TNT: TRANSPARENT NETWORK AND TRANSPORT LAYER NORMALIZER

TUAN CAO
Yahoo! Research and Development Center, Bangalore, India

VARUN KAPOOR
Yahoo! Research and Development Center, Bangalore, India

Security administrators use network intrusion detection systems (NID systems) as a tool for detecting attacks and misuse, using passive monitoring techniques. However, there are sophisticated attacks which use ambiguities in protocol specifications to subvert detection. In these attacks, the destination endpoint reconstructs a malicious interpretation, whereas the passive NID system's protocol stack interprets the protocol as a benign exchange. There is a dire need for a new software element at the entry point of the network, which transparently modifies network traffic, so as to remove all possible ambiguities. This will ensure that all internal hosts and the NIDS interpret the traffic in a uniform way, hence removing all chances of an attack sneaking past the NIDS, unnoticed and unmonitored. In this paper, we will present the design and implementation of a normalizer whose job is to eliminate evasion and insertion attacks against an NIDS at the transport and network layers.

1. Introduction

NID systems rely on their ability to correctly predict the effect of observed packets on an end-host system in order to be useful. However, as attacks are increasing in sophistication it is becoming difficult to determine when an internal network has been compromised. Examples of these are **Insertion and Evasion attacks** - attacks that use ambiguities in protocol specifications to subvert detection [4].

Examples of these ambiguities are IP fragment reconstruction and the reassembly of overlapping out-of-order TCP byte sequences. The role of the normalizer is to pick one interpretation of the protocols and to convert incoming flows into a single representation that all endpoints will universally interpret.

The normalizer is located at the edge of the network and is meant to be the first internal machine to see the traffic flowing into the site (before any of the

internal end-hosts or NIDSs see it) after the site's firewall/router has determined that a packet's intended destination lies within the internal network [6]. It is an in-kernel module which intercepts TCP/IP traffic from the external network, normalizes this traffic by removing the possible ambiguities, performs logical reassembly and forwards this traffic to its intended destination on the internal network. The normalizer also receives traffic from the internal network, but since its normalizations are half-duplex (for performance reasons), it does minimal connection state processing using this data and forwards it to the external network.

The normalizer's approach to converting ambiguous TCP streams into unequivocal, well-behaved flows lies in the middle of a wide spectrum of solutions. This spectrum contains stateless filters at one end and full transport-level proxies – with a considerable amount of state – at the other. Stateless filters can handle simple ambiguities such as non-standard usage of TCP/IP header fields with little overhead. However, they are incapable of converting a stateful protocol, such as TCP, into a non-ambiguous stream. Full transport-layer proxies lie at the other end of the spectrum, and can convert all ambiguities into a single well-behaved flow. However, the cost of constructing and maintaining two full TCP state machines – scheduling timer events, roundtrip time estimation, window size calculations, etc. – for each network flow restricts performance and scalability. The normalizer's approach to converting ambiguous TCP streams into well-behaved flows attempts to balance the performance of stateless solutions with the security of a full transport-layer proxy. Specifically, the normalizer maintains a small amount of state for each connection but leaves the bulk of the TCP processing and state maintenance to the end hosts. Thus, through interposition, the normalizer can guarantee protocol invariants that enable downstream Network Intrusion Detection Systems to work with confidence.

2. Shortcomings of an NIDS

An NIDS, which predominantly consists of network sensors, is basically a machine in promiscuous mode that logs all traffic that appears on its LAN segment, and alerts the network administrator of any traffic that it classifies as dangerous, based on a pattern-matching engine which is constantly updated with important patterns to watch out for. NID systems identify attacks using passive monitoring techniques to recognize patterns of misuse as they occur within the protocol streams that pass through the firewall, as well as those that originate within the network's perimeter. As such, they have become the second line of defense within an organization, after firewalls. Unfortunately, there are still

some weaknesses that remain unpatched in this layer of defense, which are as follows:

1. An NIDS is, by nature, a device that functions passively, and hence is inherently fail-open.
2. Because of the location of the NIDS within the network topology, it is possible that the NIDS sees some packets which the end-host may not.
3. Because they mainly depend on pattern-matching, and do not process the traffic through a TCP/IP stack, packets can intentionally be crafted in such a way so as to confuse the pattern-matching system, while still correctly being reassembled at the end-host, which may result in the end-host getting attacked.
4. There are some more powerful NIDSs that do perform full reassembly in an attempt to approximate the effect of each packet on an end-host, but sadly, this is just that, an approximation. Studies have shown that different Operating Systems handle cases like overlapping fragments and support for newly introduced TCP and IP options differently, thus making it virtually impossible for the NIDS to know exactly how a particular packet will be reassembled at a given end-host [4].

All these problems led to the rise of a new class of sophisticated attacks, called **Insertion** and **Evasion** attacks. Insertion Attacks are attacks where the NIDS accepts a packet that the end-host rejects. The exact opposites are Evasion Attacks, where the NIDS rejects a packet that the end-host will end up accepting. In both these cases, the NIDS and end-host see 2 different data streams, wherein the NIDS is shown a harmless string, while an attack is being carried out right under its nose.

In this paper, we outline the solution to these, and many more such important problems, in the form of the design and implementation of a Transport and Network Layer Traffic Normalizer. This is an active interposition mechanism that resides at the entry-point of the network and transparently modifies all incoming traffic, so as to remove all possible ambiguities at the TCP and IP levels, which will result in a uniform interpretation of network traffic by all internal hosts and the NIDS.

3. History and Related Work

Network security always has been, and probably will always continue to be, an intellectual battle between network administrators and attackers. While an attacker will first try every possible way to infiltrate the system without being detected, and then attempt to escape from the system without leaving a trace behind, the network administrator directs all his efforts towards using every

resource available at his disposal to develop tools to detect, prevent and analyze forensically, every intrusion attempt. These pressing needs gave birth to the concept of an NIDS, which helped to serve as a burglar alarm for network administrators, by inspecting all network traffic to identify suspicious patterns that may have indicated a network or system attack.

The definitions of Insertion and Evasion were first mentioned in the paper 'Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection' [4]. The word 'Ambiguous' was defined by Ptacek and Newsham as "if an important conclusion about a packet cannot be made without a secondary source of information". The term 'Normalization' first appeared in the paper 'Network Intrusion Detection: Evasion, Traffic Normalisation, and End-to-End Protocol Semantics' [2]. In this paper, Mark Handley and Vern Paxson introduced *norm*, a user level normalizer for IP, TCP, UDP and ICMP, that used libpcap to capture packets, and a raw socket to forward them. It had been tested under FreeBSD, and was released publicly in the summer of 2001, via *www.sourceforge.net*. The basic idea of traffic normalization was simultaneously introduced in the form of a *transport scrubber* [1]. A *transport scrubber* supports downstream passive network-based intrusion detection systems by converting *ambiguous* network flows into *well-behaved* flows that are unequivocally interpreted by all downstream endpoints. A current implementation of the transport scrubber exists as a modified FreeBSD kernel. These two approaches focused on defending against attacks that primarily depended on ambiguous TCP retransmissions, as defined above.

Although *norm* could normalize a TCP traffic stream at 100,000 packets-per-second using a memory-to-memory copy, it was, after all, a user-level implementation, and its authors, Mark Handley and Vern Paxson, in their paper, suggested that an in-kernel implementation of the normalizer would provide better performance, which is one of the improvements we have made over *norm*. Aside from being in-kernel and avoiding copying data back and forth between application-space and kernel-space, we have also implemented all our IP and TCP reassembly "logically". This further contributes to the fulfillment of our overall goal of upholding high performance of the network, described in more detail in section 5.8 [High Performance]. Also, as compared to *norm*, we are a more transparent entity in the network, thus implying less configuration hassles for the network administrator, and more security through a lack of our presence being detectable by machines from both, the external and internal network. More details on this can be found in section 5.1 [MAC Transparency]. Finally, we wanted to propose a good solution for companies wanting a self-contained and independent hardware box that would be ready-to-deploy in a production network as one component of a site's overall security setup, and we have achieved that through our design, which works off of its own customized mini-

network stack and abides by our state diagram, which serves to enforce all our security policies in a reliable and efficient manner.

Another line of thinking was introduced as a solution to the previously-mentioned problems of NIDSs in the paper 'Active Mapping: Resisting NIDS Evasion Without Altering Traffic'[16], which focused on "eliminating TCP/IP based ambiguity in a NIDS' analysis with minimal runtime cost, by efficiently building profiles of the network topology and the TCP/IP policies of hosts on the network, and then making the NIDS use the host profiles to disambiguate the interpretation of the network traffic on a per-host basis."

The disadvantages of the Active Mapper, as compared to our normalizer are:

1. They have a need, in general, to be aware of IP and MAC addresses, which, as they themselves admit, can lead to real-world problems when it comes to dealing with issues like NAT and DHCP. We, however, face no such problems, since we are transparent at the TCP, IP and MAC layers, and only classify traffic as internal or external based on the interface on which it comes in.
2. Maintenance and integration with a site's existing security setup seems to be a problem with them, since they need to be given permission on internal hosts' ACLs, so as to send them probe packets and build up their profile database, and then they need to modify the NIDS' code to use this profile information gathered by them. This modification to the NIDS is rarely ever feasible in a real-world production network. Our normalizer has no such issues and can just be plugged into the network and will start functioning correctly, invisible to everyone.
3. Maintaining a profile database of the network topology and individual host behaviour is not only an impractical task (as mentioned in [2]), but also a security concern, since the Active Mapper getting compromised and this vast amount of information falling into the wrong hands could be dangerous for the site. Furthermore, the Active Mapper requires the NIDS to be configured to ignore traffic to and from the mapping machine. If an attacker gets control of the Active Mapper, these kind of special privileges can lead to him having free reign over the internal network, which is a possibly catastrophic circumstance.
4. The Active Mapper cannot solely depend on the profile built up,since they themselves have mentioned that in the case of internal machines' Network Stacks being modified from the stock version provided by default with the Operating System, their profile information will be inconsistent. They also cannot work correctly in the case of customized proprietary network stacks, which are becoming a very common occurrence in today's world of start-ups, all having developed their own Network Stacks or even specialized Kernels from scratch.

5. Ambiguous retransmissions cannot always be fixed by them, in which case they do need something like our normalizer in the picture to augment their functioning. We need no such augmentation and, as previously mentioned, are totally self-contained in our functioning.
6. Finally, and very importantly, they suffer from the same problems an NIDS does, due to their requirement of being located at a network point that is topologically equivalent to the link the NIDS is watching. This problem manifests itself if an attacker can get the Active Mapper to go down, because since the Mapper is fail-open, the network is still open to attack, whereas our normalizer is an active device in the path of all flows, so if it goes down, all access to the network from the external world is also restricted (fail-closed). This is a desirable property when security is more important to a site than accessibility.

One of the most common misconceptions in Network Security today is that a passive NIDS can effectively protect an organization from these attacks [12]. Unfortunately, this belief is misplaced for the reasons described in the previous section. As shown in paper [1], researchers already proved that passive NIDSs can only effectively identify malicious flows when used in conjunction with an active interposition mechanism. Through interposition, a Traffic Normalizer can guarantee protocol invariants that enable downstream intrusion detection systems to work with confidence. In the following sections, we will present our unique approach towards achieving this goal in an effective and computationally efficient manner.

4. Design

Our design was guided by, and was meant to conform to, the following principles:

4.1. *Design Goals*

1. **Generic and Reusable Design:** The key distinction of our work is a totally generic design whose policies are guided by an optimized state diagram, which, accompanied by a customized-mini stack, totally bypasses the in-built kernel stack.

2. **Security Without Compromising High Performance:** Regarding the security aspect, we guarantee to remove Insertion and Evasion attacks against an NIDS, which then helps the NIDS work with high assurance and confidence. To uphold the high performance of the network, we keep a minimum amount of state, performing only "logical reassembly" of incoming packets, with absolutely no movement of data, and leave all complex TCP processing to the end-hosts, while still being able to extrapolate the state of the TCP connections on both ends by examining the TCP headers of packets passing through us. This is explained in more detail in section 5.8 [High Performance].

3. **Robustness In The Face Of Attacks:** Our normalizer is an active element in the forwarding path of all traffic into and out of the site, so it plays an important role in the availability of the network. Therefore it was designed to not only survive, but also continue to function correctly, in the face of both direct (as mentioned in section 5.1 [MAC Transparency]) and indirect (as mentioned in section 5.6 [Self-Defense]) attacks.

4. **Independent and Self-Contained Implementation:** Eventually, our network stack, combined with a customized memory management module and slightly tweaked network device driver code, would be all the software our normalizer would need to work as a stand-alone hardware box of its own.

4.2. *Design Features:*

1. **Active interposition mechanism** → Is a “bump in the wire” [1], so all traffic must pass through it, hence it cannot be evaded. This also makes it fail-closed.
2. **Totally Transparent** → Even at the link layer level, thus requiring absolutely no re-configuration of any element of the network. This also makes it virtually impossible for attackers to direct attacks at our normalizer at the TCP,IP or MAC layers, as explained further in section 5.1 [MAC

Transparency].

3. **Customized mini-stack** → We totally bypass the kernel stack from just before the IP layer, using our specialized IP and TCP packet-handlers.
4. **Optimized State Diagram** → The backbone and guiding force for all of our normalizer's policies.
5. **Scalability** → Is half-duplex and implemented In-Kernel, keeps a small amount of state, performs minimal processing, and leaves all complex window management and all timers to the end-hosts.
6. **Reliable RST** → Innovative technique to ensure that end-hosts and NIDS maintain same state, especially in the case of connections being reset [2].
7. **Triage** → Robust mechanism for self-defense against Stateholding Attacks [2].
8. Provides a solution to the problem of the highly tricky and hard-to-detect IP Identifier and Stealth Port Scans [2].
9. Unique way of reliably instantiating valid state in the event of cold start.
10. At all points, we have ensured that TCP/IP semantics of packets sanctioned by us to enter the internal network, do not violate the RFC specifications in any way.

5. Implementation Details

The normalizer is implemented as a loadable kernel module for the Linux 2.4 kernel [13], which can easily be installed and removed using a single command. The normalizer module receives sk_buffs containing incoming IP packets as input from the first Netfilter hook. This ensures that we are receiving the packets from a safe point in the kernel [7]. Once we get this packet, we steal it and send it through our own customized mini-stack. From this point onwards, we do not use the kernel stack at all.

5.1. *Mac Transparency*

MAC transparency helps reduce the need for unnecessary tedious configurations by the network administrator and also ensures that users and attackers are unaware of our presence.

This was done by modifying the receive chain of packets in the Linux 2.4.20 kernel.When our normalizer module is not installed, this modification to the receive chain makes the machine behave like a ***logical bridge***, just blindly forwarding packets from the external to the internal network, and vice – versa.

We had already insured that the normalizer machine itself was absolutely inaccessible by any other machine on the network, by not giving it an IP address.

Thus, no packets can be directly addressed to the normalizer, and it will hence only process traffic at the IP and TCP layers for machines other than itself.

MAC transparency takes this self-defense against direct attacks one step further by making sure that any Ethernet frame whose protocol field does not specify IP, is sent untouched to the network on the other side of the normalizer, ensuring that we do not process anything at the MAC layer, and as a result, inoculating ourselves against any possible attacks at this level.

Thus, ARP poisoning and other such MAC layer attacks are impotent when attempted against our normalizer. Also, protocols that have a link-layer dependency or component, like ARP, RARP and DHCP can work totally unbroken and unchanged in the presence of our normalizer.

The normalizer consists of two main modules:

1. IP level normalizations module
2. TCP level normalizations module

5.2. *IP level normalizations module*

This module gets input from the first Netfilter hook as mentioned above. We begin by walking through the IP header [3] of these packets, normalizing fields which may cause ambiguities. If the packet is fragmented, we ***logically reassemble*** this packet from its fragments, hashing each packet to its proper slot in a hash queue. Next, if the amount of memory consumed by fragment reassembly has surpassed a configurable upper limit, we perform IP triage to defend ourselves against a Stateholding attack. Once we get a complete IP packet, we check the protocol in its header, and in case it is TCP, we pass it to the next module.

5.3. *TCP level normalizer module*

This module receives complete IP packets, either as a single sk_buff (an unfragmented IP packet) or as a list of sk_buffs containing fragmented IP packets. Initially, we check whether this packet's header is malformed or contains an incorrect checksum, in which cases, we drop it. Also, we ensure that all incoming SYN or RST TCP segments enter the internal network with no payload whatsoever.

5.4. *Safe and Consistent Retransmissions*

By performing full TCP and IP reassembly in a light manner, and thus enforcing a single interpretation of each data stream, our normalizer guarantees that its

output to the internal network is totally contiguous and free of holes, and in the case of retransmissions, it assures that the data it resends, is an exact copy of what it had sent out previously, leaving absolutely no scope for ambiguous retransmissions.

5.5. *Cold Start Problem*

In order to ascertain that we can be inserted seamlessly into any existing network, we have tackled the Cold Start problem ingeniously. This is done by following these 2 simple rules:

1. If a packet on an unknown connection seen after cold start is from the internal network, instantiate state, since the internal network is trusted.
2. If a packet on an unknown connection seen after cold start is from the untrusted external network, transform the packet into a keep-alive ACK packet and forward it inside. If there actually was a connection before cold start, the internal host will reply to this keep-alive, and hence state will reliably be instantiated, according to the previous rule.

5.6. *Effective Self-Defense*

One more handy feature of our normalizer is the fact that it can defend itself effectively against indirect DOS and Stateholding attacks, by following a policy of triage. According to this policy, we monitor the amount of in-use memory for TCP connections currently being tracked, and if this memory crosses a configurable upper threshold, we discard state for connections that haven't shown activity in a long time.

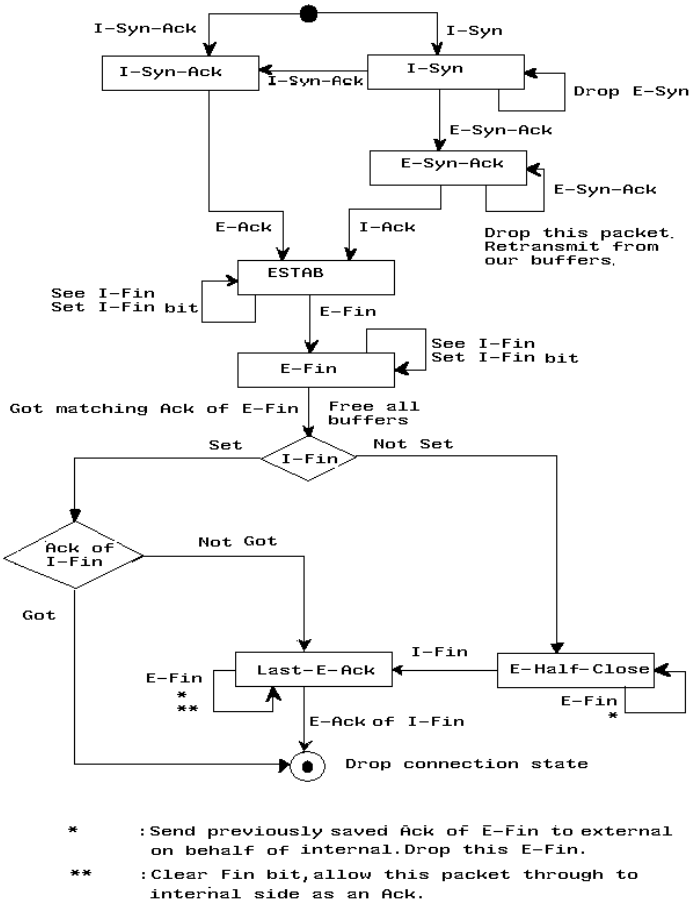
The information about relative activity of connections is maintained in a simple and efficient manner, by hashing all our connection states to a hash queue, and on every access of a particular connection, moving that connection's state to the front of its queue. Thus, triage victims are selected in a Least Recently Used fashion by just deleting the state of the connection at the end of each queue for each hash slot.

Also, our state diagram dictates that all state is to be instantiated based on the internal machines' reactions only, so as to prevent the normalizer falling prey to a SYN flooding attack.

5.7. *State Diagram*

The major part of our design phase went in creating our optimized and reduced state diagram, which tries to take care of all possibilities of retransmissions in

different situations, and makes sure traffic flowing into the network is totally innocuous. We have also dealt with small but important details, like half-close, simultaneous-open and simultaneous-close efficiently and simply. Further, the stringent rules enforced by our state diagram block many stealthy and dangerous pre-attack scans, the most notable ones being *nmap*'s stealth NULL scan, stealth XMAS scan and stealth FIN scan [9].



5.8. *High Performance*

Because of being an active part of all flows, our basic design goal had to be to uphold the high performance of the network, which we have achieved by:

1. Holding a minimal amount of state. Whereas Linux's TCP data structure occupies 424 bytes (*struct tcp_opt*), we achieve our goal by holding ONLY 60 bytes (*struct tnt_tcp_conn*) of state per connection.
   That is 7 times less state than the Linux Kernel!!!
2. Performing the basic minimum amount of required processing to function correctly, by leaving all complex window management and timer processing to the end-hosts.
3. Doing all our reassembly, at both the IP and TCP layers, ***logically***. We don't strip and prepend headers, or defragment and refragment IP packets, or physically reassemble and compress fragments into 1 big sk_buff, like the kernel does. But we achieve the same effect as kernel reassembly, without the added overhead.
4. Normalizing only incoming traffic, making our processing ***half-duplex***.
5. An ***in-kernel implementation***, thus reducing unnecessary copying of data between application and kernel-space.

These facts should help inspire confidence that the scalability of our normalizer is not an issue.

6. Testing

6.1. *The "root" of all NIDS problems*

As [2] and [4] have stated, when an end-system receives *overlapping* IP fragments that differ in the purported data for the overlapping region, some endsystems may favor the data first received, others the portion of the overlapping fragment present in the lower fragment, others the portion in the upper fragment.

Operating System	Overlap Behavior
Windows NT 4.0	Always Favors Old Data
4.4BSD	Favors New Data for Forward Overlap
Linux	Favors New Data for Forward Overlap
Solaris 2.6	Always Favors Old Data
HP-UX 9.01	Favors New Data for Forward Overlap
Irix 5.3	Favors New Data for Forward Overlap

The table above is a summary of the research done in [4] to point out IP fragment overlap behaviour for various OSs, and has been used here verbatim to further explain the problem of ambiguous retransmissions.

This leads to a problem that is not easily reliably solvable by an NIDS, because of the exponential explosion of possibilities for it to analyze. As a result, this translates into a very easily generatable cause of failure of an NIDS, thus rendering it ineffective.

(All packets in the test below were generated with the help of an open-source, free packet-generator called packETH [17], which was also used for the majority of our basic-correctness testing, with the help of which we were able to test all flow paths through our customized state-diagram (section 5.7) and verify that things worked as expected.)

To demonstrate the problem of ambiguous retransmissions and to test that our normalizer correctly solved this problem, we hijacked an existing telnet session and sent a hole that looked like this….

_ _ _ _ R O O T

Then, we retransmitted this overlapping segment….

_ S E R G R U B

and finally patched up the hole by sending

U

Without our normalizer installed
Windows reassembled **U S E R R O O T.**
Linux reassembled **U S E R G R U B.**

With our normalizer installed
Windows and ***Linux*** reassembled **U S E R G R U B.**

This was a very important proof-of-concept test for us, since every ambiguous reassembly attack that an NIDS is susceptible to, can be reduced to this simple test, thus proving the usefulness of and need for our normalizer.

6.2. *Nmap Tests*

To test the correctness of our state-diagram (section 5.7), we used the full gamut of tests provided by nmap, and were successfully able to block all their stealthy and dangerous pre-attack scans, the most notable ones being their stealth NULL-scan, stealth FIN-scan, and stealth XMAS-scan.

As concrete proof of our generic and complete approach to normalizing traffic, we were also able to defeat nmap's most diligent OS fingerprinting scan, without making any extra efforts, or intending to do so. We just ran their OS fingerprinting scan through our normalizer, and all the normalizations we had already performed took care of blocking it automatically, without us having to make a single change anywhere.

6.3. *Fragroute Tests*

Fragroute by Dug Song [10] is a tool that "intercepts, modifies, and rewrites egress traffic destined for a specified host, implementing most of the attacks described in the paper "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection" [4].

This tool has been able to break several commercial NIDSs deployed in the real-world, by providing implementations for all the different categories of attacks made famous in Ptacek and Newsham's seminal paper on the shortcomings of IDSs and the ways in which they could be exploited, and was therefore invaluable to us and formed the backbone of our testing methodology. Other related works like [2] and [16] have also used this as one of the primary tools in their test setup, thus establishing fragroute as the common benchmark for any work in this field to be validated against.

Our test setup consisted of 2 machines representing the external network, 5 machines representing the internal network, and the our normalizer residing between these 2 networks.

We installed fragroute on the external machines, which along with valid traffic, would also inject bogus and malicious packets into the internal network, according to the rules specified by us. Thus, we simulated a multitude of known reasons-of-failure for an NIDS, and were successfully able to defeat and defend against all of the possible attacks that can be generated by fragroute's ruleset (including combinations of ambiguous and overlapping retransmissions, insertion and evasion attacks), which is, in itself, no small feat.

6.4. *Benchmark Tests*

6.4.1. *Internet Downloads*

We connected to the internet on a 128 kbps broadband connection, through the

normalizer machine, and ran some basic download tests with and without our normalizer module installed. The timings are as shown in the table below.

Size (in MB)	Time Without Our Normalizer	Time With Our Normalizer
2.55	7:42	7:57
3.71	12:05	12:18
3.77	12:10	12:21
6.33	18:46	18:53
10.86	25:12	25:22

6.4.2. *Real-time Performance Testing*

To back our performance claims, we also listened to Yahoo!'s real-time internet streaming radio, Launchcast with and without our normalizer module installed and the results are shown below.

TB: Total Buffering

Song Duration (in minutes)	TB Without Our Normalizer (in seconds)	TB With Our Normalizer (in seconds)
3: 02	5	7
4:29	8	9
5:37	9	10
7:41	10	10
16:23	12	7

7. Future Work and Enhancements

Our first enhancement is to be ***totally independent*** of the kernel stack. To do this, we need to design our own memory management module. In the case of Linux, this would imply having to write our own version of an sk_buff, because we found that Linux's sk_buff [5] contained too many fields that were of no use to us, so that is a totally unnecessary overhead. Also, we would have to provide a set of our own access functions for our new data structure.

Due to our dependence on the native Linux network device driver code [8], this version of our normalizer sometimes shows limited performance. Hence, we would like to modify our NIC device drivers [11] to customize the reception and transmission queuing policies to achieve better performance in these pathological cases.

Once we fulfill these 2 goals, our normalizer would then *truly* be working with

our ***own full-fledged customized network stack***.

The implementation introduced in this paper is basically the framework for an industry-level product, which will deliver much higher performance and will be capable of surviving in environments of 1Gbps and beyond. To turn this dream into a reality, we realize that nothing short of hardware implementation will do. Right from the design phase, we have made a conscious and deliberate effort to minimize the amount of external interference and configuration needed for our normalizer to run. This was done keeping in mind our ultimate end-goal of being a transparent Plug-n-Play device on the wire.

8. Conclusion

An NIDS is like a network sensor, which is used to detect various attacks or patterns of misuse. If this sensor can be fooled, many attacks may go unmonitored and unnoticed. Since our normalizer guarantees the proper functioning of the NIDS, this is a huge threat we mitigate. Furthermore, the NIDS log is reviewed by the network administrator for forensic analysis, to try and see what went wrong. Our normalizer will help by showing very clearly, exactly what happened, by removing all ambiguous or confusing traffic. Our normalizer fixes an inherent problem in all NIDSs, which is to log many false positives (false alarms) and false negatives (failure to alert). In spite of being an active part of all flows, the normalizer manages to uphold the high performance of the network, which is a huge point in its favor. Finally, due to the increasing inability of NIDSs to faithfully detect various new attacks, companies have to reinvest and spend more money by shifting to Host-based IDSs, which are more expensive and difficult to maintain. Thus, our normalizer can ease this financial burden on companies, by making what they already have invested in (i.e. NIDSs), work with high assurance and correctness.

References

1. David Watson, Matthew Smart, G. Robert Malan, and Farnam Jahanian, “Protocol Scrubbing: Network Security through Transparent Flow Modification, IEEE/ACM Transactions on Networking, Vol. 12, No. 2, April 2004.
2. M. Handley, C. Kreibich, and V. Paxson, “Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics,” in Proc. 10th USENIX Security Symp., Washington, DC, Aug. 2001.
3. W. R. Stevens, TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, 1994.
4. T. H. Ptacek and T. N. Newsham, “Insertion, evasion, and denial of service: Eluding network intrusion detection,” Secure Networks, Inc., Tech. Rep.,

Jan. 1998.
5. Thomas F. Herbert, The Linux TCP/IP Stack: Networking for Embedded Systems. Charles River Media, 1st edition, May 2004.
6. Elizabeth D. Zwicky, Simon Cooper & D. Brent Chapman, Building Internet Firewalls, O'Reilly Media, Inc.; 2nd edition, January 15, 2000.
7. Guo Chuanxiong, and Zheng Shaoren, Analysis and Evaluation of the TCP/IP Protocol Stack of LINUX, [Online]. Available: http://edu.jsvnet.com/~xguo/IP_Stack_Analysis_2000.pdf
8. Robert Love, Linux Kernel Development, Novell Press; 2 edition, January 12, 2005
9. Fyodor, *nmap*, 2001. [Online]. Available: http://www.insecure.org/nmap/
10. Fragroute, D. Song. [Online]. Available: http://www.monkey.org/~dugsong/fragroute/
11. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers, O'Reilly Media, Inc.; 3 edition, February 7, 2005.
12. Sarah Sorensen, Intrusion Detection and Prevention: Protecting Your Network from Attacks. [Online]. Available:www.juniper.net/solutions/literature/white_papers/wp_idp.pdf
13. Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, O'Reilly Media, Inc.; 2 edition, December 2002.
14. Douglas E. Comer and David L. Stevens, Internetworking with TCP/IP (volume 1), Prentice Hall; 3rd edition, June 15, 1998.
15. Andrew S. Tanenbaum, Computer Networks, Prentice Hall PTR; 4 edition, August 9, 2002.
16. Umesh Shankar & Vern Paxson, “Active mapping: Resisting NIDS evasion without altering traffic”, Security and Privacy, 2003.
17. Packeth – Ethernet packet generator [online]. Available: http://packeth.sourceforge.net/