ISSN No. 0976-5697

Volume 4, No. 2, Jan-Feb 2013

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A dynamic approach to generate behavior patterns of Virus and Worms for Intrusion Detection System

Shahnawaz Ansari*, Rekh Ram Janghel
Disha Institute of Management and Technology, Raipur, India
shanu.anas81@gmail.com*, janghel1310@gmail.com

Abstract: In today's society people become more and more dependent on computer systems. It is therefore vital that such systems are up and running at all times. One factor that has the power to destroy the availability is computer network attacks (CNA). (CNA are defined as "methods aimed at destroying, altering or obstructing information in computers, computer networks or the networks themselves"). Unfortunately, the Internet show an increasing trend regarding the usage of malicious activities such as intrusion attempts, denial-of-service attacks, phishing, spamming and worms. Some automated attacks can compromise a large number of computers in a short period of time. To try to minimize this threat, it would be nice to have a security system which has the ability to detect new attacks and react on them. This work focuses on seeing how good IDS rules that can be generated automatically based on data logged by a simple honypot. The result will be based on data collected by a network intrusion detection system named SNORT, a low-interaction honeypot named honeyd and a vulnerability scanner named Nessus.

Keywords: Intrusion Detection, Honey pots, Longest common Substring, Worms, Confusion matrix

I. INTRODUCTION

This paper covers issues regarding behavior and implementation of a simple honeypot and the use of such technology in creating IDS rules. A honeypot is a computer that is implemented in a network for the purpose of attracting attackers. This computer has nothing to do with the production network, thus all traffic into the honeypot is by definition malicious [1]. This work focuses on signature generation. At present, the creation of these signatures is a tedious, manual process that requires detailed knowledge of each software exploit that is supposed to be captured. Simplistic signatures tend to generate large numbers of false positives, too specific ones cause false negatives.

The goal is to attract attackers by pretending to be an interesting network. The log files from the honeypot serve as data collectors in conjunction with other widely used data collectors such as tcpdump [4] if needed. A security scanner named Nessus [2] is used to generate traffic towards the honeypot, leaving us with full control of the entire system. We use SNORT [3], a signature based Network intrusion detection system (NIDS) to check if the rules we create are usable. The main goal is to see how good SNORT rules that can be made, with as little user intervention as possible, based on information from the collected data.

II. LITERATURE REVIEW

A. Intrusion Detection System:

Many IDS's in use today are signature based. These IDS' are only capable of detecting already known attacks (attacks which have a signature entry in the database of the IDS).

This is a huge problem when new attacks arrive. A signature based IDS are only capable of detecting alterations of already known attacks at best. Therefore there is an interest in trying to make a rule generating system to automatically generate new rules when new attacks arrive.

In this work we look at the possibility of using a low-interaction honeypot to address this problem. The important question is then if the honeypot logs sufficient information to make rules out of. We propose a measurement method to see how good the rules we create are, compared to original rules alerting on the same threat.

Generally, a good signature must be *narrow* enough to capture precisely the characteristic aspects of exploit it attempts to address; at the same time, it should be *flexible* enough to capture variations of the attack. Failure in one way or the other leads to either large amounts of false positives or false negatives.

Our system supports signatures for the Snort [3] NIDSs. We include Snort here because of its current popularity and large signature repository.

B. Honeypots:

Honeypots are decoy computer resources set up for the purpose of monitoring and logging the activities of entities that probe, attack or compromise them[5][6][7]. Activities on honeypots can be considered suspicious by definition, as there is no point for benign users to interact with these systems. Honeypots come in many shapes and sizes; examples include dummy items in a database, low-interaction network components like preconfigured traffic sinks, or full-interaction hosts with real operating systems and services. Our system is an extension of honeyd [8], a popular low-interaction open-source honeypot. honeyd simulates hosts with individual networking *personalities*. It intercepts traffic sent to nonexistent hosts and uses the simulated systems to respond to this traffic. Each host's personality can be individually configured in terms of OS type (as far as detectable by common fingerprinting tools) and running network services (termed *subsystems*).

C. String-based Pattern Detection Algorithms:

Our system is unique in that it *generates* signatures. In contrast to NIDSs, it cannot read a database of signatures upon startup to match them against live traffic to spot matches.

Thus, the commonly employed pattern-matching algorithms in NIDSs are of no use to us. Instead, the system tries to spot patterns in traffic previously seen on the honeypot: we overlay parts of flows in the traffic and use a longest common substring (LCS) algorithm to spot similarities in packet payloads. Like pattern matching, LCS algorithms have been thoroughly studied in the past. Our LCS implementation is based on suffix trees, which are used as building blocks for a variety of string algorithms. Using suffix trees, the longest common substring of two strings is straightforward to find in linear time [9]. Several algorithms have been proposed to build suffix trees in linear time [10][11].

III. RULE GENERATING SYSTEM

The following sections explain individual aspects of our system in detail.

A. *How to generate SNORT rules:*

In this section we explain how we generate the rules used in the experiments. Before the rule generating can begin, we need a dataset with malicious traffic. This is taken care of by the honeypot and Nessus. We use Nessus to scan a specific service on a specific virtual host on the honeypot. The traffic is also run through the SNORT IDS to see what traffic raises alerts. Then the dataset is edited to only include data SNORT alerted on. This is because we only want to create rules for traffic we know SNORT has a rule for. The reason for this is that we need to have a counterpart in order to measure the differences between the original rules and the new rules. The goal is to see if we are able to create working rules based on information logged by a low-interaction honeypot as honeyd. We will compare the new rules to the originals by measuring the differences based on performance (False positives/False negatives), in addition to ranking each missing field by their importance as we see it. An important part of the rule generating was to make it as automatic as possible, using only information given by the honeypot and the standard way of writing SNORT rules. We had to use some assumptions in the procedure regarding what fields SNORT most likely would use for the attacks we deploy.

B. *Longest Common Substring algorithm:*

We use the LCS algorithm [12] to reduce the number of rules created. By using this algorithm it is possible to create one rule for several similar attacks. This is important because it is a relation between SNORT's processing speed and the number of rules it loads.

The LCS algorithm [12] is used to find the longest string(s) that is a substring or is substrings of two or more strings. There are several ways of implementing this algorithm, such as using suffix trees or dynamic programming (matrix). We chose to use the latter because our strings are short; hence the computational overhead is not so important. It is also the easiest to understand. The problem though is that the LCS is not suited for polymorphic worms [13]. This is because polymorphic worms change too much of its payload for LCS to get a good result out of.

C. *True/False Positive Ratio:*

True Positive Ratio (TPR) is a way of showing how good the IDS is at alerting on real attacks. In our setting we use this to show how good our rules are compared to the originals.

TPR is obtained by the following formula:

$$TPR = \frac{TP}{TP + FN}$$

Where: TP = The number of alerts on malicious traffic, FN = The number of missing alerts on malicious traffic. The total number of intrusions is given by TP + FN.

False Positive Ratio (FPR) shows the proportion of instances, which were not an attack but still were alerted on. FPR is a result of the following formula:

$$FPR = \frac{FP}{FP + TN}$$

Where: FP = The number of alerts on benign traffic, TN = The number of correct decisions on benign traffic. The total number of no-intrusions is given by FP + TN.

A perfect IDS would have TPR = 1 and FPR = 0. This would result in alerts only on malicious traffic, and no alerts on benign traffic.

The confusion matrix in Fig 1 illustrates what FP, FN, TP and TN mean.

		Alert?	
Attack?		YES	NO
	YES	TP	FN
	NO	FP	TN

Figure 1: Confusion matrix

D. *Signature Creation Algorithm:*

The philosophy behind our approach is to keep the system free of any knowledge specific to certain application layer protocols. Each received packet causes system to initiate the same sequence of activities:

a. If there is any existing connection state for the new packet, that state is updated, otherwise new state is created.
b. If the packet is outbound, processing stops here.
c. The model performs protocol analysis at the network and transport layer.
d. For each stored connection:

(a). System performs header comparison in order to detect matching IP networks, initial TCP sequence numbers, etc.
(b). If the connections have the same destination port, System attempts pattern detection on the exchanged messages.
(c). If no useful signature was created in the previous step, processing stops. Otherwise, the signature is used to augment the *signature pool* as described in Section III-.
(d). Periodically, the signature pool is logged in a configurable manner, for example by appending the Bro representation of the signatures to a file on disk.

Figure 2 illustrates the algorithm. Each activity is explained in more detail in the following sections.

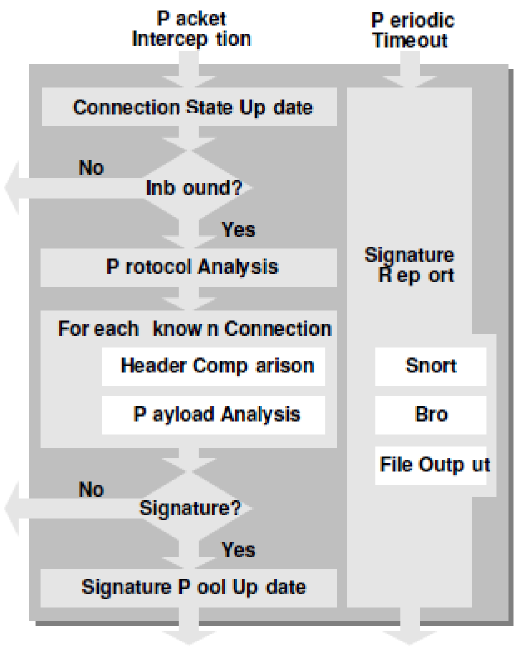

Figure 2: High level overview of Signature creation algorithm.

E. *Connection Tracking:*

The rule generating system maintains state for a limited number of TCP and UDP connections, but has rather unique requirements concerning network connection state keeping. Since our aim is to generate signatures by comparing new traffic on the honeypot to previously seen one, we cannot release all connection state immediately when a connection is terminated. Instead, we only mark connections as terminated but keep them around as long as possible, or until we can be sure that we will not benefit from storing them any longer.

Connections that have exchanged lots of information are potentially more valuable for detecting matches with new traffic. The system must prevent aggressive port scans from overflowing the connection hash-tables which would cause the valuable connections to be dropped. Therefore, both UDP and TCP connections are stored in a two-stage fashion: Connections are at first stored in a “handshake” table and move to an “established” table when actual payload is exchanged.

The system performs stream reassembly: for TCP, we reassemble flows up to a configurable total maximum of bytes exchanged in the connection. We store the reassembled stream as a list of exchanged messages up to a maximum allowed size, where a message is all the payload data that was transmitted in one direction without any payload (i.e., at most pure ACKs) going the other way. For example, a typical HTTP request is stored as two messages: one for the HTTP request and one for the HTTP reply. For UDP, we similarly create messages for all payload data going in one direction without payload data going the other way. Figure 3 illustrates the idea.

F. *Protocol Analysis:*

After updating connection state, The proposed system will create an empty signature record for the flow and starts inspecting the packet. Each signature record has a unique identifier and stores discovered *facts* (i.e., characteristic properties) about the currently investigated traffic independently of any particular NIDS signature language.

The signature record is then augmented continuously throughout the detection process, maintaining a count of the number of facts recorded.

In proposed system, protocol analysis will be performed at the network and transport layers for IP, TCP and UDP packet headers, using the header-walking technique previously used in traffic normalization [14]. Instead of correcting detected anomalies, we record them in the signature, for example invalid IP fragmentation offsets or unusual TCP flag combinations. Note that for these checks, System does not need to perform any comparison to previously seen packets. We refer to a signature at this point as the *analysis signature*.

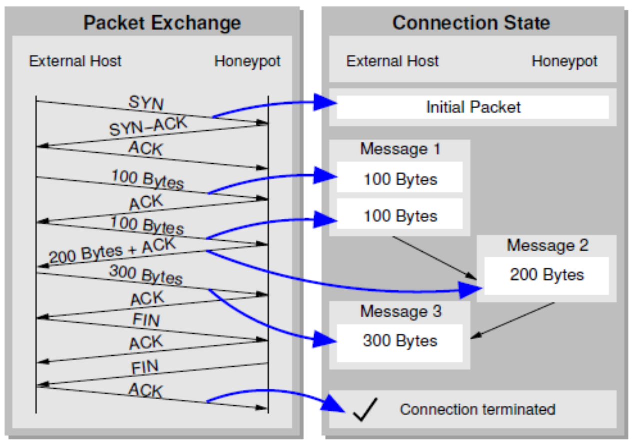

Figure: 3

Fig. 3. A TCP packet exchange (left) and the way System traces the connection (right). The packet initiating the connection is copied separately. afterwards, two 100-Byte payloads are received and assembled as one message. 200 Bytes follow in response, forming a new message. This in turn is answered by another 300 Bytes, forming the final message. The successful completion of the TCP teardown triggers the labeling of the connection as “terminated”.

The proposed system will then performs header comparison with each currently stored connection of the same type (TCP or UDP). If the stored connection has already moved to the second level hash-table, Honeycomb tries to look up the corresponding message and uses the headers associated with that message.

G. *Pattern Detection in Flow Content:*

After protocol analysis, The system proceeds to the analysis of the reassembled flow content. The model applies the LCS algorithm to binary strings built out of the exchanged messages. It does this in two different ways, illustrated in Figures 4 and 5.

a. ***Horizontal Detection:*** Assume that the number of messages in the current connection after the connection state update is *n*. The model then attempts pattern detection on the *n*th messages of all currently stored connections with the same destination port at the honeypot by applying the LCS algorithm to the payload strings directly.

b. ***Vertical Detection:*** Honeycomb also concatenates incoming messages of an individual connection up to a configurable maximum number of bytes and feeds the concatenated messages of two different connections to the LCS algorithm. The point here is that horizontal detection will fail to detect patterns in interactive sessions like Telnet, whereas vertical detection will still work.

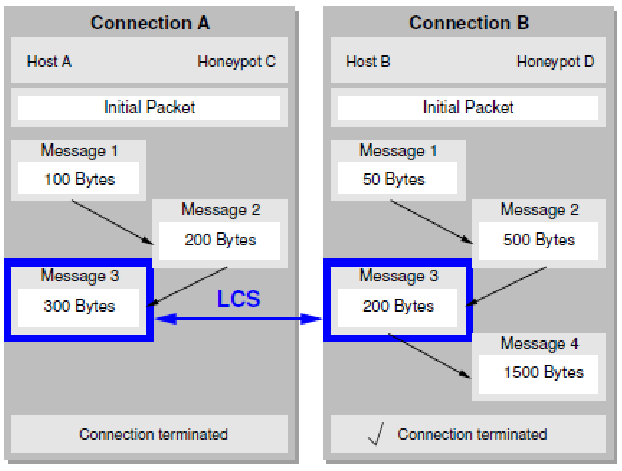

Figure. 4. Horizontal pattern detection.

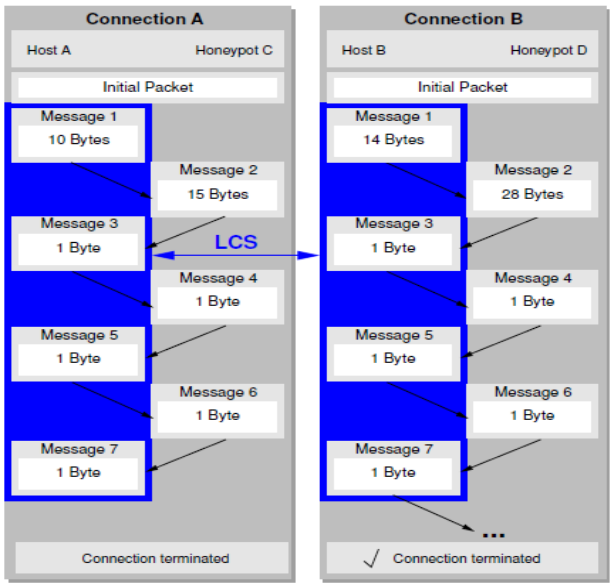

Figure. 5 Vertical pattern detection: for both connections, several incoming messages are concatenated into one string and then passed as input to the LCS algorithm for detection.

In either case, if a common substring is found that exceeds a configurable minimum length, the substring is added to the signature as a new payload byte pattern.

IV. EVALUATION

The implementation will consist of roughly 9000 lines of C code, with about 3000 lines for a separate library implementing the LCS algorithm. The system will be tested on an unfiltered cable modem connection in three consecutive sessions, covering a total period of three days. We will particularly interest in the traffic patterns and signatures created for a typical home-user connection, which can be assumed to be often only weakly protected, if at all.

A. Signature Lifecycle:

If the signature record contains no facts at this point, processing of the current packet ends. Otherwise, System will check how the signature can be used to improve the signature pool, which represents the recent history of detected signatures.

B. Signature Detection:

The proposed system will create a number of signatures for hosts that just probed common ports. These are relatively long; on average they contain 136 bytes. The longest strings are those describing worms.

C. Performance Overhead:

We will measure the performance overhead involved when running proposed system compared to normal honeyd operation.

V. DISCUSSION

In implementation part we are going to show that the proposed system works and produces interesting signatures, and how easily the generated signatures can be used in production environments, and how the system performs under higher load.

However, honeypots generally see only relatively little traffic, so this problem should be manageable.

VI. SUMMARY

In this proposed model, we have presented a system that can produce NIDS signatures automatically by analyzing traffic on a honeypot. The system will produce good-quality signatures on a typical end user's Internet connection. The system is particularly good at producing signatures for worms.

VII. BIBLIOGRAPHY

[1.] Honeynets article. http://project.honeynet.org/papers/honeynet/index.html. Visited Oct2011.

[2.] Nessus, vulnerability scanner. www.nessus.org.

[3.] Snort. http://www.snort.org/. (Visited Nov. 2011).

[4.] Van Jacobson, Craig Leres, and Steven McCanne. Tcpdump, intercept and display communications. www.tcpdump.org. (Visited Dec. 2011).

[5.] C. Stoll, The Cuckoo's Egg. Addison-Wesley, 1986.

[6.] W. R. Cheswick, .An Evening with Berferd, in which a Cracker is lured, endured, and studied,. in Proceedings of the 1992 Winter USENIX Conference, 1992.

[7.] L. Spitzner, Honeypots: Tracking Hackers. Addison-Wesley, 2003. [Online]. Available: http://www.tracking-hackers.com/book/

[8.] N. Provos, .Honeyd - A Virtual Honeypot Daemon,. in 10th DFN-CERT Workshop, Hamburg, Germany, February 2003.

[9.] D. Gusfield, Algorithms on Strings, Trees and Sequences. Cambridge University Press, 1997.

[10.] P. Weiner, Linear pattern matching algorithms. in Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1.11.

[11.] E. M. McCreight, .A space-economical suf_x-tree construction algorithm. Journal of the ACM, vol. 23, pp. 262.272, 1976.

[12.] Dan Gusfield. Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA, 1997.

[13.] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. IEEE Computer Society Washington, DC, USA, 2005.

[14.] M. Handley, C. Kreibich, and V. Paxson, .Network Intrusion Detection: Evasion, Traffic Normalization, end End-to-End Protocol Semantics,. In Proceedings of the 9th USENIX Security Symposium, 2000.