Proceedings of the 2002 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 17–19 June 2002

Connection-history based anomaly detection

Thomas Toth, Christopher Kruegel

I. Abstract

In the past few years, many vulnerabilities of wide-spread software services, often running on of publicly accessible hosts, have been discovered. These vulnerabilities allow hackers to gain access to those machines, thereby compromising their security.

When a vulnerable service is deployed on large numbers of publicly accessible hosts, software tools (called *worms*) can automate the task of intruding a machine and spreading to new locations. These worms consume a large amount of bandwidth by attempting to find and infect other vulnerable machines and compromise the security and confidentiality of these hosts.

Because of the fact that worms are most of the time implemented as directly executable code, they are processed at very high speed. This allows a worm to quickly spread over a network. Usually, it is too late when one manually detects the presence of a worm – in most cases the whole network has already been infected. This makes an automated response mechanism imperative. In this paper, we present an approach to automatically identify worms and perform damage limitation by firewall rule modification.

II. Introduction

The constant increase of attacks against networks and their resources causes a necessity to protect these valuable assets. Although well-configured firewalls provide good protection against many attacks, some services (like HTTP or DNS) have to be publicly available. In such cases a statically configured firewall has to allow incoming traffic from the Internet without restrictions. The programs implementing such services are often complex and old pieces of software. This inevitably leads to the existence of programming bugs. Skilled intruders exploit such vulnerabilities by sending packets with carefully crafted content to the services. This content causes situations which have not been taken into account by the programmers and allow the intruder to gain access to the machine. Worms like the Morris Internet worm [5] are pieces of executable code that try to replicate themselves by automatic exploitation of vulnerable services running on publicly accessible hosts.

T. Toth: Distributed Systems Group, Technical University Vienna, ttoth@infosys.tuwien.ac.at.

C.Kruegel: Distributed Systems Group, Technical University Vienna, chris@infosys.tuwien.ac.at.

A major problem in connection with worms is the fact that they spread with a very high speed. Generally, it is too late when one notices that a system has been infected, because the worm has already inflicted its damage and continued to replicate itself.

Another problem is that the behavior (and damage functionality) of a worm often cannot be predicted by simply monitoring its activity over a short period of time. Hidden features might exist and worms often use a wide variety of methods for spreading, having several exploits available to target different services. In the best case, worms only perform a denial of service attack against the infected network by scanning it for vulnerable services and attempting to infect other machines. Unfortunately, worms can carry arbitrary pieces of software with them, even remote management tools that allow third parties to access the compromised host. Such a combination of a worm and a remote management tools like Netbus of Back Orifice not only wastes valuable resources but threatens confidentiality. Remote management tools allow to execute arbitrary commands on a host allowing others to sniff information like passwords, credit card numbers or confidential documents from all kind of devices (e.g. keyboard, hard disc, local network traffic).

As a worm is not only a nuisance to the system administrator but can have a serious impact on system security, our claim that a worm infection can be considered as intrusive behavior is justified. Intrusion Detection Systems (IDS) are security tools that are used to detect traces of malicious activities which are targeted against the network and its resources. IDS are traditionally classified as signature based [7] or anomaly based [1]. In signature based systems, an administrator defines malicious patterns that the system has to detect. Anomaly based intrusion systems [8] on the other hand base on the construction of a profile of known good behavior and attempt to monitor deviations thereof.

The problem with misuse based systems is that they require a signature in order to find attacks which is usually only the case when a certain exploit is wide-spread and well known. If an attack is performed against a service with an exploit whose signature is unknown, the signature based IDS does not recognize the malicious behavior. Because of the fact that a worm can appear in a huge number of different flavors, a unique signature or pattern cannot be

ISBN 0-7803-9814-9/$10.00 ©2002 IEEE

given to such a system. Another problem are worms that are new and unknown to the ID creator (or vendor).

Anomaly based IDS traditionally focus on user or program behavior. Profiles of such behavior are created in advance or during normal system operation. New events are then compared against those profiles. This has the advantage of being able to recognize unknown exploits which manifest themselves in an unusual or abnormal way. The disadvantage is that anomaly based IDS raise many false alarms (therefore having a high false positive rate). Normally, every incident should be investigated carefully manually, but if thousands of alarms are generated every day, the use of the IDS is voided.

A well-known intrusion detection system that explicitly deals with the problem of detecting worms is GrIDS [9]. GrIDS is an intrusion detection system that attempts to detect worms by building so called *activity graphs* that represent network connections between hosts and by searching for predefined patterns in these graphs. GrIDS provides support for having constraints on parts of the pattern via assertions and allows the combination (aggregation) of individual nodes. The aim of this system is to be able to handle a large number of hosts as their intended target are large enterprise networks.

Although GrIDS offers many interesting features and is a step into the right direction, it still suffers from several disadvantages.

First, their detection is aimed at large scale detection of worms. This process requires data exchange between many nodes in different networks which results in a considerable bandwidth overhead. To make this mechanism scalable, a data reduction scheme had to be introduced which causes valuable information to be lost.

Second, their worm detection is based on tree shaped patterns with branches at certain nodes (i.e. multiple connections originating at a certain machine). Nevertheless, such branches do not necessarily have to occur in a connection pattern of a worm (although they often do).

Third, the worm detector does not include the packet payload into the correlation process. GrIDS only considers events such as the establishment or closing of a connection or takes very specialized information into account such as the user name that was used for a login. Using such a reduced event base makes correlation possible, but does not make full use of the available information. Detecting similarities in the payload of different packets can enable the system to increase the certainty that observed connections really represent a worm. This would allow the system to lower the false positive rate.

Fourth, GrIDS does not take into account the suspicious occurrence of connections to non-existing services or non-existing hosts. These are strong indicators for the existence of worms that attempt to locate vulnerable services and remote hosts.

Fifth, GrIDS does not include any response mechanism in the case that a worm has been detected. This is a major shortcoming, because by utilizing available components like firewalls, further damage could be prevented.

In this paper, we present a model that addresses the issues mentioned above in connection with GrIDS. Our aim is to protect the uncompromised machines in a network by quickly identifying the behavior of a spreading worm and automatically responding with adequate mechanisms in near real-time. In addition, we discuss the implementation of our proposed mechanisms and present some evaluation results.

III. Requirements

We have defined several requirements for our model that our design has to fulfill. These are enumerated and explained below.

- Automatic worm propagation determination
- Worm detection with a very low false positive rate
- Effective Countermeasures
  - automatic responses in near real-time
  - prevent infected hosts from infecting other hosts
  - prevent non-infected hosts from being infected
- Easy configurability

Automatic worm propagation determination means that the system has to be capable to determine the mechanisms that a worm uses for replicating itself. This is done by examining connections between different hosts together with their temporal relationships.

Worm detection with a very low false positive rate demands that the system should not report an intrusion in the case that there is none. Because of the fact that worms might use arbitrary connection patterns, misuse based IDS (which use predefined patterns) are definitely not adequate. Anomaly based intrusion detection systems which are based on statistical approaches and default behavior profiles usually have a high false positive rate. So neither a pure signature based system nor a pure anomaly based system seems adequate. Only a mixture of both methods is suitable to fulfill this requirement.

Countermeasures have to be performed when an actual intrusion (in the form of a worm) is detected. In this paper, we limit our countermeasures to the appropriate changes of firewall rules. Nevertheless, other response mechanisms might also be suitable. The response mechanisms must be triggered as soon as signs for a spreading worm are spotted. In order to be efficient, they have to be executed at a speed comparable to the speed of the worm propagation itself.

In general, the countermeasures serve two different purposes. The first is to prevent infected hosts from infecting others. This is reasonable, because when the source for further infections can be isolated immediately, the rest of the hosts remain unaffected. As an attacker could have included mechanisms into the malicious worm code to cir-

ISBN 0-7803-9814-9/$10.00 ©2002 IEEE

cumvent such protection approaches, the second requirement demands that non-infected hosts are protected from being infected. When the worm has been faster than the response mechanism on the infected host itself, the majority of hosts could, for example, be saved by disabling access to the vulnerable service. If access to services is not granted to infected hosts, the worm cannot spread anymore.

Easy configurability means that a system administrator must be able to customize the detection algorithm's parameters. This is solved by a configuration file.

IV. Basic Model

For our model, we assume that each node participating in the network has installed a personal firewall which is configurable at runtime. The individual hosts can run an arbitrary operating system as long as it provides such a service. One host per local network segment (i.e. broadcast segment) runs a monitoring tool that puts the network interface into promiscuous mode and sniffs the TCP traffic. Especially important are TCP packets that are used to set up a communication channel (packets with an enabled syn flag in their header).

The monitoring station maintains a history of all recent connections and attempts to find recurring data in connections or connections which do not fit into the picture of normal operation. Notice that interesting patterns are not provided a-priori by external means - instead they are extracted from the connection history. While connections are monitored, a connection profile is built. Connection patterns that indicate spreading worms have to follow the properties listed below.

- Similarity of connection patterns
- Causality of connection patterns
- Obsolete connections

The *similarity of connection patterns* describes the fact that a worm exhibits similar behavior when it attempts to spread from node to node. This is caused by the fact that a worm only contains a certain number of modules that can launch different attacks against vulnerable services. It is very likely that a worm attempts to repeatedly exploit the same vulnerability at different machines, especially because it is often the case that all hosts of a network are all running the same version of a certain service. Even when different modules are at hand, they are limited in number. Therefore it is justified to assume that there will be detectable similarities in the connection patterns. Nimda [6] was one of the first worms that made use of multiple vulnerabilities.

The *causality of connection patterns* means that a certain connection pattern or event depends on the occurrence of another, preceding event. For example, it is obvious that a node has to be infected first, before it starts to act in an infected manner. Although such causality of events or patterns is not a certain sign of an intrusion, it can provide a strong indication. An interesting situation is the observation that the destination node of a certain connection opens a similar connection to another host after a short period of time. No regular service behaves that way with the exception of a DNS server that tries to resolve a request from a client which it cannot handle. In this case, the query has to be forwarded and causes a similar connection to another machine. Causal relationships are of interest because they help to dramatically reduce the search space for pattern matching algorithms as only a few connections that happen after a certain event have to be taken into account.

Obsolete connections are caused by a worm that has successfully compromised a host and attempts to locate new victims which can be infected. Finding other vulnerable services is usually implemented by connecting to services at random IP-addresses. In the case that the random host does not exist or the targeted service is not available, an obsolete connection is recorded. Such a connection is a strong indicator for either a misconfiguration or for a worm searching for new targets. Even when the worm does not use TCP messages to find other hosts, it still has to use some available communication services (e.g. ICMP echo requests or UDP messages) to do so. Such occurrences are also monitored on the network.

Following these three properties, we define the characteristics of the behavior of a worm. The occurrence of events are evaluated according to these principles and are assigned a certain severity. The following section explains the mechanisms that are utilized to rate connections and to find potential offending patterns.

V. Worm pattern detection

Before suspicious patterns can be detected, they need to be defined in some way. As mentioned above, such patterns are often specified in advance (e.g. signature based systems). Our approach is different in the way that we do not provide any specific patterns that should be searched for. Instead, we define a metrics that allows the system to evaluate the likelihood that an observed pattern is malicious (i.e. caused by a worm). This enables the system to distill the behavior of the worm from live connection data.

To make the explanation of the pattern evaluation algorithm easier to follow, we first introduce some definitions.

Definition: A *connection* is a tuple (timestamp, srcHost, srcPort, dstHost, dstPort, data) representing a successfully opened TCP connection at time timestamp from machine srcHost, srcPort to machine dstHost, dstPort. data does not contain all bytes that have been exchanged, but only the first z ones (where z is a configurable number).

A *connection-set* is a set of connections where each connection is assigned a unique number. The numbers are assigned in the order the connections occur.

Definition: A *chain* is a subset of a connection-set. For

ISBN 0-7803-9814-9/$10.00 ©2002 IEEE

the elements c_i ($1 <= i <= l$) of a chain of length l, the following holds.

- For all connections c_i and c_{i+p} of the connection-set, the property $timestamp(c_i) < timestamp(c_{i+p})$ holds (with $p >= 0$).
- For each connection c_i and $i > 1$, the destination of the connection c_{i-1} is the source of c_i.

A chain represents a number of TCP session establishments that are sent consecutively from one host to another. An outgoing connection from a host is an element of the chain only if another host opened a connection to the mentioned host before. Loops (the same host being present twice in a chain) are allowed as the timestamp allows ordering of connections.

Definition A *trail* of a node N consists of all chains that end at node N.

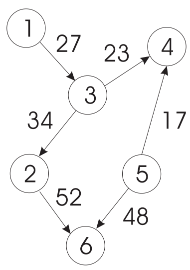

Fig. 1. Example for chains and trails

In the example shown in Figure 1 the numbers associated with the edges denote the time of occurrence of the connection. An example for a chain are the connections at time 27, 34 and 52 from node 1 to node 6 (via node 3 and node 2). The connections at times 27 and 23 do not form a chain because they violate the timestamp condition. The trail of node 6 consists of the chain starting at node 1 and leading to node 6 and the chain that starts at node 5 and leads to node 6.

These definitions allow us to explain the pattern evaluation algorithm in detail below.

A. Combined determination and identification of worm patterns

As mentioned above, a central analyzer collects the connection data of all nodes in a local area network segment. The following data structures are utilized.

- Connection history
- Connection trap list
- Possible worm pattern pool

Connection history: The connection history for each host contains all connection tuples that have been opened **to** that host. This includes data extracted from the first few TCP segments (i.e. the first z bytes). A *ignore list*, which is also provided for each host, allows the specification of events which should not be inserted into the connection history. This a-priori knowledge helps to reduce the storage overhead by focusing on interesting occurrences.

Connection trap list: The connection trap list for each host summarizes the information of its connection history. Its elements are lists, one for each chain that ends at this node (i.e. one element for each chain in this node's trail). The elements of these lists are pairs of (destination port, content) of all previous connections for the corresponding chain. This obviously includes connections that do not directly end at that node, but are in chains which eventually end there.

When an outgoing connection is detected that matches an entry of any element of a list of this connection trap list (i.e. a connection with a destination port and a content that is exactly similar to an element of the trap list), a part of a potential worm pattern has been found. The complete list where the matching element is member of (i.e. one element of the trap list) is then inserted into the *possible worm pattern pool* of the new connection's destination node.

Possible worm pattern pool: This pool contains the lists that have been identified as parts of a possible worm. For each entry in this possible worm pattern pool (i.e. for each list), an anomaly score is calculated to determine whether it is malicious or not. This is done with the the following metrics.

$$anomalyvalue = repeatcount * rfactor + nehost * hfactor + neservice * sfactor$$

where

- *repeatcount* is calculated as the sum of all elements in the list that appear more than once.
- *nehost* is the number of connections opened to distinct non-existing hosts, where the characteristic of the connections have to be present in the currently evaluated chain.
- *neservice* is the number of connections opened to non-existing services at existing servers, where the characteristic of the connections have to be present in the currently evaluated chain.
- *rfactor, hfactor and sfactor* are configurable factors to weight the gathered numbers.

Not all connections to non-existing services / non-existing hosts are counted, but only those which have been observed after the first connection of the corresponding chain under consideration is monitored.

This metric is evaluated for each chain of the trails of all nodes. Notice however, that this calculation has only to be done when a chain changes (i.e. it gets extended with a new connection).

With the *ignore list* described above, many chains can be eliminated that would otherwise have to be considered as parts of worms. A worm pattern is deduced in the case that the anomaly-value of a chain exceeds a predefined, configurable threshold. In such a case, the information about the worm pattern is disseminated to all the other hosts of the network via a broadcast in order to trigger appropriate response mechanisms (as explained below).

B. Example

This subsection presents an example that shows the detection of a worm. Assume that we have 5 nodes, which are numbered from 1 to 5. The following connections have been recorded.

time	src	srcport	dst	dstport	content
27	1	1231	4	80	GET \..\..
28	2	8329	4	25	...
33	4	2215	ns	80	GET \..\..
35	5	9782	3	80	GET index.h
39	4	11251	5	25	contentA
41	5	5214	nh	25	contentA
42	5	28315	3	80	GET \..\..
44	3	18763	2	80	contentA
...	...	...	...	...	...

The shape of the connection graph together with the timestamps and destination ports of every individual connection is shown in figure 2.

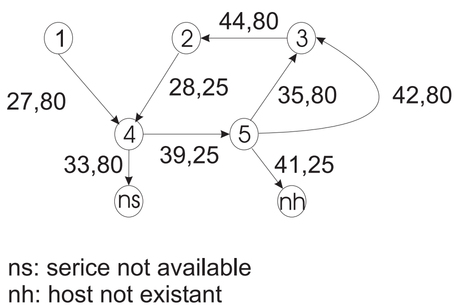

Fig. 2. Worm detection example

The following parameters have been utilized for the detection algorithm.

rfactor	hfactor	sfactor	trigger value
0.3	0.2	0.05	1.0

Events that represent connections to unreachable or unavailable services get a comparably low weight. This is justified by the fact that a service might become unavailable by simply crashing. In contrast to this, connection to non-existent hosts and repeated actions are weighted with a higher value.

The first connection from Figure 2 results in a trap for connections leaving node 4 to port 80 with the recorded content. An additional trap is inserted there after the second connection from node 2 to node 4. A connection from node 4 to a non existing service is performed afterwards, creating an anomaly value of 0.05 at host 4. The connection from 5 to 3 is not important for this example because the causality relationship does not hold (but a trap is inserted at node 3). The connection from 4 to 5 does not have any effect because the content is different from the one before (2 to 4). The connection from 5 to a non existing host results in an anomaly value of 0.55, the one from 5 to 3 in a value of 0.85 and the one from 3 to 2 in a value of 1.15. At this point the worm is recognized, its means of spreading are identified as ports 25 and 80.

VI. Automatic response mechanisms

As stated above, a broadcast is used to disseminate the characteristics of a detected worm. At all hosts, appropriate countermeasures can be launched in reaction to such a broadcast. This is done by reconfiguring the local firewall to block the ports utilized by the worm's spreading mechanisms. This step is necessary to prevent the worm from infecting new hosts.

Additionally, hosts that are identified to be infected insert rules into their firewalls that prevent outgoing traffic to the identified vulnerable services. This helps to prevent the worm from flooding the network with reconnaissance packets when it searches for new victims. The mechanism also helps to protect other hosts that are not directly covered by our proposed approach (e.g. hosts on the Internet).

Even when not all possible ways have been identified that the worm can utilize to spread, the threat has been vastly reduced. When the worm attempts to use an alternative way, it will be detected very soon and the firewalls are updated accordingly. The known mechanisms are no longer effective as they are blocked at the firewall.

VII. Implementation

We have implemented a prototype that realizes the approach described above. A single node runs a network sniffer in promiscuous mode (with full TCP stream reassembly) together with the above explained model. In combination with a dynamic firewall configuration daemon at all hosts this is enough to perform the desired tasks. While the sniffer collects the available network traffic (TCP, UDP and ICMP messages) and detects worm signatures the daemons configure the firewalls at the other hosts accordingly.

Our prototype has been implemented in C on a Linux 2.4.18 (SuSE 8.0) system. Parameters from the model are configurable via a configuration file that is read at startup time. These values are considered constant during normal operation. Because memory is a limited resource, the amount of data that can be stored is not infinite. Therefore, the connection history only stores connection data of a certain period of time. This can be justified because the strength of worms is their ability to spread very quickly over whole networks.

ISBN 0-7803-9814-9/$10.00 ©2002 IEEE

The daemon uses the `iptables` module to insert into and remove firewall rules from the system. These firewall rules can be configured such that they only remain active for a limited amount of time. This is motivated by the fact that hosts that are infected with the worm also have a firewall that has closed the outgoing ports. When the worm activity has stopped (because the personal firewalls have been reconfigured according to the worm patterns) the ports used by the services can be made available again.

The communication channel that is used to broadcast reconfiguration and worm property information over the network is protected. This is done by calculating a MD5 sum of the payload that is sent out from the monitoring station to the individual client daemons and encrypting it with its secret key. The public key of the monitoring station is present at all hosts of the network. Using this public key, the authenticity of the message can be determined by the daemon (that obviously only accepts valid and authentic messages).

VIII. Evaluation

We made experiments in a network that consists of nine Linux hosts where different services (e.g. HTTP, DNS, SSHD) as well as personal firewalls have been installed. The behavior of different worms were simulated with `tcpreplay` [3], including connections to non-existing hosts or services. A single machine was running our prototype implementation with the secure infrastructure as explained above. We used the same parameters as in the example shown above. In all seven cases, our system has been able to reliably detect the worm pattern. Especially the inclusion of the payload has been valuable. This allowed us to identify similar connections (e.g. identical ports) with different payloads as different without causing the system to detect this case as an alarm. As an optimization, we inserted known connections occurring in our network into the ignore-list which then reduced the amount of data that had to be processed.

We collected experimental results with our prototype. The sniffer performed the TCP reassembly of the first 512 bytes which were then included into the connection history. The timeout for the connection history had been set to 120 s. With this scenario the system followed the network activity easily on a heavy loaded network. The average message length that has been broadcasted by the monitoring station was 81 bytes. The calculations of MD5 sums of the request payloads lasted 0.002 ms on average, while the encryption of the MD5 sums lasted 0.085 ms. These measurements were taken on an `Intel Pentium III` with 550 MHz.

The modification of the dynamic firewall rules starting from the point when the monitoring station initiated the action to the activation of the rule lasted 23 ms on average. This justifies our claim that the system can protect all hosts of a network with appropriate speed once a worm has been detected.

IX. Conclusion and further research

In this paper, we have presented the design and implementation of a system that monitors a network for connection patterns that represent spreading worms.

In contrast to misuse based systems, the signatures that specify a worm are not provided a-priori. The system itself identifies the characteristics of the worm by deriving the relevant information by an anomaly based metrics and interesting temporal relationships between different connections. Also connection attempts to non-existing hosts or services provide a high level of confidence that witnessed behavior represents a malicious worm.

In contrast to classic anomaly based systems, we don't build profiles of good behavior and then attempt to detect malicious deviations. This makes our system more resistant against attacks in which an attacker behaves in a way such that his illegal actions are considered normal.

We include a security framework with authentication to ensure that only rules from the legitimate monitoring station are dynamically inserted into the distributed firewalls. Currently, the single monitoring station is a single point of failure. In future versions, a replicating this service should be taken into account in order to make it more fault tolerant.

References

[1] Denning, Dorothy An Intrusion-Detection Model, 1986 In *IEEE Symposium on Security and Privacy*, Oakland, USA

[2] Anderson, James P. Computer Security Threat Monitoring and Surveillance February 1980

[3] tcpreplay http://www.tcpdump.org/cgi-bin/cvsweb/tcpreplay/

[4] Staniford-Chen et al GrIDS - A Graph-Based Intrusion Detection System for Large Networks In *Proceedings of the 19th National Information Systems Security Conference.*

[5] E. Spafford. The Internet Worm Program: Analysis. *Computer Communication Review*, January 1989.

[6] The nimda worm analysis http://aris.securityfocus.com/alerts/nimda/010921-Analysis-Nimda-v2.pdf

[7] Giovanni Vigna and Richard A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. In *14th Annual Computer Security Applications Conference*, December 1998.

[8] Laurent Eschenauer. Imsafe. http://imsafe.sourceforge.net, 2001.

[9] Karl Levitt,Matt Bishop GrIDS design http://seclab.cs.ucdavis.edu/arpa/grids/design.html