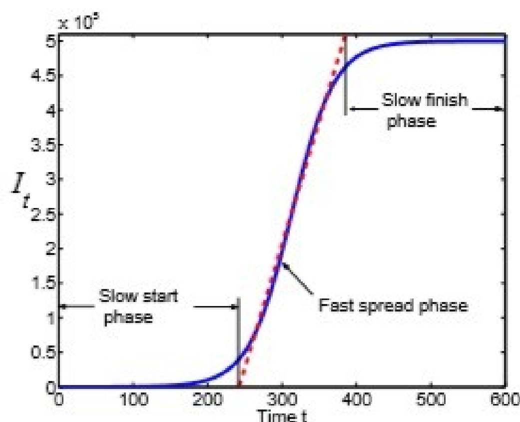

Figure-2. Epidemiological model of worm propagation.

When this distribution is examined over time, the number of infected nodes can be predicted at any given instant. The Epidemiological model can be used as a rough estimate to predict the trend of worm propagation throughout the network. The model assumes that the network is free from human intervention and counter measures and that all the nodes operating in the network are functional and are equally probable to get infected. The rate of infection is when plotted as a function of time approximates to the epidemiological model and hence when we observe this trend, then the network may be suspected of being infected with a worm. However, this model contained inaccuracies which were addressed by a discrete model proposed by Chen et al [10]. The AAWP model is a discrete form of the Epidemiological Model which takes into account that some machines will probably be non operational and thus cannot be infected, or the fact that there is a redundancy in the number of hosts marked infected. Thus it can be observed that the Epidemiological Model overestimates the number of hosts which are infected at any given time[6]. In this model, $I(t + 1)$ can be expressed as,

$$I(t + 1) = I(t) + [N - I(t)][1 - (1 - (1/\Omega))\eta I(t)] \quad (3)$$

Figure-2 shows a comparison between the estimated number of infected hosts, generated by the two models, and a simulation result which shows the variation of the number of infected hosts over time.

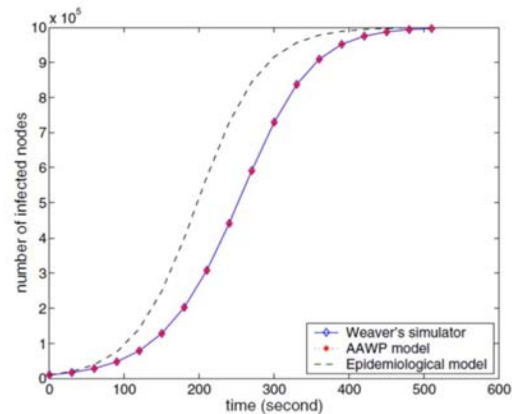

Figure-3. Comparison between AAWP and Epidemiological.

However these models were not capable of accurately predicting new worm behavior because they were largely models(J.Jung and et.al. 2004) dependent on previously stored worm signatures for their analysis. A new and more robust method needed to be developed in order to effectively study worm behavior (Zou, Cliff C.; Gong, Weibo; Towsley, Don; and Gao, Lixin, 2004), (N.Kawaguchi and et.al., 2006,), (S.Staniford-Chen and et.al., 1996).

GRAPH BASED METHODS FOR WORM DETECTION

GrIDS - A graph based intrusion detection system

GrIDS (Graph-Based Intrusion Detection System) is an excellent example of a graph based IDS that was first introduced by Santiford. GrIDS collects data about activity on computers and network between them. GrIDS is capable of analyzing network activity of TCP/IP connections on networks that have tens and thousands of hosts [6][7]. When a worm intrudes a network with GrIDS, the network activity associated with its propagation causes GrIDS to build a tree-like graph. Previously detected data can recognize this tree-like graph as a potential worm. This evaluation might count the number of nodes and branches in the graph. Recognition (detection) occurs when the counts exceed a user-specied threshold, thus reporting a worm. GrIDS is not a scalable model and hence it proves to be ineffective against silent worms, DoS attacks and disruptions or faults in the network.

Protocol graphs

This method of using protocol graphs in order to detect worms was pioneered by M. P. Collins (Redjack),from Software Engineering Institute, Carnegie Mellon

©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

University and the Department of Computer Science,and Michael K. Reiter (CERT), University of North Carolina at Chapel Hill. We have adopted this, and suggested an efficient implementation method and also designed a fully functional system which analyses input data in real time.

A protocol graph is defined as a graph in which the vertices are IP addresses which are unique and the edges are comprised of sets of IP addresses which have communicated with each other. The sets of edges are undirected, therefore an edge from the source to the destination implies that there is an edge from destination to source as well.

Protocol graph is a representation of traffic log for a single protocol, namely HTTP or FTP. It comprises of vertices and edges. Vertices represent a single IP address and edges represent communication between those addresses. Graph size and Largest Component size are two parameters which can be analysed which is extensively used in this methodology. An alarm is raised if the largest component or graph size exceed predicted sizes. An frr (false alarm rate can be set to either moderately or aggressively detect attacks. A visual representation of a Protocol Graph is shown in Figure-4.

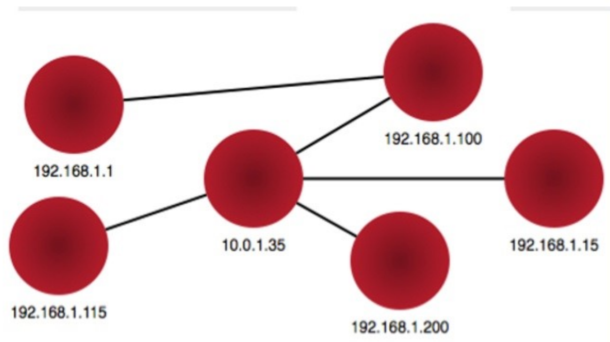

Figure-4. A visual representation of a Protocol Graph with a largest connected component size of six.

Consider a log file (set) $A = A1, An$of traffic records. Each record A has fields for IP addresses, namely source address$A.sip$ and destination address $A.dip$. In addition,$A.port$ denotes the address of the server in the protocol interaction($A.port < A.sip,A.dip >$), though we emphasize that we require $A.port$ it is crucial in our detection or attacker identification mechanisms. Given A, we define an undirected graph $G(A) =< V(A),E(A) >$, where $V(A)$ are the set of vertices and $E(A)$ are a set of edges,

$$V(A) = setof < A.sip,A.dip > \quad (4)$$

$$E(A) = setof < A.sip,A.dip > \quad (5)$$

G(A) is sampled every 60 seconds or some pre-defined amount of time *dur* and the graph is thus constructed. One set of records is created for each half a day and is denoted by $G(A)\pi$; and the largest connected component for that time period is denoted by $Cdur\pi$ and the number of vertices is denoted by $V dur\pi$ [8].

We denote the log file by $A\pi$. log file that is recorded during the interval $\pi[00 : 00GMT, 23 : 59GMT]$ on some specified date.

NetFlow reports flow logs, where a flow is a sequence of packets with the same addressing information that are closely related in time. Flow data is a compact summary of network traffic and therefore useful for maintaining records of traffic across large networks. Flow data does not include payload information, and as a result we identify protocol traffic by using port numbers. Given a flow record, we convert it to a log record A of the type we need by setting A.port to the IP address that has the corresponding service port; e.g., in a flow involving ports 80 and 3946, the protocol is assumed to be HTTP and the server is the IP address using port 80.

These scans are added to the graph and then they need to be rigorously filtered in order to eliminate any unwanted data; namely scans for addresses that don't exist and other interferences from normal data.

Analysis of protocol graphs

Once the algorithm has completed finding the largest connected components and the number of vertices, then the results can be collected and tabulated. The mean and standard deviations for each C and V are plotted separately for each protocol and their corresponding normal distributions are plotted. After plotting these distributions, a threshold is set based on certain probability conditions that will be explained.

After suitable thresholds have been set and the system trained, the system will not require any human intervention. Thus enabling an automated detection process.

More precisely, we divide the day into two intervals, namely $am = [00 : 00GMT, 11 : 59GMT]$ and $pm = [12 : 00GMT, 23 : 59GMT]$. For each protocol we consider, we define random variables $V60sam$ and $V60spm$.

We raise an alarm for a protocol graph if either of these observations hold good:

$$V(A\pi) > \mu_{V\pi dur} + t\sigma_{V\pi dur} \quad (6)$$

$$C(A\pi) > \mu_{C\pi dur} + t\sigma_{C\pi dur} \quad (7)$$

$$P[X \leq x] = 1/2[1 + erf(x - \mu/(\sqrt{2}\sigma)] \quad (8)$$

The false (alarm) rate frr for a given threshold t:

$$Frr = 1 - Pr[Vdur \leq \mu V\pi dur + t\sigma V\pi dur] \quad (9)$$

Similarly, we can evaluate the false alarm rate for the largest connected component variable C. A suitable threshold can then be determined based on the value of the false alarm rate. Additionally, suitable frr can be set by observing the network under normal conditions.

VOL. 10, NO. 3, FEBRUARY 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

Therefore, the threshold t is given by,

$$\sqrt{\ } _t = \quad 2erf^{-1}[0.5 - (frr/2)] \qquad (10)$$

Note that the use of frr/2 in the - equation ensures that each of conditions 6 and 7 contribute at most half of the target frr and consequently that both conditions combined will yield at most the target False Negative Rate .

IMPLEMENTATION OF PROTOCOL GRAPHS

The Protocol Graph HitList Worm detection scheme depends on two variables, namely the number of vertices in a graph and the largest connected component in the graph. Counting the number of connected components in the graph and hence finding the largest connected component. The Union Find Algorithm, which operates on a time complexity of $O(mn)$](Wayne, Kevin, and Robert Sedgewick, 2014), and the rank optimized version which operates on a time complexity of $O(mlog_2n)$](Wayne, Kevin, and Robert Sedgewick, 2014). Here, m is the number of objects and n is the number of times the Union-Find operation is performed, which leads to a worst case of $O(n^2)$ and $O(nlog_2n)$ respectively. We would like to compare this one to our implementation, which is basically a modification of the Breadth First search Algorithm, which allows us to count the Number of Vertices and also the size of the largest connected component in each disjoint set.

Union find algorithm

There are two basic functions that are involved in the UnionFind algorithm, namely:
Make Set: This function creates singleton sets of all the individual elements, initially.
Find: This function is used to find which particular subset, an element belongs to in the collection of disjoint sets.
Union: This function merges two disjoint subsets together to form a single set.

```
1: find(int parent[], int i):
2: if (parent[i] equals -1)
3: return i
4: end if
5: return find(parent, parent[i]);
6:  end find
```

```
1: Union(int parent[], int x, int y):
2: xset = find (parent, x)
3: yset = find(parent, y)
4: parent[xset] = yset
5: end Union
```

Analysis of the Nave implementation

The analysis of a nave union find algorithm is relatively easy, and the total worst case cost can be proven to be $O(mn)$. Where m is the number of elements and n represents the number of times the Union-Find Operation is called (Wayne, Kevin, and Robert Sedgewick, 2014).
Analysis of weighted union find with path compression: The total worst case cost can be proven to be$O(m + nlog_2n)$. Where m is the number of elements and n represents the number of times the Union-Find Operation is called. This algorithm can be improved using a technique called using weights or ranks, an a process called path compression. Path compression is just a flattening of the tree such that we set every nodes parent to its grandparent, and so on hence decreasing the depth and also the complexity of the algorithm (Wayne, Kevin, and Robert Sedgewick, 2014, CMU 15-451/651 (Algorithms), 2014)

There are several methods to optimize the union find algorithm, the first way, called union by rank, is to always attach the smaller tree to the root of the larger tree. Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which increases the depth if the depths were equal. One element trees are defined to have a rank of zero, and whenever two trees of the same rank r are united, the rank of the result is $r + 1$.

```
1: Union(int a, int b):
2: element x = find(a)
3: element y =
find(b)
4: x.parentID = y.ID
5: if (x.parentID equals y.parentID)
6: return
7: end if
8: if (x.rank < y.rank):
9: x.parentID = y.ID
10: else  if ( x.rank >
y.rank):
11: y.parentID = x.ID
12: else
13: y.parentID = x.ID
14: end if
15: x.rank = x.rank + 1
16: end Union
```

```
1: elementfind(element x):
2:   if (x.parent equals -1)
3: return i
4: end if
5: return find(x.parent)
6: end find
```

The set of elements can be represented as a linked list and each member of the list can have an ID and the ID of its parent. Initially all elements parent ID is set to the element ID itself. When two elements are merged, then the parent ID of the second element, is changed to the first elements ID. The first element then becomes the root of the tree. Thus, trees are formed and any element which is a member of the tree, has its parent ID set to the ID of the root element of the tree.

Consider one element, x of any arbitrary set$S$. The MakeSet function runs in linear time as it just involves setting $x.parentID = x.ID$ for m elements initially. The find function takes constant time, as the element being queried

©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

has to return *x.parentID*. The *Union* function on the other hand needs to merge two sets together to create a larger set. Let us consider the element *x*, and say that *x* and *y* should be operated by union. This leads to a formation of set *x,y*. The worst case scenario is that all elements of a set are initially paired with each other resulting in n sets of pairs of elements. (Considering an even number of elements). Next, the union function will be called *n/*2 times to merge the pairs into quads, and so on until all the elements have the same root, and are in the same large set. This takes $log_2 n$ steps, and each step has to iterate through a possible n elements, hence giving an overall worst case complexity of $O(nlog_2 n)$ for unions and a cost of $O(m+nlog_2 n)$ for the entire algorithm (CMU 15-451/651 (Algorithms), 2014). Now, this algorithm is not practical because once the tree structure is formed, the deletion of vertices or edges can be difficult. The functionality of being able to flexibly insert and delete edges without having to collapse and reconstruct the entire data structure is most important in the practical implementation of such a system. This criteria gave us an idea of using an actual graph data structure called the Adjacency List in order to implement a solution to this problem.

Novel implementation of an algorithm to count connected components

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called ”open” and then once examined are placed in the container ”closed”. We have modified this algorithm in order to facilitate the detection of the largest connected component in a Protocol Graph representation of a network.

Algorithm (informal)

1: Mark all nodes as undiscovered initially and mark one start node.
2: If the element sought is found in this node, quit the search and return a result.
3: Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
4: Repeat the process until all the nodes are marked as discovered.

The pseudo code of our implementation is:

1: PGraph(G,v) is
: 2: create a queue
Q
3: create a vector set *V*
4: create an integer *V counter* , *Ccounter*, *Ncomponents*
5: create an array *sizeOfDisjointComponenet*[*V.length*]
6: enqueue *v* onto *Q*
7: add *v* to *V*
8: while *Q* is not empty
loop:
9: Increment *Ccounter*, *Ncomponents*
10: *sizeOfDisjointComponenet*[*Ncomponents*] = *Ccounter*
11: *t* = *Q.dequeue*()
12: if (*t* is what we are looking for then)
13: return *t*
14: end if
15: for all edges *e* in *G.adjEdges*(*t*) loop:
16: *u* = *G.adjV ertex*(*t,e*)
17: if *u* is not in *V* then
18: add *u* to *V*
19: enqueue *u* onto *Q*
20: increment *V counter*
21: end if
22: end for
23: end while
24: return none
25: print Vcounter
26: for all *Ncomponents* in *sizeOfDisjointComponenet*[*V.length*] loop:
27: print *sizeOfDisjointComponenet*[*Ncomponenets*]
28: end for
29: end PGraph

The time complexity of BFS can be expressed as O(E +V) since every vertex and every edge will be explored in the worst case.

RESULTS

In order to test our system we analysed a set of PCAP files from the Wireshark Book website and the Netresec group. These files contain data captured for less than 60 seconds, and hence are ideal for testing our system.

1) Normal HTTP session

This test was done using a PCAP file which contained packets exchanged during a normal HTTP session for a time period of sixty seconds. We observed that there are no large connected components *C*, and the number of vertices *V* are low as well. Hence this kind of traffic will not trigger the detector.

2) Port scan

This test was done using a PCAP file which contained packets exchanged during a port scan being performed in the network in which a worm is present for a time period of sixty seconds. We observed that there are a large number of connected components *C*, and the number of vertices *V* is high and as well. Hence this kind of traffic will trigger the detector.

3) A PCAP containing Malware NAPENTHES from NE-TRESEC

VOL. 10, NO. 3, FEBRUARY 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

www.arpnjournals.com

This test was done using a PCAP file which contained packets exchanged in the network in which a worm is present for a longer time period. This file contains a moderately large no. of vertices, and one large connected component which has 14 nodes, this suggests malware activity. This was featured in Capture the hacker 2013 competition (by Dr. David Day of Sheffield Hallam University).

Table-1. Summary of results.

PCAP file	No. of vertices (V)	No. of disjoint components
httpbrowse.pcap	9	6
portscan.pcap	58	29
Ncapture.pcap	96	40

MERITS AND LIMITATIONS

We can easily compare this system to similar implementations such as GrIDS (and D S.Staniford-Chen and et.al., 1996) discrete Anti Worm (DAW) System by Chen .S. [9]. The methodology of this algorithm was originally pioneered by Collins M.P. (Collins, M. P., & Reiter, M. K., 2007) and they implement this methodology using union-find algorithms.

Discrete anti worm system (DAW)

A DAW agent is deployed on all edge routers of the ISP and a management station that collects data from the agents. Each agent monitors the connection-failure replies sent to the customer network that the edge router connects to. It identifies the offending hosts in the customer network and measures their failure rates.If the failure rate of a host exceeds a pre-configured threshold, the agent randomly drops a minimum number of connection requests. A temporal rate-limit algorithm and a spatial rate-limit algorithm are used to constrain any worm activity to a low level over the long term. A temporal rate-limit algorithm is designed to bound the maximum number of failed requests per day. The temporal rate-limit algorithm constrains both the maximum failure rate and the maximum number of failed requests per day. The spatial rate limit algorithm works in such a way that it thresholds the number of network addresses used.

1) Slows down the network's operation when a worm is detected.
2) There is only one parameter which is used to determine if the connection is worm infested, which can be highly error prone.

Hierarchical worm defense model

This model assumes a tree structure where the internal nodes of the tree are rewalls and leaves are servers vulnerable to worm attacks. The rewalls are assumed to be immune to infections. It is also assumed that we have sensors at the vulnerable hosts that can detect an infection and report it. Once the number of infection reports amongst a nodes (rewalls) children reaches the threshold, the rewall turns on the lter rules protecting all of its children, and alerts its parent that the sub-tree below it is infected but now protected. This escalation of alerts from one level to the next higher level in the hierarchy and protection of sub-trees takes place successively as the threshold for infections is reached at each node.

1) The model can be inefficient because the network has to be realized in the form of a tree.
2) There is no clear indication of any detection methodologies.

Limitations of our algorithm

The system is highly robust, scalable and highly reliable. The idea of using Protocol Graphs for worm detection uses two variables and the joint probability of both the variables exceeding the threshold is calculated. However there are a few limitations of using this algorithm, namely:

1) The system is not tuned to mitigate the spread of the worm. Only detection is possible after which further processing is required.
2) Testing and deployment of the algorithm on the network can take significant time, as the thresholds need to be set carefully.
3) The flow records of the network need to be streamed in at a constant rate, without interruption, as the system works real time.

FUTURE SCOPE

Polymorphic worms

Network worms are malicious programs that spread automatically across networks by exploiting vulnerabilities that affect a large number of hosts. Unfortunately, worms can be polymorphic. That is, they can mutate as they spread across the network through self-encryption mechanisms or code manipulation techniques.

Distributed architecture

In order to be more effective, a distributed detection of Silent worms based on Protocol Graphs is needed. As the network size increases, it becomes more difficult to accumulate and analyse all network logs in the network to a single detection engine. In order to address these challenges, a more distributed architecture is advisable (Bin, L., Chuang, L., Jian, Q., Jianping, H., & Ungsunan, P. 2008) (Chen, S., & Tang, Y., 2007).

Containing or mitigating the spread of worms

Detecting the worm is fairly is simpler compared to containing the worm. The best way to contain the worm and block its access. Therefore, rate limiting algorithms or content blocking algorithms can be used to mitigate the rapid propagation of destructive worms(Chen, S., & Tang, Y.,2007) .

CONCLUSIONS

In summary, we have implemented a novel worm detection system which encompasses of a system which

ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

uses a graph search algorithm to detect largest connected component size and number of vertices in a Protocol Graph representation of the entire network. Suitable tests have been conducted in order to test the effectiveness of the system, and we have proved that it can clearly differentiate between records that contain worm activity and those that do not. This system is extremely flexible and robust, and can run on both Windows and Linux.

REFERENCES

Zou Cliff C., Gong Weib., Towsley Don. and Gao Lixin, ”Monitoring and Early Detection for Internet Worms”.University of Massachusetts, Computer Science Department Faculty Publication Series. Paper 45-77. 2004.
J. O. Kephart and S. R. White. Directed-graph Epidemiological Models of Computer Viruses. In Proc. of IEEE Symposimum on Security and Privacy, pages 343-359, 1991.

J. O. Kephart and S. R. White. Measuring and Modeling Computer Virus Prevalence. In Proc. of IEEE Symposimum on Security and Privacy, 1993.

Bin L., Chuang L., Jian Q., Jianping H. and Ungsunan P. A NetFlow based flow analysis and monitoring system in enterprise networks. Computer Networks, 52(5), 1074-1092. 2008.

J.Jung and et.al. Fast portscan detection using sequential hypothesis testing. In Proc of the IEEE Symposium on Security and Privacy. pages 1-7. 2004.

N. Kawaguchi and et.al. Actm: Anomaly connection tree method to detect silent worms. In Proc. of IEEE AINA 2006, volume vol.1, pages 901-906, 2006.

S.Staniford-Chen and et.al. Grids: A graph-based intrusion detection system for large networks. In Proc. of the 19th National Information Systems Security Conference, pages 361-370, 1996.

Collins M. P. and Reiter M. K. Hit-list worm detection and bot identification in large networks using protocol graphs. In Recent Advances in Intrusion Detection. Springer Berlin Heidelberg. pages 276-295. January, 2007.

Chen S. and Tang Y. DAW: A distributed antiworm system. Parallel and Distributed Systems, IEEE Transactions on, 18(7), 893-906. 2007.

Z. Chen L., Gao. and K. Kwiat. Modeling the Spread of Active Worms, In IEEE INFOCOM, 2003.

Wayne Kevin. and Robert Sedgewick. ”1.5 Case Study: Union-Find.” Case Study: Union-Find. Princeton University, 25 Sept. 2013se. Web. 25 April. 2014.

CMU 15-451/651 (Algorithms), Fall 2013. CMU 15451/651 (Algorithms) Fall 2013. Carnegie Mellon University, 2013. Web. 25 April. 2014.

NETRESEC AB. ”Publicly Available PCAP Files.” Public PCAP Files for Download, 2013.