Feedback-based Dynamic Traffic Load Balancing for High Speed Network Intrusion Detection

Kyle Wheeler Lambert Schaelicke

August 14, 2003

Abstract

With hacking attempts and the importance of computer security on the rise, Network Intrusion Detection Systems (NIDS) are more valuable than ever. At the same time, increasing network speeds and volume are making it more and more difficult for commodity-hardware to support current NIDS architectures with any accuracy. This paper details a load-balancing algorithm based on feedback, and discusses its advantages over alternative approaches.

1 Introduction

Network traffic speeds and volume are increasing at an exponential rate. Simple ethernet networks have increased in speed tenfold almost every four years for the past several years, far outstripping the ability of network hosts to keep track of every packet of information that goes by on the network. At the same time, computer security is gaining in importance, as computers become responsible for more important things, and exploitation attempts are on the rise. As such, network intrusion detection systems (NIDS) have become more important for detecting such attempts.

In general the speed is necessary to keep up with the volume of large networks trying to communicate with each other through a very few, large connections. However, at such speeds commodity general-purpose computing hardware is incapable of keeping up with every packet that goes by on the network, [3] much less evaluate each packet for possible malicious intent or keep track of streams of data for the same purpose.

To solve this problem, a possible solution is distribution of load. The network traffic can by divided into sections that are each small and slow enough for a single commodity computer to evaluate completely. The trick is to distribute the network traffic in a way that is both useful and allows stream-based analysis, but avoids overloading any single machine and is fortified against malicious traffic patterns.

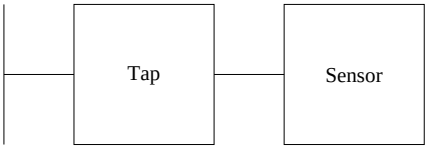

Figure 1: Simple NIDS

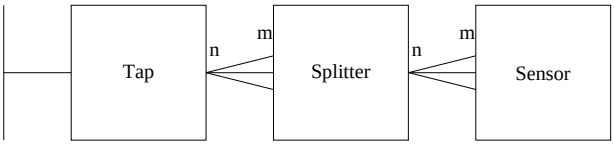

Figure 2: Distributed NIDS

2 The Generic Architecture

The basic network intrusion detection system (NIDS) setup is fairly simple—that is, a NIDS sensor is placed on the network, as illustrated in Figure 1.

High speed links cannot be reliably examined at network speed, and the easiest and most obvious solution is to split the network traffic up into chunks that are small enough that they can be examined in real-time, and then passing those chunks (or slices) of the network traffic on to an array of sensors that can examine the network slices. Such a generic architecture is illustrated in Figure 2. This is based off of the architecture presented in [2], but is more generic.

The connection between the tap and splitter in this case is a 1-to-m relationship, so many splitters may conceivably listen to a single tap. The connection between the splitter and sensor in this case is an n-to-m relationship, so that there can be many splitters, and an unrelated number of sensors (preferably more sensors than splitters). The splitter and sensor blocks are merely conceptual blocks, and may internally consist of many individual splitter or sensor devices (respectively), organized in any fashion.

3 Splitters

There are two basic approaches to how to organize and implement the Splitter section of the generic architecture. One option is that the splitter can be part of the detection framework, and the other is that the splitter can be generic and independent of the detection framework[1].

3.1 Detection Framework Splitting

Splitting the network traffic up in an intelligent manner that is aware of the detection framework and the rules within it can be a great way to limit the amount of traffic. The primary complexity of the sensor is that it is examining the network traffic packet by packet and comparing each one to a long list of rules in order to discover interesting and significant packets. It is conceivable, therefore, that a single, simple rule may allow a splitter to intelligently divide and even weed out traffic to the sensors that is known to be good. The rule implemented in the splitters, especially the first splitter(s), must be very simple, so that they can be implemented easily in hardware, and can be used to decide the destination of each packet at full network speed.

For example, as illustrated in Figure 3, if the only traffic that the sensors know how to analyze is http, ftp, and smtp traffic, then it is easy for a splitter to simply send all http packets one way, all ftp packets another way, all smtp packets a third way, and drop everything else. What this initial splitter sends the packets to could be sensors or more splitters. Any subsequent splitters may be able to do even more logical separation of the network traffic. The more hierarchical logical separations occur, the less traffic is going any one particular route, and the sensors that do full analysis get a very small slice of the original network traffic picked up by the tap. This allows each sensor to be more specialized, being responsible for fewer rules to evaluate each packet with, which improves the sensor's efficiency and capability to handle large amounts of traffic.

There are, however, some drawbacks to this approach to distributing the network traffic across an array of sensors. One drawback is that the hierarchical layout is not easily reconfigurable. If new attack vectors are discovered and new rules for the sensors are written, the layout of the splitters may need to be re-organized, which will likely require that the whole NIDS be taken offline long enough to reconfigure the splitter tree. Another problem with including too much of the analysis system in the splitters is that the tree is not detector agnostic, which is to say, the splitter tree is tied to a particular network traffic analysis software package, and more generally a particular network traffic analysis schema. Thus, changing or adding network traffic analysis software may require reconfiguring the splitter tree, or may be completely impossible, depending on how different the new software is from the old software.

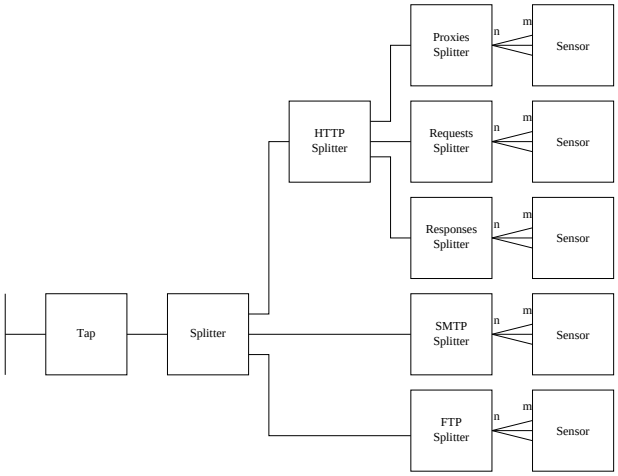

Figure 3: Simple Detection Framework Splitter Tree

The primary drawback, however, of distributing traffic in this manner is narrow denial of service attacks. This architecture cannot easily detect and handle load variations, and so cannot respond effectively to a sudden dramatic increase of traffic that follows a specific path down the splitter tree, in which case a single or a group of sensors may be required to suddenly deal with the full network bandwidth. In generalized traffic, this would likely not be a problem, but traffic mixes may change (requring more or less of the analysis logic to be offloaded into the splitter tree) and in any case this architecture leaves that attack vector open.

3.2 Generic Splitting

The other method of splitting network traffic up is to split it up according to much simpler and more arbitrary criteria—perhaps simple contextual rules like IP address pairs or TCP/UDP port numbers (this will be discussed in more depth in a later section).

The primary benefit of this method is that load changes can be easily handled by changing some simple configuration settings. Load changes can be detected either by traffic counting, or by listening for feedback from the sensor systems. Because it can be easily turned into an adaptive technique, additional load can also be handled by simply adding more hardware at the detection system, without keeping track of configuration settings.

[1] This may make the splitter architecture useful for other kinds of load balancing as well

4 Hashing

Simply splitting the network traffic up into chunks in an unintelligent way is easy—simply splitting based on the length of the packet, or a count of how many 1-bits or 0-bits are in the packet. However, there may be some intelligent requirements that must be placed upon the hashing method.

4.1 Preliminary Network Chunking

The first requirement that may be placed upon the hashing method is that it should preserve connections. Even if you presume that the sensors will communicate to a certain extent, this is still an important requirement because packet streams may be fragmented, and sending different fragments to different sensors only serves to help the attackers. Full connection tracking, however, is both computationally and memory intensive, and opens a possible avenue for an algorithmic attack. Thankfully, fine-grained full connection tracking isn't necessary to keep streams intact, and a simple association based on a few key items in the packet headers may be all that is needed.

The first and most obvious choice that comes to mind is to base the hashing algorithm simply on the IP address, and slice the target network into manageable pieces that way. This, however, doesn't give much information, and means that a denial of service attack against one IP address is also a denial of service attack against one section of the sensor array. Adding TCP or UDP port numbers to the hash (if applicable) is a useful additional way to subdivide the traffic. Not all traffic is TCP or UDP, although there is a limit to how much protocol knowledge can be added to the splitter without losing speed. Some suggestions (their usefulness depends on the speed of the implementation) are ethernet frame type, IP pair (if it's an IP packet), port number (if it's a TCP or UDP packet), and packet type (if it's an ICMP or IGMP packet). More packet fields may be added as desired, but the goal is to stay fast, and to stick to fields that cannot be simply twiddled by attackers to manipulate the hashing algorithm into splitting connection streams.

4.2 Response to Feedback

The hashing algorithm needs to produce more output values than there are sensors, preferably a power of two or so higher than the number of sensors. These values can be indexes into an array to translate them into sensor id's (hardware addresses or IP's or an internal ID number, depending on how the rest of the splitter is implemented). The value of this array is that it allows the values (which represent chunks of the network that contain (probably) several streams) to be reassigned to new sensors at runtime.

When feedback packets are received from the sensors, the array can be manipulated to redirect some (but not all) packet streams away from the sensor(s) in question and toward other sensors, which will hopefully alleviate the problem without losing too much of the state information inherent in receiving connection streams.

5 Hot Spots

The primary problem with all somewhat intelligent network splitting methods is hot spots. Attackers can find (even unintentionally) a specific path through the splitting logic and send large amounts of network traffic through it. At the same time, such paths are generally desireable for analyzing traffic to detect attacks that are not denial of service attacks, and thus logic that streams similar packets should not be simply discarded. The problem of protecting the system from narrow denial of service attacks (or NDoS[2]) is two-fold. First, there must be a way to detect that a path through the logic is being overloaded and second, there must be some way to handle it.

5.1 Detecting Hot Spots

There are two obvious methods for detecting a hot spot in the logic system of a load distributor. The first is packet counting, and the second is feedback. Packet counting is simple but inflexible and requires a homogenous set of sensors, and feedback is very flexible, but is complicated to implement.

5.1.1 Packet Counting

Packet counting is the simple method of keeping a count of the number of packets that have travelled a certain path in the logic system. In this implementation, adding a counter to each entry of the big array the hash-map function uses is sufficient. Then some logic must be applied to keep all the counters (making allowances for what happens when the counters roll-over) within a certain arbitrary delta of each other, thus balancing the traffic. When one counter goes outside that delta on the high end, it can be easily said that the hash value that delta is associated with is a hot spot.

5.1.2 Packet Counting Problems

The problem with packet counting is its inflexibility. For a generic, simple logic with packet counting to work, the sensors it is making decisions about must all be able to handle approximately identical amounts of packets. Even with this allowance, however, packet counting has more

[2] NDoS are denial of service attacks that consist of packets that are very similar or closely associated in the logic of detection systems

problems. It presumes that network traffic will split nicely and evenly over the hash. While such a clean spread is the sign of a good hash function and is probably a generally favorable state, it is not absolutely required at all times. Traffic may spread unevenly over the hash at certain times and evenly over the hash at others. If the traffic never gets to be too much for the sensors, this is not a problem although it may be treated as such by a packet-counting algorithm.

5.1.3 Feedback

Feedback is the more flexible and powerful method of the two. It allows the sensor network to be heterogeneous, and it allows for uneven hashing. The basic idea is that each sensor machine has a method (for example, a modified BPF/LPF kernel module) for detecting when the machine's load is high before it starts to drop packets. This method must, then, be able to emit a notification packet to the splitter informing it of this state. As a final requirement, the method for detecting the machine's load must be able to notify the splitter with enough lead-time for the splitter to effectively redirect traffic to prevent the sensor from dropping packets. This information is obviously not as specific to the splitting implementation as packet counts, and so exactly which of the logic paths that lead to the sensor that is sending feedback may not be obvious (perhaps this could be combined with a form of packet counting to make a good guess which of the hash values assigned to the sensor is the most busy).

5.2 Splitting Hot Spots

Once a hot spot has been detected, something must be done to alleviate the situation and to somewhat rebalance the load. Ideally, this can be done quickly and with a minimum amount of disturbance to existing connection streams.

When there is an indication that one of the sensors is getting an excessive amount of traffic (either from feedback or from packet counts), some of that excessive traffic should be routed away from the affected sensor to other sensors. With packet counts which hash value (or logic path) is producing the most traffic (and there may be more than one) is readily apparent. If the system uses feedback, the exact hash values that are the problem are not obvious, and there must be some way to pick one or more of the hash values that are assigned to that sensor in order to redirect them away to other machines.

A simple method for redirecting traffic is to randomly or sequentially select one of the hash values assigned to the sensor and direct it elsewhere, in the hopes that the reduction in traffic will help. Further load warnings (feedback) will simply cause the system to redirect more and more hash values until the load warnings stop being generated.

This simple solution has the possible problem of "thrashing," where a single hash value is generating more traffic than a single sensor can handle. Thus the machine it is assigned to will generate load warnings continuously, causing many of its assigned hash values to be reassigned, until the high-traffic hash value is reassigned to another machine—whereupon this new machine will start to generate load warnings until the problem hash value is again reassigned. This problem is a narrow hot spot.

5.3 Narrow Hot Spots

Narrow hot spots present the same two problems that hot spots generally present, as well as a third and new problem. Detecting them, and dealing with them are the same, and coalescing when the hot spot cools. Because of the nature of narrow hot spots detection cannot be done on a per-sensor basis, and must be done with some sort of per-hash traffic detection. Packet counts is a good place to start, although some sort of age should be associated with the packet counts to allow for hashes that have had large amounts of traffic but do not currently. An easy method may be to simply make the packet counts report only the number of packets in the last second (have a timer zero out all the counts). Further, more accurate methods for determining relative traffic on the hashes assigned to a single sensor may be developed, although they are limited by the need for speed.

Once a narrow hot spot has been identified, the next question to answer is what to do about it. There are two obvious possibilities, namely using a new hash and round-robin distribution, which can work together rather well.

5.3.1 Hierarchical Hashes

One way to redistribute the traffic that is mapped to a certain hash value is simply to hash the packet again, with a new hash value. For speed purposes, this likely means that multiple hash values should be computed at the same time, regardless of whether they will all be used. These values should be generated by hashes that should be designed to distribute packets in very different ways, in order to avoid simply re-creating the narrow hot spot. How many hash algorithms are used is implementation dependent, although they add complexity and a little bit of time to the processing of every packet.

Each layer of hashing must keep track of traffic the same way as the first layer, and evaluation of feedback load warnings must involve evaluation of all of the hash values that are assigned to the sensor at all layers of hashing (however, this should not be very difficult).

5.3.2 Round-Robin Distribution

It is possible that the narrow hot spot (in some form) will remain, regardless of how many hashing algorithms it has been through. In this case, there is no immediately obvious, easy way to redistribute the flow without breaking streams. That being the case, it is better to break streams than to break the NIDS entirely, and so extremely narrow hot spots should fall back to a round-robin distribution scheme. Thankfully, this fall-back is completely and fully evenly distributed, and thus does not have to be considered when feedback load complaints arrive.

5.3.3 Coalescing

The final issue to consider when dealing with narrow hot spots is how to go back to the way things were before, when the hot spot cools down. Multiple levels of hash are all well and good, but unless, as hot spots go away, the additional layers go away as well, eventually all traffic will be filtered through all hashing layers and will eventually be distributed in a round-robin fashion, which defeats the entire purpose of having intelligent distribution in the first place. Here lies perhaps the most inscrutable of the problems with intelligent load distribution. Perhaps packet counts can again be useful, letting the system know when traffic levels in one layer of hash or in the round-robin distribution has dwindled sufficiently to return to the next layer up. Or perhaps some sort of positive feedback from the sensors woucl be useful for categorizing hash values as being low-traffic. This area of research remains to be explored.

6 Method

For testing and demonstration of this method of load balancing, I intend to use `ssim`, a network routing simulator designed by Tom Slabaugh. By Fall Break, I intend to have the simulator working and understood. By Christmas Break, I intend to have the simulator routing using a generic hash mechanism, without feedback. By Spring Break, I intend to have the feedback mechanism in place, with some simple coalescing handling. By the end of the year, I intend to have done sufficient testing to have gathered data to show that my method works, and by August 2004 I intend to have my Masters Thesis written and ready to defend.

References

[1] Simon Edwards. Vulerabilities of Network Intrusion Detection Systems: Realizing and Overcoming the Risks. http://www.toplayer.com/content/resource/white_papers.jsp, requires registration, May 2002.

[2] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 2002. IEEE Press.

[3] Branden Moore, Thomas Slabach, and Lambert Schaelicke. Profiling Interrupt Handler Performance through Kernel Instrumentation. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, San Jose, CA, October 2003. IEEE Press.