Automated Extraction of Threat Signatures from Network Flows

Piotr Kijewski
CERT Polska/NASK

Introduction

The generation of network threat signatures used in intrusion detection and prevention systems is mostly a manual process, thus prone to errors and slow. Reaction to a new threat must be fast if it is to be effective, and at the same time appropriate so as to reduce any chance of unexpected, even negative side effects. An attempt to achieve this is the goal of automatic extraction of network threat signatures being developed as part of the ARAKIS Early Warning project of the CERT Polska team.

Definition of a network threat signature

A signature may be defined as a representation of a set of features of a threat. These features may vary: for example, they may include information from network packet headers, packet payload, an analysis of the frequency of appearance of certain ASCII characters, system calls used, temporal characteristics of flows etc. From a security point of view the most important aspect of a threat is its method of attack, not the actions performed after infecting a system. This is because a new, unknown method of propagation results in a threat spreading much more widely. Thus, in this article, when we talk of extracting network threat signatures, we refer to the attack signature of the threat - not to the body of the threat (malware).

Characteristics of a good signature

Apart from the **timely generation**, a **high true detection rate** and a **low false alarm rate**, from a practical point of view it is important that the signature derived remains **independent** of application level protocols. This means that the signature can be used in any intrusion detection or prevention software – one that does not have to understand the relevant application level protocol. This allows for the signature to be more universal: a) the signature may be used in a larger number of intrusion detection systems b) it may be applied to new protocols.

From a prevention point of view, it is useful that the attack signature represents the vulnerability utilized. We then have a more general signature, less dependant on the actual exploit used. However, from an informational (early warning) point of view, it is also useful to acquire a signature that identifies a particular exploit. In practice, it is much easier to identify a particular exploit than build a more generic signature that represents the actual vulnerability (ie. encompassing all exploits).

The timeliness and universality criteria suggest that the “de facto” standards in intrusion detection signature should be utilized, as they are the building blocks of current intrusion detection and prevention systems. This standard is the representation of a signature as a sequence of bytes that make up an attack. This allows the signature to be used at the network level, giving added value in an early warning system. If the signature is placed at the network level, a large amount of vulnerable hosts may be immediately protected, until more complex methods of protection are enabled (host based) or the systems patched.

Architecture of a signature extraction system

The first stage of signature extraction is the identification of the network traffic that must be processed. This means that an anomaly must be detected in the traffic. The flows that make up the anomaly must be classified as malicious. Two aspects that are particularly important to look for at this stage are the novelty of an event (anomaly) and its repeatability. This is because it is assumed that a novel threat – an automated one such as a worm or a bot in particular – is characterized by a repetition of actions required for propagation. Once we have a set of flows attributed to a threat identified, we proceed to the next stage and attempt to extract a threat signature. The separation of this extraction process from the process of anomaly detection allows for the use of more complex extraction algorithms, which are too inefficient to function “on-line”. The result of the extraction process is then verified by checking the signatures extracted on traffic that is known to be “normal” or through whitelists of signatures representing “normal” traffic or by vetting by a human expert. Thus we are able to achieve higher quality signatures, not prone to generating false alarms. The next stage is an attempt at classifying the signature, based on its similarity with previously labeled signatures.

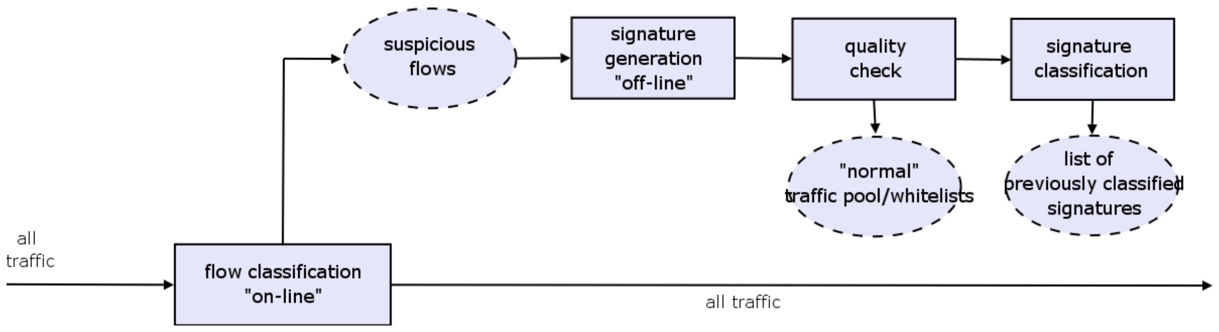
suspicious flows
signature generation "off-line"
quality check
signature classification
"normal" traffic pool/whitelists
list of previously classified signatures
all traffic
flow classification "on-line"
all traffic

Fig 2. Overview of the signature extraction process

Comparing by hashing – identification and signature extraction in one step

The simplest way of attack identification is to compare and catalog packets based on cryptographic hash functions, such as MD5 or SHA1.

Identical packets that are frequently seen from different sources to different destinations may be a sign of a new threat propagating. In this case the hash (and its equivalent payload) is the signature of the threat. Such a technique however is inefficient on a high speed production network, generating a large number of false positives, due to the fact that a large amount of traffic is monitored and it is mostly non-malicious. It becomes more efficient in a honeynet environment, where the traffic is smaller and mostly associated with malicious activity. This method is frequently employed in today's environment, for example in the Internet Motion Sensor [1] project. Nevertheless, this approach has flaws: any small modification of an attack results in a new hash being generated. The "signature" extracted fails to identify the sequence that represents the attack, it identifies the entire packet payload instead.

For better identification of the attack sequence (variant identification), instead of considering an entire packet, it is possible to apply a sliding window mechanism on the packet. A set of sliding window hashes can be computed over a packet or flow, and subsequently be compared across other packets or flows. After a certain threshold is exceeded, based on the count of distinct source host addresses and destination host addresses that are seen across the same hashes, an alarm may be signaled.

However, when using the sliding windows, every packet has a large amount of hashes computed on it, their number being dependant on packet length and the size of the window (if s is the packet size in bytes and β the window length, the amount of hashes is $s - \beta + 1$).

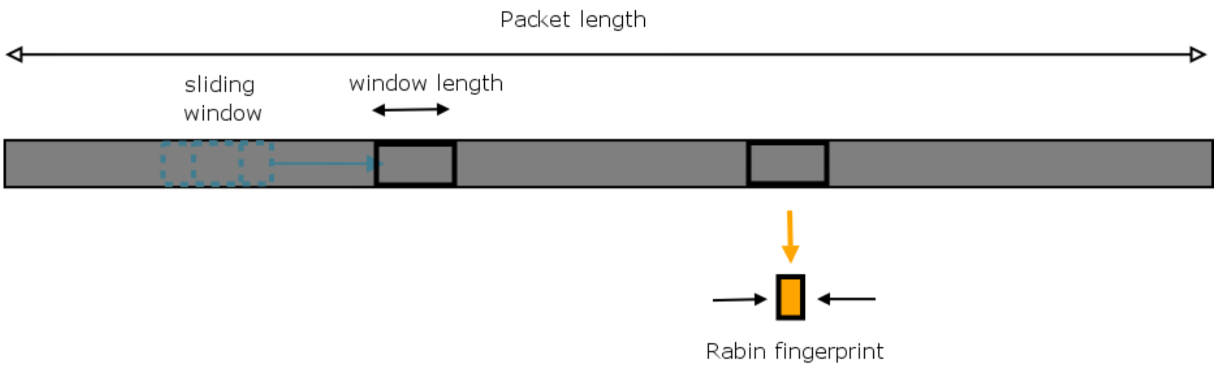

Fig 2. Sliding window across a packet

This large amount of computations means that from an efficiency point of view, the use of cryptographic hash functions becomes problematic. In their place, Rabin fingerprints [2] can be used. Rabin fingerprints are the basis of the Rabin-Karp algorithm, one of the fastest known string searching algorithms. Rabin fingerprints are very efficient over a sliding window, as they allow for a new sliding window hash to be computed based on the previous hash computation. Rabin fingerprinting as a method for detecting network threats was first proposed in [3].

In order to further increase the efficiency it is possible to employ a bitmask, as the basis of a fingerprint sampler. The sampling is value based, and the amount of fingerprints sampled equals $½^k$, where *k* is the length of the bitmask. This method of increasing efficiency is not risk free – sampling increases the chance of missing a threat, because its fingerprints where never sampled. The probability that a threat of length x is detected equals $1-e^{-f(x-\beta+1)}$ where *f* = $½^k$ [3]. For example, for a mask of length of 4 bits, a threat of length 100b and a window length of 32b, the probability of detection equals 98,66%.

The shorter the window length, the higher the probability of detection but also the higher the chance of a false alarm. According to [4], it is necessary to employ a window of length 150b in order to reduce false alarms to zero (mostly due to the long length of Microsoft RPC queries).

Another way of improving efficiency is monitoring only one direction of a connection. If we are interested in attacks that are launched by the side initiating a connection, such as by threats that utilize scanning as a propagation method, we compute hashes on flows from the client side. If we are interested in threats that are spread through servers (passive mode of propagation), Rabin fingerprints can be computed on flows from servers.

Rabin fingerprints, due to their high efficiency can be used not just in honeynet environments but on high speed production networks.

Comparing by hashing can be used not just to classify packets or flows, but to compute signatures. If we have a set of Rabin fingerprints that are representative of a packet (flow), thresholds can be set so as to be associated with a set of fingerprints (for example, the set must be similar across packets by a certain percentage). The signature can then be a selected Rabin fingerprint.

Extracting signatures "off-line"

If Rabin fingerprints can be computed in real-time why separate the signature generator to an off-line module? We suggest this because of:

- Restrictions of a set window size. The lower the window length the higher the probability of attack detection, but the higher the probability of false alarms if we use the fingerprint as a signature. The longer the window length the larger the probability that a true attack sequence was detected but at the price of missing shorter attack sequences.
- Restrictions of sampling. The Rabin fingerprint sampled may not be the best representation of an attack sequence.

- Polymorphism. Polymorphic attacks may be missed by Rabin fingerprinting, as their repeatable sequences will be too short to fill a window.
- Efficiency. More sophisticated algorithms than Rabin fingerprinting are too slow to be used “on-line”.

In order to detect repetitions of a sequence of bytes, techniques other than comparing by hashing can be used. One such technique is the Longest Common Substring (LCS) algorithm, that can be applied across different packets or flows. This algorithm was first used to generate signatures by the honeycomb [5] system, which functions as a honeyd [6] plugin. The problem with honeycomb is its inefficiency when used to monitor a large set of IP addresses and the large amount of signatures generated, which are difficult to manage. This is because all flows stored in memory are compared with each other, without any prior classification.

To address the above problems, we **propose a method that uses both Rabin fingerprinting and LCS**. Rabin fingerprinting is used to classify “on-line” a flow based on the packet payload. This is how an anomaly can be detected: expired flows are **clustered** based on their Rabin fingerprint set similarity according to different **rules** (for example, all expired flows to a certain destination port are grouped together, if they contain 30% of the same fingerprints). Additionally, every rule checks each cluster **with different heuristics**. For example, the amount of distinct source hosts exhibiting similar behavior (ie. in the same cluster) is counted. If a certain threshold is exceeded, the entire cluster is sent for further analysis, which because of its time complexity is carried out “off-line”.

A representative of the “off-line” algorithms is the LCS algorithm. An LCS is computed over a cluster of similar flows. If we compared all the data across all flows we can get a signature of the threat. If we compare packet vs packet across all flows we can get a signature of the exploits used.

This approach allows for the use of small Rabin window sizes, increasing the probability of detection, and at the same time allows for the final signature to be computed by the more elastic LCS algorithm.

Using Rabin fingerprints in this way means that polymorphic attacks can be detected, as long as a honeynet is being monitored, not a production network. This can be achieved through a rule that, when clustering flows based on Rabin similarity, sends for further analysis only clusters made up of one representative. These single member clusters are then grouped together. Thus, polymorphic attacks may be detected based on their lack of Rabin similarity. Of course, computing an LCS on such flows is unlikely (depending on the quality of the polymorphic code generator). However, as shown in [7], it is possible to compute a signature based on disjoint sequences of bytes, because even assuming perfect polymorphism, there must remain fixed value short byte sequences, so that the polymorphic exploit will be able to function correctly. Such algorithms, like the Smith-Waterman algorithm, are being considered by us for implementation. However it is worth remembering that so far, no self propagating code has utilized polymorphic exploits. The closest to such code was the Witty worm [8], that randomly padded exploit packets.

Reducing false alarms

Signature quality is not the most important when signature extraction performs a purely informational role, helping in the understanding of the context of events being observed on a network. However, if we want it to be used in an IDS/IPS system on a production network, it is imperative that the signature does not block any legal traffic. This requires that the signature must be checked for its false positive alarm rate. This can be achieved either by maintaining whitelists of signatures, or pools of normal traffic. When using LCS as the final signature generation algorithm, it would seem that the longer the signature, the higher the probability that it contains an attack sequence and the lower the chance of a false positive. However, if the protocol over which the signature was extracted has long headers, it is possible that the LCS will detect these protocol constants as the longest common sequence of bytes. Thus the longest common substring may not be the best common substring.

Classifying the extracted signature

Classifying the extracted signature allows for better evaluation of new security events on a network. This process can be automated by comparing a new signature with previously classified ones. The process of signature comparison can begin once a signature is extracted. The new signature can be checked for an exact match with an existing labeled one, whether it is a subset of an existing one or if its similar in any way. The most complex process is involved with checking for similarity.

To identify classes of signatures, we use clustering algorithms. An example of such an algorithm implemented by us is a simplified version of the well known *dbscan* [9] algorithm. Historical signatures are periodically clustered with a similarity metric. We picked the Levenshtein distance between strings as the metric. As the signatures can be of different length, the radius of the clusters is variable, dependant on the length of core member of the cluster. This means that the longer the core cluster length the larger the radius of the cluster. This allows for the clustering algorithm to work over both short and long LCS.

Implementation

The implementation of the presented architecture is currently in a testing phase. The basis for the implementation is open source software, snort [10] for on-line analysis and Apache for off-line analysis. Rabin fingerprints have been implemented as a snort plugin (called *rabin*), on top of the standard *flow* and *stream4* plugins. The rabin plugin is the basis of the *flow-classifier* plugin, responsible for the classification of flows. Signature extraction (LCS) is implemented as an Apache module. When the flow-classifier rules detect a threat cluster, the cluster is transferred to an Apache module, together with a list of signature extraction algorithms that they should be subjected to. The communication between snort and Apache is TCP based, which enables the splitting of flow classification and signature extraction tasks between different hosts. The subsequent signature classification process is also off-line and is implemented as a stand-alone PHP5 program. Signature classification is supported by the signature definitions of the *Bleeding Snort* project [11].

Test results

The above implementation was tested in a honeynet environment, with the modified snort process monitoring traffic to a honeyd process, which in turn proxied some of the traffic to an nepenthes/mwcollect [12] process for better vulnerable application emulation. The honeyd process emulated five /26 subnets.

Over a 24 hour period, a total of 775 716 packets were monitored. Only similar packets (defined as 30% common Rabin fingerprints) that came from at least 3 distinct sources in a space of 5 minutes were sent by the *flow-classifier* snort plugin to Apache. The *mod_lcs* Apache plugin generated a total of 408 LCS signatures. These signatures were subsequently grouped into 63 clusters. 63 signatures were then generated over these clusters, one signature per each cluster.

The signatures were then checked against a trace of "normal" traffic, of 2 691 341 packets. As a result, 7 signatures were found that generated false positives. Further vetting of signatures produced a total of 35 signatures, which could potentially be used in a production IDS/IPS system. The dropped signatures involved primarily SMB standard traffic. Out of the remaining 35 signatures, the following were identified manually (with support from the Bleeding Snort ruleset):

- LSA exploit (port 445/TCP) – 10 clusters
- ASN1 exploit (port 445/TCP, port 139/TCP) – 8 clusters
- Winpopup spam (ports 1026-1029 UDP) – 5 clusters
- RPC DCOM (port 135/TCP, 1025/TCP) – 4 clusters
- Shellcode x86 NOOP (port 445/TCP) – 2 clusters
- Port 1026/UDP unknown[1] – 2 clusters
- SQL Slammer (port 1434/UDP) – 1 cluster
- Port 1433/TCP unknown[2] – 1 cluster

[1] Probably related to Winpopup spam

[2] A large amount of short packets to the standard MS SQL Server port - possibly a brute force attempt. It was not identified by any Snort rules.

- NetBIOS query (port 139/TCP) – 1 cluster
- HTTP OPTIONS query (port 80/TCP) – 1 cluster

Example Slammer signature in Snort format:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 1434 (msg:"Slammer"; content:"|04 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 dc c9 b0|B|eb 0e 01 01 01 01 01 01 01|p|ae|B|01|p|ae|B|90 90 90 90 90 90 90
90|h|dc c9 b0|B|b8 01 01 01 01|1|c9 b1 18|P|e2 fd|5|01 01 01 05|P|89
e5|Qh.dllhel32hkernQhounthickChGetTf|b9|llQh32.dhws2_f|b9|etQhsockf|b9|toQhsend|be 18 10 ae|B|8d|E|d4|P|ff
16|P|8d|E|e0|P|8d|E|f0|P|ff 16|P|be 10 10 ae|B|8b 1e 8b 03|=U|8b ec|Qt|05 be 1c 10 ae|B|ff 16 ff d0|1|c9|QQP|81
f1 03 01 04 9b 81 f1 01 01 01 01|Q|8d|E|cc|P|8b|E|c0|P|ff 16|j|11|j|02|j|02 ff d0|P|8d|E|c4|P|8b|E|c0|P|ff 16 89 c6
09 db 81 f3|<a|d9 ff 8b|E|b4 8d 0c|@|8d 14 88 c1 e2 04 01 c2 c1 e2 08|)|c2 8d 04 90 01 d8 89|E|b4|j|10
8d|E|b0|P1|c9|Qf|81|";)
```

Example LSA exploit signature in Snort format:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 445 (msg:"0x31 LSA"; flow:to_server,established;
content:"111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111";)
```

Summary

The signature generation methods presented in the article are a small subset of the possible methods of addressing the problem. They were selected based on their potential practical use in real environments. The current implementation works on honeynets. An important issue not fully covered in the paper is the management of the generated signatures. Moreover, in the case of LCS it is uncertain that the generated signature is really the best and if it's suitable for use in IDS/IPS systems on production networks. Currently, the setup requires manual vetting of signatures. This is especially necessary when the signature clusters are first formed. Over time, new clusters appear in small increments, but the examining of these, due to their smaller amounts is less time consuming.

We are focusing on implementing new algorithms for signature generation within the presented architecture in order to improve signature quality – to enable generic signatures with a low false alarm rate. However, regardless of the signature quality, the presented methods allow for a much better understanding of network security events and can serve as an element of a network threat early warning system.

References

[1] University of Michigan Internet Motion Sensor Project http://ims.eecs.umich.edu/

[2] Michael O. Rabin. "Fingerprinting by random polynomials". Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[3] Sumeet Singh, Cristian Estan, George Varghese, Stefan Savage, "Automated Worm Fingerprinting". In *Proceedings of the 6th ACM/USENIX Syposium on Operating System Design and Implementation (OSDI)*, 2004

[4] P. Akritidis, K. Agnagnostakis, E.P.Markatos, "Efficient Content-Based Detection of Zero-Day Worms". In *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2005.

[5] Christian Kreibich, Jon Crowcroft, "Honeycomb - Creating Intrusion Detection Signatures Using Honeypots". In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*. Cambridge Massachusetts: ACM SIGCOMM, 2003. http://www.cl.cam.ac.uk/~cpk25/honeycomb/

[6] Developments of the Honeyd Virtual Honeyport, http://www.honeyd.org

[7] James Newsome, Brad Karp, and Dawn Song, "Polygraph - Automatically Generating Signatures for Polymorphic Worms", In *IEEE Security and Privacy Symposium*, May 2005

[8] Colleen Shannon, David Moore, "The Spread of the Witty Worm", CAIDA, 2004, http://www.caida.org/analysis/security/witty/

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", *Proc. 2nd int. Conf. on Knowledge Discovery and Data Mining (KDD '96)*, Portland, Oregon, 1996

[10] Snort Intrusion Detection System, http://www.snort.org

[11] Bleeding Snort The Aggregation Point for Snort Signatures and Research http://www.bleedingsnort.com

[12] Nepenthes – finest collection, http://nepenthes.mwcollect.org

[13] Hyang-Ah Kim, Brad Karp, "Autograph: toward automated, distributed worm signature detection", In *Proceedings of the 13th USENIX Security Symposium*, August 2004