To appear in the proceedings of the 38th Hawaii International Conference on System Sciences (HICSS-38). January, 2005.

# Elephant: Network Intrusion Detection Systems that Don't Forget

Michael G. Merideth and Priya Narasimhan
*School of Computer Science*
*Carnegie Mellon University*
*5000 Forbes Ave, Pittsburgh, PA, USA*
*{mgm, priya}@cs.cmu.edu*

## Abstract

*Modern Network Intrusion Detection Systems (NIDSs) maintain state that helps them accurately detect attacks. Because most NIDSs are signature-based, it is critical to update their rule-sets frequently; unfortunately, doing so can result in downtime that causes state to be lost, leading to vulnerabilities of attack misclassification. In this paper, we show that such vulnerabilities do exist and provide a way to avoid them. Using the open-source NIDS Snort, we present* Elephant*, an approach and implementation for updating rule-sets that provides a way to cause Snort to enter a safe quiescent point, load the new rules into memory, and remove the old rules from memory—all while preserving the state that is required to make sure that the NIDS does not miss attacks. We provide a critique and performance evaluation of our technique.*

## 1. Introduction

Network Intrusion Detection Systems (NIDSs) [4] provide the critical security function of detecting attacks and compromises on a network. As NIDS technology improves, the reliance on NIDSs will likely increase. Most commercial NIDSs are signature-based, so they require a database of attack signatures (for detecting network-based attacks) and a corresponding rule database (for classifying and taking action on any malicious network traffic). Because new attacks emerge rapidly, these databases must be updated frequently with signatures and rules for the new attacks.

Many modern NIDSs track various types of connection-state, as discussed in Section 3, in addition to performing stateless packet analysis. This state is often associated with various protocols (such as TCP, HTTP, and FTP) and helps the NIDS avoid the possibility of misclassifying packets that are sent using these protocols. This state tracking helps defend against a class of attacks that specifically targets the inherent lack of "memory" of stateless NIDSs.

The need to update the rule and signature databases conflicts with the desire to have the NIDS running all of the time on the network. Any scheduled downtime incurred in updating the rules reduces the NIDS's online availability. As we discuss in Section 10.1, simply replicating the NIDS does not necessarily reconcile this conflict or solve this problem.

NIDSs are often deployed such that they passively monitor network traffic on a network segment. Passive monitoring means that the NIDS might analyze the traffic, but does not attempt to modify or filter it. This passive monitoring has the advantage that it is minimally intrusive because it does not affect the flow of network traffic. However, this also means that NIDSs *fail-open*: attacks can occur *even if the NIDS is down*. If a NIDS crashes or is down for updates, the network will continue to function without the monitoring and protection provided by the NIDS.

The current state of the practice for updating the signatures and rules in the popular open-source NIDS known as Snort [11] (discussed further in Section 4), is to download or create rule and signature updates, merge them into the current databases using a rule-management tool like Oinkmaster [6], and *restart* the NIDS in order for the updates to take effect.

The problem with this update strategy is that it does not account for the state tracking that is performed by the NIDS. The NIDS might be optimized to restart very quickly, so a restart might add only a negligible amount of downtime in terms of number of seconds. However, if the state that the NIDS maintains is not retained over a restart, the NIDS may misclassify or entirely ignore important attack traffic once it is restarted.

In this paper, we demonstrate a timing vulnerability introduced whenever the rules of the NIDS are updated in a manner that does not preserve the state of the NIDS. We present *Elephant Reload* as a way to prevent these attacks through no-restart, state-preserving NIDS rule updating. We present an implementation for Snort called *Elephant STR,* an empirical evaluation, and a critical discussion of the related issues.

The organization of this paper is as follows. We begin by discussing our assumptions. Section 3 discusses the reasons that modern NIDSs maintain state. In Section 4, we present the vulnerability that is created by using restart as a mechanism for rule updating. Section 5 presents the current techniques for reloading Snort's rules, and Section 6 presents our alternative model and algorithm for online rule updating. We present an implementation for our model and a performance evaluation of it in Sections 7 and 8. Section 9 presents potential enhancements to our implementation, while Section 10 considers insights and lessons learned from our experience. Section 11 is devoted to related work, and Section 12 concludes the paper.

## 2. Assumptions

When we discuss the availability of a NIDS, we are concerned with online availability. This means that we are concerned with cases in which the desired behavior of the NIDS is real-time intrusion-detection performed on live network traffic. In certain situations, NIDSs may be used to analyze network-traffic traces that were captured to a file; we consider this to be offline processing.

Our strategy of performing online rule and signature updates does not mean that the updates must be performed automatically—it means simply that the NIDS does not need to be restarted to take advantage of the updates.

We assume that the rule updates are valid. Current best practices for updating the rules and signatures dictate that the updated databases should be tested (statically or dynamically) before use in a production system [2]. Before updating the NIDS using Elephant Reload, this testing can still be performed.

In this paper, we focus on only one particular source of potential downtime: the need to update the rule and attack signatures to account for new attacks. However, there are many sources of potential scheduled and accidental downtime for deployed NIDSs. For example, in certain cases, the NIDS may crash, the machine it runs on may crash, its network connection may fail, or a new version of the NIDS might need to replace the current version.

Our experiments and implementation are based on the popular open-source network intrusion detection system, Snort [11]. Where we can generalize our findings and experience to another NIDS like Bro [7], or to NIDSs as a class of systems, we do so through the use of the term *NIDS*; where our experience relates directly to Snort, we use the word *Snort* in place of *NIDS.*

## 3. The State Maintained by Modern NIDSs

The Snot [14] and Stick [3] attacks were developed to overwhelm stateless NIDSs. These attacks function by generating packets that are expected to trigger alerts by the NIDS. These attacks seek to cause so many alerts that valid attack traffic will be able to slip by unnoticed in the noise; this heavy volume of alerting can even cause a NIDS to crash under the load of analyzing each packet and producing the corresponding alerts [5].

To create a high volume of misleading traffic, Snot and Stick generate and send packets with no regard to TCP connection state. Skipping the TCP connection establishment step helps this type of attack for the following reasons: (1) it saves on the time-overhead involved in creating a connection, (2) it does not require the existence of a TCP server to which to connect, (3) the attacker can spoof the source IP address because no response is expected, and (4) the attacker can continue to send bad traffic without concern that the connection might be closed or reset by the server.

Modern NIDSs maintain state in order to thwart attacks like Snot and Stick. This state allows a NIDS to discard the types of attack packets generated by Snot or Stick preemptively without actually examining the contents of the packets. In particular, stateful NIDSs typically assume that it is safe to discard packets that are not sent over an established TCP connection.

Working from the assumption that it is reasonable to discard TCP packets that are not part of an established TCP connection, a stateful NIDS attempts to track the connection state of each of the end-hosts on its network segment. The NIDS watches for occurrences of the TCP three-way handshake, which indicates that a TCP connection was successfully established between a client machine and a server machine (one or the other of which is on the NIDS's network). The NIDS simply drops any TCP traffic that does not occur over an established TCP connection that it knows about. For accurate attack detection, it is important that the connection state on the NIDS stays in sync with the actual TCP connections that it watches.

By automatically dropping traffic that is not sent over a legitimate TCP connection, the NIDS avoids the problem of generating alerts for pseudo-attack traffic that will have no effect on end-hosts. The assumption that this behavior is safe is based on the fact that, in general, the TCP protocol stack of the target server will also discard these packets; therefore, this traffic will not be able to exploit any potential vulnerability in the server. Additionally, the traffic might not be handled by any target server at all—the Snot attack tool can be configured to send TCP packets to random host addresses within a network without regard as to whether the target hosts exist.

## 4. Vulnerability in Snort

Snort uses a single set of rules that acts as both the signature database and the rule database. Snort was introduced in 1998, and originally provided only stateless intrusion detection. TCP session-tracking was added to

Snort in 2001 with the introduction of the *stream4* preprocessor. According to the release documentation, *stream4* is capable of stateful inspection for upwards of 32,000 simultaneous TCP connections [13].

All active Snort preprocessors have the opportunity to act on packets before the packets are sent to the stateless detection engine. These preprocessors maintain individual state databases for a variety of purposes such as packet-fragment reassembly, TCP session inspection, TCP session reassembly, and application protocol tracking. The current version of Snort (2.1.2) ships with 12 preprocessors (based on an examination of the preprocessor source code directory). Of these, 5 preprocessors (*frag2, stream4, http_inspect, rpc_decode,* and *bo*) are enabled by default. These preprocessors can be quite complex; for example, the *stream4* preprocessor implementation file contains 4980 lines of commented *C* code.

Snort contains a vulnerability based on the interaction between its TCP connection-tracking and its rule updating that can be exploited when Snort is restarted after a rule update. Despite the relatively complex state tracking performed by the preprocessors, the support for updating Snort's rule-set is very minimalist. As discussed in Section 5, the current procedure for updating Snort's rule-set is to restart Snort after downloading the new rules. The problem is that Snort loses its state during the restart.

When configured to defend against the Snot and Stick attacks, Snort uses the *stream4* preprocessor to drop packets that appear to have been sent outside of a valid TCP connection. When restarted, Snort loses the state maintained by its preprocessors, including the state about the TCP connections that are active on its network. Because of this, Snort will simply discard any packets—even actual attack packets—that are sent over TCP connections that were established before it was restarted, even if it knew about the TCP connections before it was restarted.

## 4.1 Demonstration of the Vulnerability

We now show that this vulnerability does exist in Snort. In order to demonstrate the vulnerability, we first require a tool that sends attack traffic over established TCP connections. The previously mentioned Snot tool reads publicly available Snort rule files and generates packets that specifically target those rules in order to trigger alerts. However, in its present form, Snot does not attempt to establish a TCP connection before sending TCP packets.

We use a version of the Snot 0.92a tool, which we have modified in a manner that we describe next; we refer to our modified version of Snot as TCPSnot. Unlike Snot, our TCPSnot tool *does* establish a TCP connection before sending attack traffic so that, as discussed in Section 3, the traffic that it generates should not be discarded by a NIDS that is configured properly for TCP state tracking. TCPSnot uses the stream socket interface instead of the raw socket interface that is employed by the standard Snot tool. The use of the stream socket interface makes TCPSnot somewhat less flexible than Snot in that TCPSnot cannot forge or modify parts of the TCP or IP headers. Creating a more flexible version of TCPSnot could be achieved by lower-level modifications to the kernel.

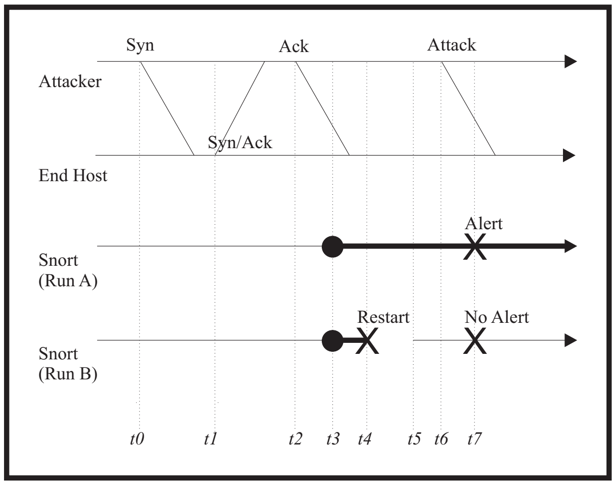

**Figure 1. Two trial runs that together illustrate the vulnerability in Snort using TCPSnot.**

To demonstrate the vulnerability, we perform an experiment consisting of two trial runs. In both runs, we use identical Snort configurations that track TCP connection state. With Snort already running, we have TCPSnot establish a connection to a server on the Snort server's network, pause for a fixed length of time (at least the length of time required to reboot Snort), and then send an attack packet that should trigger an alert by Snort, based on its current rule-set.

The two runs are depicted in Figure 1. During both runs, TCPSnot establishes a TCP connection to the end host (*t0-t2*)*,* a fact that Snort records at *t3*. In Run B, Snort is restarted at *t4* and finishes restarting at *t5*. At *t6,* the attacker sends an attack packet. In Run A, Snort correctly recognizes the attack as occurring over a TCP connection, so it generates an alert at *t7*. But in Run B, Snort fails to do so because it has lost its state; Snort will also ignore any subsequent attack packets sent over this TCP connection.

The two runs differ as follows:

1. In Run A, we do not restart Snort. As expected, Snort detects the attack packet and generates the expected alert.

2. In Run B, we restart Snort (using the Snort Reload technique that is discussed in Section 5) after TCPSnot has established a connection to the server. Once Snort has finished restarting, TCPSnot sends the attack packet. As predicted, Snort *fails* to generate an alert for this packet.

The cause for this misleading behavior is clear: Snort has lost its state after its restart.

## 5. Current Snort Rule Updating

In order to update its rules, Snort is typically restarted using one of two techniques, which we refer to as *Snort Restart* and *Snort Reload*. Both techniques result in a loss of state that can lead to the vulnerability discussed in Section 4.

Snort Restart is characterized by a manual two-step approach. The current Snort process is manually shut down, a new Snort process is then manually started. These two steps require operating-system-provided facilities, such as a way to terminate the current Snort process and a way to launch a new Snort process.

Snort Reload is similar to Snort Restart, but it is done in memory, automatically by Snort, and uses only a single process. Under Snort Reload, the Snort process first calls the Snort shutdown routines, but does not terminate. Instead, once the shutdown routines have completed, the process uses the POSIX *execv()* call (or *execvp()*, depending on configuration options) to replace the current Snort process image with a newly started, clean Snort image.

To perform a Snort Reload, the administrator sends a SIGHUP signal to the Snort process; Snort treats the SIGHUP signal as an indication that it should restart itself. Snort Reload does not work in its current state if Snort is run in a *chroot jail* or if it is set to run without the privileges of the super user.

## 6. Elephant Online Rule Updating

From an architectural standpoint, as discussed in Section 4, Snort is designed so that its intrusion detection engine (the place where the rules and attack signatures are used) is stateless, while its preprocessors may maintain state. It is this characteristic that can make online rule updating relatively straightforward. We are only updating rules—we are *not* updating the runtime state, the code that acts on that state, nor the interfaces to that code.

Our basic algorithm, which we call *Elephant Reload*, for updating the rules, is as follows: first, find a quiescent point, *i.e.,* a point where the rules are not in use; next, load the new rules into memory; and, finally, replace the current rules with the new rules.

From an application consistency standpoint, it is important that the rule update be applied at a quiescent point in order to avoid the possibility of packets being checked partially using the old rule-set and partially using the new rule-set. From an implementation standpoint, applying the rule update at a quiescent point makes it much simpler to avoid stale runtime references to old rules that could be cached while the rules are in use.

## 7. Elephant STR Implementation

In order to implement Elephant Reload for Snort, we need the following: (1) a way for the administrator to signal that the rules should be updated, (2) a quiescent point, (3) a way to load the rules into memory, and (4) a way to remove the old rules from memory. As we will see in this section, Snort provides us (1–3) with relative ease, but (4) poses more challenges.

As seen in Figure 2, the modified running Snort system can be in one of three states: the NORMAL_STATE, in which Snort executes as it did without modification; the INTERRUPT_STATE, which is the direct consequence of a request for rule reloading and which flags the need for a rule update; or the QUIESCENT_STATE, where the rule-set is not being accessed and where the rule updates can therefore safely occur.

We have created an implementation of Elephant Reload for Snort 2.1.2 that we call *Elephant STR* (single-threaded reload). Snort is single-threaded by design; the Snort developers have stated that this is, in part, because of a desire to retain platform portability [9]. Therefore, Elephant STR does not rely on the addition of any new threads (hence the STR). In Section 9, we propose more complex implementation variants of Elephant Reload, such as one that uses multiple threads, and another that does an in-memory merge of the current and new rules in order to reduce memory overhead. Here, we discuss only the Elephant STR implementation.

### 7.1 Requesting the Reload

We require a way to signal that Snort should reload its rule-set. It would be possible to implement this in a number of ways, including having Snort listen for special network traffic, having Snort listen on a socket, or by using the UNIX Signal API.

Of these options, we feel that the UNIX Signal API [15] provides the most direct and natural interface for sending “commands” to Snort, when given a finite, small set of commands. An added benefit is that this behavior fits naturally with Snort—indeed, Snort already handles a number of signals; for example, Snort currently catches SIGUSR1 for displaying statistics about packet processing.

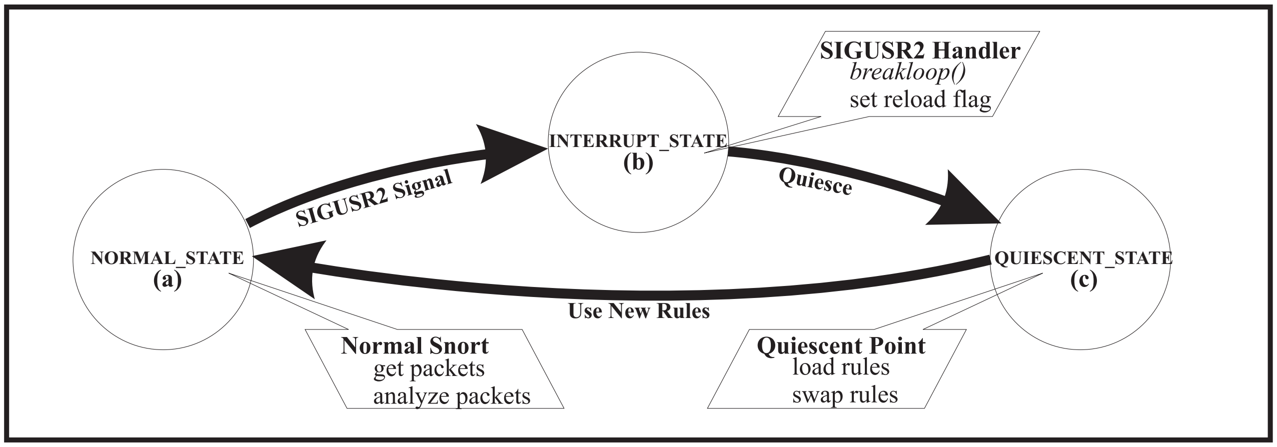

**Figure 2. Elephant Reload Implementation (Elephant STR).**

The SIGUSR2 signal is reserved for application-designated purposes, although it is only rarely used by applications. Using the UNIX Signal API, we modify Snort by adding a SIGUSR2 handler that treats the SIGUSR2 signal as a request to reload the rule-set. We adhere to Snort's source-code conventions of allowing nested signals and disallowing recursive signal handlers (*i.e.,* those caused by the same signal).

For Elephant STR, we cannot use this new SIGUSR2 handler as the point at which actually to reload the rules because Snort may not be quiescent when the handler is invoked. UNIX signal handlers asynchronously interrupt the executing application code, so it is possible that the SIGUSR2 handler would interrupt the detection processing in which the rule-set is being used. Even if we did not care about this level of consistency (in which not even a single packet can be partially processed by both rule-sets), we still could not use the signal handler as the quiescent point to load new rules, because, under UNIX, it is not safe to allocate heap memory in a signal handler [15].

For these reasons, we use the SIGUSR2 signal handler as the INTERRUPT_STATE, which asynchronously indicates that we would like the rule-set to be reloaded, but we defer the rule updating to a truly quiescent point.

## 7.2 Quiescent Point

The point at which we reload the rules must be a quiescent point in which the rule-set is not being used. Finding a quiescent point may seem complex, but our implementation makes it straightforward. We take advantage of the fact that Snort is single-threaded in order to know that Snort is not using the rule-set at the point where we perform our rule updates.

For our purposes, the point at which Snort is about to check for the arrival of a new network packet is a quiescent point. Snort spends the majority of its time, after initialization, in a loop reading packets, as they arrive on the network interface, and processing these packets, one at a time, in the Snort preprocessors and detection engine. Because Snort is single threaded, while it is waiting in this loop for a packet to arrive, it is necessarily not processing any packets.

In order to take advantage of this quiescent point, we must be able to insert our own code for the implementation of the QUIESCENT_STATE, where the rule updating can be safely performed. We do this by programming Snort to exit from the packet-reading loop when a rule update request has been signaled (using the mechanism described in Section 7.1). We then have Snort enter the QUIESCENT_STATE, immediately following its exit from the loop, to perform the rule update. Because the loop may exit for a variety of reasons (including errors and application termination), we have Snort check that the exit from the loop was the result of a request for a rule update.

The standard implementation of Snort's packet-reading loop uses the *libpcap* [16] packet capture library, specifically the *pcap_loop()* function. This function reads from a network interface and sends the packets that it reads to a pre-specified handler function; in this case, the handler function starts the Snort packet analysis. In addition to watching for packets on the network interface, *pcap_loop()* also checks a flag; if this flag is set, then *pcap_loop()* exits. The flag is set by calling the function *pcap_breakloop()*.

The flow of actions required to reach the QUIESCENT_STATE is as follows. The administrator sends a SIGUSR2 signal to Snort. This causes Snort to enter the INTERRUPT_STATE, call *pcap_breakloop(),*

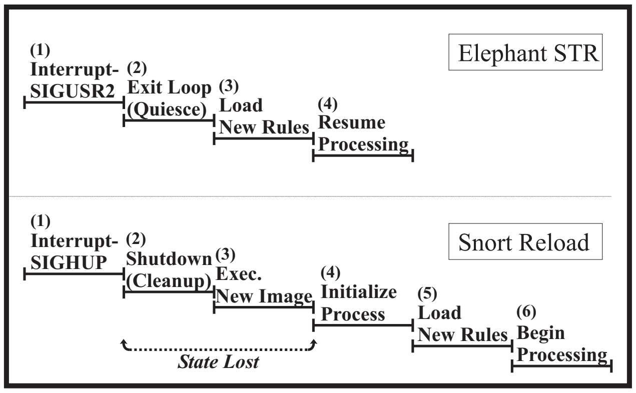

**Figure 3. Steps required for Elephant STR *vs.* Snort Reload (not to time-scale).**

and set a reload flag indicating that a rule update has been requested by the administrator. On the basis of the loop interrupt and the reload flag, Snort enters the QUIESCENT_STATE and starts the reloading process before retrieving another packet for processing. Either before or after the rules are reloaded, the memory for the current rule-set can be de-allocated.

### 7.3 Loading the New Rules

In the QUIESCENT_STATE, we piece together parts of the built-in initialization code from Snort in order to load the new rules into memory after first deleting references to the old rules.

We must modify the default rule-parsing functionality somewhat in order to disable certain types of actions that can be triggered by parsing the rule file. In particular, given Snort's current architecture, certain actions, such as initializing the preprocessors, can be done only once per run; however, the rule-file may contain directives that specify that preprocessors should be loaded. We modify Snort's parser to ignore these directives during a rule update in a way that we believe does not affect the loading of the rules.

### 7.4 Removing the Old Rules

Unfortunately, Snort is not designed to de-allocate the memory for its current rule-set. This task is left for the operating system when the Snort process terminates. This, coupled with the fact that there are no memory de-allocation routines pre-coded into Snort, makes de-allocating the memory a somewhat involved task as the rules are stored in memory in multiple chains (for faster indexing). For our purposes in this paper, memory de-allocation is relatively unimportant, so our code does not currently de-allocate the memory for the old rule-set.

### 7.5 Putting it All Together

Figure 3 contrasts the steps required for Elephant STR with the steps required for Snort Reload. Elephant STR consists of four high-level steps: (1) the user signals a rule-reload request by sending a SIGUSR2 signal to Snort; (2) this causes the packet-reading loop to exit (after waiting to finish processing the current packet, if there is one); (3) Elephant STR then loads the new rules; and (4) resumes packet processing.

Snort Reload, on the other hand, requires six high-level steps: (1) the user signals a rule-reload request by sending a SIGHUP signal to Snort; (2) this causes Snort to execute its shutdown routines; (3) once the shutdown routines are complete, Snort calls *execv()* to overwrite its current process image with a new one; (4) Snort initializes the new process image (as it would on initial startup); (5) Snort then loads the new rules; and (6) begins packet processing. The state of the Snort process is lost during steps (2) and (3).

## 8. Performance Evaluation

Although our primary goal in implementing Elephant Reload is to avoid the vulnerability described in Section 4, it is also the case that, as seen in this section, our Elephant STR implementation can result in improved performance in terms of the time required for rule reloading.

In this section, we compare Elephant STR to the Snort Reload technique introduced in Section 5. We do not evaluate the Snort Restart technique introduced in that section because Snort Restart requires manual intervention by either the administrator or a script running on behalf of the administrator. The way in which this intervention is done would vary from site to site. Given the intervention, we would expect Snort Restart to perform like Snort Reload in the best case, and to perform worse than Snort Reload in the common case.

We also do not evaluate the technique that we discuss in Section 10.1, which involves starting a second Snort process before shutting down the first process. This technique should lead to different performance impacts; in particular, we would expect this technique to require additional memory overhead, but result in less downtime. This technique will not preserve the state of the original Snort process.

### 8.1 Hardware and Software Configuration

Our performance evaluation configuration consists of a single server with 512M of RAM, based on an Intel Pentium 4 CPU running at 2.4Ghz. This server runs Redhat Linux 9.

**Table 1. Performance of Elephant STR and Snort Reload for the default Snort rule-set.**

| | MEAN | MIN | MAX |
|---|---|---|---|
| **(A) Elephant STR** | 203,767 $\mu s$ | 202,216 $\mu s$ | 206,147 $\mu s$ |
| **(B) Snort Reload** | 259,921 $\mu s$ | 256,824 $\mu s$ | 264,839 $\mu s$ |
| (B1) Shutdown | 0.09% | 0.10% | 0.08% |
| (B2) Child Proc. Init. | 95.24% | 95.07% | 94.10% |
| (B3) Misc. Overhead | 4.66% | 4.83% | 5.82% |
| **(C) Improvement** | **21.60%** | **21.26%** | **22.16%** |

We start Snort with super-user privileges because Snort requires these privileges initially to access the network interface in promiscuous mode under Linux.

Our experiments consist of multiple runs so that we can measure average case behavior; for each experiment, we stop the Snort process used in the current run (after loading the new rules and taking performance measurements) and then start a new copy of Snort for the next run.

The rule loading time for any of the evaluated techniques includes time to print processing statistics to the terminal, a feature that is enabled in Snort by default.

## 8.2 Analysis with the Default Rule-Set

Table 1 compares our Elephant STR technique with Snort Reload, and shows the times required for various components of Snort rule loading, based on our measurements over seven runs. This data represents the times to reload Snort's default rule configuration (as specified by the *snort.conf* configuration file) that ships with Snort 2.1.2. This rule configuration loads 1679 rules.

Snort Reload (B) requires a total of 259,921 microseconds on average. This time includes the time for Snort to perform its shutdown routines, the time for it to reinitialize and reload the rules, and some miscellaneous overhead time, including the time for the *execv()* call itself.

The shutdown routines (B1) require 0.09% (244 microseconds) of Snort Reload on average. Initializing Snort in the default configuration again (B2) requires 95.24% (247,560 microseconds) of Snort Reload on average. The miscellaneous overhead (B3) consumes the remaining 4.66% (12,118 microseconds) on average.

Our Elephant STR implementation (A) requires 203,767 microseconds on average. This represents a 21.60% improvement (C) in time on average when compared to the Snort Reload technique

## 8.3 Analysis with Rule-Sets of a Range of Sizes

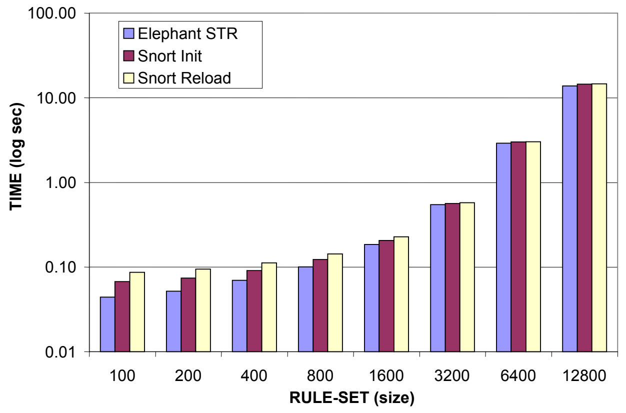

**Figure 4. Elephant STR vs. Default Snort.**

In Section 8.2, we compare Elephant STR with Snort Reload for the task of loading a rule-set of default size. In this section, we compare Elephant STR with Snort Reload for loading rule-sets of a range of sizes.

Each rule-set is created by multiple concatenations of the Snort *ftp.rules* file (which contains signatures and rules for FTP based attacks) and a truncation to the appropriate number of rules. Snort will blindly load multiple identical rules, so the method used to create the rule files should not impact the time required to load the rules.

Figure 4 shows that Elephant STR follows the trend of Snort Reload and the standard Snort startup initialization. It plots the average time consumed by each of the techniques to load an entire rule-set over twenty runs for each of the rule-set sizes. The absolute difference in time between Elephant STR and Snort Reload remains nearly constant at approximately 0.04 seconds throughout the range of rule-sets from size 100 to size 1600.

Because the time required to load large rule-sets is much greater than that required for smaller sets, Figure 4 is on a log scale, which makes it appear that the absolute difference in time between Elephant STR and Snort Reload is less for large rule-sets than for small rule-sets. In fact, we observe the opposite—the difference is approximately 0.04 seconds for 100 rules, but 0.85 seconds for 12,800 rules.

Elephant STR uses the same mechanisms as both Snort Reload and the standard Snort initial rule loading procedures, so it comes as no surprise that Elephant STR follows the same performance trends as those two techniques. On the other hand, we expected that, given the lack of memory de-allocation in our current implementation, the performance of Elephant STR would degrade somewhat more rapidly than the performance of Snort Reload. However, we have not observed this behavior in our evaluation.

## 9. Proposed Enhancements for Reloading

We propose two basic strategies for decreasing the runtime overhead of our online rule-reloading approach. In Elephant STR, the main execution thread is used for loading the new rules and for standard Snort execution. During the time that it takes to reload the rules, packets can be queuing in network buffers. If these buffers become full, traffic can be lost, which can cause the state that Snort knows about to diverge from the actual state of the end-hosts. This can cause attack packets to be missed.

The first enhancement strategy, Elephant MTR (multi-threaded reload), to improve the performance of our rule reloading is designed to decrease the length of the "pause" in normal program execution that occurs when the reloading occurs. In Elephant MTR, we introduce a second helper thread to reload the rules. Then, only the actual rule switchover needs to occur on the main thread.

A disadvantage of this approach is that the CPU time devoted to the rule reloading will be more spread out (because loading will happen concurrently with packet processing), and, therefore, the rule reloading may appear to take longer. To disrupt the processing of packets even less, this second thread can also be run at a lower priority than the main thread, but this might cause the reloading to appear to take even longer. We expect that Elephant MTR can use the same quiescent point as Elephant STR for the rule switchover.

The second strategy is to decrease the memory overhead of our rule reloading. The default Snort rule database loads 1679 rules. In memory, this rule database requires approximately 24MB based on our observations. Therefore, if we load a second rule-set of similar size before de-allocating the current rule-set (such as would be required by Elephant MTR), we temporarily require almost 50MB of RAM. In practice, it seems highly plausible that the two rule-sets will have a great deal of overlap, and that the new rule-set will reflect few changes to the current rule-set except for the addition of new rules. Therefore, we can envision an efficient in-memory comparison and merge of the current and new rule-sets.

## 10. Insights and Lessons Learned

### 10.1 Why Replication is Insufficient

It would be possible to start a new Snort process using the new rule-set while the current process is still running. This would provide an approximation of a way to update Snort's rules with zero downtime (with the added complication of sorting through duplicate alerts generated while both processes are running). A major problem with this strategy is that the state maintained by the current process would still be lost.

This problem could potentially be handled by a state-exchange mechanism that transfers the state from the current process to the new process. However, such a strategy would be complex in that it would require a way for all packets received during the transfer to be applied by the new process to the state that it receives. This might be done by buffering the packets, and it might be made more efficient by requesting incremental state updates. Such a strategy would also be complex in that it would require the current process to enter a quiescent point before transferring the state, a fact that could force the current process to also buffer (and possibly drop) packets.

Such a state-transfer strategy would be more complex than Elephant STR. It is unclear that it would actually be faster or require less packet buffering. We feel the multi-threaded Elephant MTR approach suggested in Section 9 is simpler and would likely require less packet buffering.

### 10.2 Comparison to Software Upgrading

Our Elephant Reload algorithm for NIDS rule updating is simpler than a complete solution to software upgrading. For example, the problem of updating Snort's rule database is less complex than the problem of updating Snort itself.

As we have identified, the Snort rule database is itself stateless. From an architectural perspective, this means that we can simply swap it for a new rule database at a quiescent point. By contrast, if we were to try to update Snort itself, we would need first to save all of the state that is used by the code that is updated, then to find a quiescent point (or perhaps multiple quiescent points if no single point exists in the software), and finally to reload the state (perhaps first converting it to a new format for the new code). Even with this complexity, updating Snort would still be simpler in some ways than updating a distributed application in which multiple systems would have to be coordinated during the update**.**

### 10.3 Architecting NIDSs for Online Updates

Snort has not been designed to allow for simple online rule updating. This is evident in at least two ways: as discussed in Section 7.3, rule loading is intertwined with Snort initialization; and, as discussed in Section 7.4, there are no facilities for removing rules from memory.

For the purpose of online updates, a NIDS would ideally separate rule loading from program initialization, and provide an efficient way to remove a batch of rules from memory.

### 10.4 Guaranteeing Default Behavior

We would like to be able to show conclusively that we do not perturb the behavior of the NIDS with the introduction of our Elephant Reload rule updating technique. In particular, while it is relatively simple to show that the new rules work and that the old rules are no longer active after the rule-set has been updated, it is more difficult to be certain that we have not affected some time-sensitive behavior.

As an example, Snort *threshold* objects seek to minimize the number of alerts sent if the volume of packets that would generate the same type of alert is high. These objects keep track of the number of alerts with the same rule identifier that they have generated during the past period of time; they do not generate alerts if this number reaches a pre-specified threshold. A concern is that the pause in packet processing caused by our single-threaded Elephant STR strategy could upset the behavior of these time-sensitive objects, the preprocessors, or other Snort functionality. Elephant MTR is less likely to suffer this problem.

While this is a concern for Elephant STR, it is clear that the time that Snort takes to process packets will already vary based on a number of factors, including the complexity of the pattern matching of the rules, the resource utilization of the machine on which Snort is running, and the amount of traffic to be analyzed. Therefore, it appears that the time-sensitivity of Snort is not terribly strict.

Automated testing and code-analysis are two possibilities for increasing our certainty in the correctness of our procedure, and are potential areas for future work. Model checking and static analysis would be interesting to explore in this context.

## 11. Related Work

We know of no other work that directly addresses the problem of maintaining protocol state during a NIDS rule upgrade. We also know of no work describing attacks that target a failure to maintain this state. However, there is a variety of work related to the general problem of high availability for NIDS and to our solution for online rule updating.

The fact that NIDSs fail-open is well known. Because of this and a desire to provide reliable network intrusion detection systems, there has been work done in industry to provide high-availability NIDSs. The most prominent solutions focus on replicated load-balancing as a technique for avoiding resource exhaustion problems that can cause a NIDS to fail. TopLayer Inc. sells a network-switch-type product called IDS Balancer [17] that can merge incoming traffic streams and then distribute the merged traffic to multiple NIDSs. IDS Balancer can be configured to follow a variety of policies for distributing the traffic (*e.g.,* by port number, by protocol, or round-robin). IDS Balancer can also sense the failure of a NIDS, and rebalance the traffic accordingly.

There has been work done on tools for offline management of the rule updates that become available when new attacks are discovered. Oinkmaster [6] and SnortCenter [12] provide ways to download rule updates for Snort and to merge the new rules into the current rule-set in an offline fashion. They do not provide any mechanism for applying the updates to a running Snort process. These tools are complementary to online rule-reloading, and can be used in conjunction with Elephant Reload.

There is a body of literature on techniques for software upgrading. Ajmani [1] and Segal and Frieder [10] provide surveys of the literature, focusing on techniques for zero-downtime software upgrading, the former with a particular emphasis on distributed systems. Much of this work could be applicable to our needs, but may be overly complex in that it attempts to provide a general solution for software upgrading and so must assume that application state, or the code that acts on that state, may need to be saved. As we discuss in Section 10.2, our Elephant Reload algorithm avoids this problem because it updates only the rule and attack databases, which can be separated from any code that maintains state.

## 12. Conclusion

There is a critical tension between modern NIDSs that maintain state and the need to restart these NIDSs in order to update their rule and attack-signature databases. This may result in vulnerabilities that hide malicious network traffic in connections about which the NIDS "forgets" after it is restarted.

In this paper, we have demonstrated a specific vulnerability in Snort that takes advantage of this type of behavior. We have proposed Elephant Reload, a technique for online rule-updating that does not require NIDS downtime and which therefore handles this vulnerability.

Our Snort-based Elephant STR implementation of Elephant Reload avoids the vulnerability, and, additionally, provides a 21.6% average performance benefit over the current technique built into Snort for reloading its default rule-set.

There are a number of potential enhancements to our technique and potential expansions on our work. We have presented one implementation for one NIDS; it would certainly be possible to expand this to multiple implementations or to different NIDSs. Our current implementation is not ready for production use in that it does not de-allocate the memory for the old rule-set; we expect this problem to be fully solvable with additional engineering. Another interesting area of potential work is the exploration of ways to verify that the rule updates are valid and will behave as expected.

## Acknowledgments

We would like to thank Sonya Johnson, Deepti Srivastava, and the anonymous referees for their helpful comments on drafts of the paper.

This work is supported by the Army Research Office through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at Carnegie Mellon University.

## 13. References

[1] Ajmani, Sameer. *A Review of Software Upgrade Techniques for Distributed Systems*. http://pmg.csail.mit.edu/~ajmani/papers/review.pdf. August 2002.

[2] Caswell, Brian, *et al. Snort 2.0 Intrusion Detection*. 2003. Syngress. Rockland, MA.

[3] Coretez, Giovanni. *Fun with Packets: Designing a Stick.* Draft White Paper on Stick. http://www.eurocompton.net/stick/. June 2004.

[4] Mukherjee, Biswanath *et al*. *Network Intrusion Detection*. IEEE Network. 8(3). May/June 1994. pp. 26-41

[5] Newman, David *et al. Crying Wolf: False Alarms Hide Attacks*. Network World. June 24, 2002.

[6] *Oinkmaster*. http://oinkmaster.sourceforge.net/. June 2004.

[7] Paxson, Vern. *Bro: A System for Detecting Network Intruders in Real-Time*. 7th Annual USENIX Security Symposium. Amsterdam, Netherlands. January 1998. pp. 2435-2463.

[8] Ptacek, Thomas H. and Newsham, Timothy N. *Insertion, Evasion, and Denial Of Service: Eluding Network Intrusion Detection.* Technical Report, Secure Networks, Inc., January 1998.

[9] Roesch, Martin. *Re: [snort] multithreaded snort?*. Snort Users Mailing List. February 25, 2000.

[10] Segal, Mark E. and Frieder, Ophir. *On-the-Fly Program Modification: Systems for Dynamic Updating*. IEEE Software, vol. 10, no. 2. March 1993. pp. 53-65.

[11] *Snort*. http://www.snort.org. June 2004.

[12] *SnortCenter – Snort Management Console*. http://users.pandora.be/larc/. June 2004.

[13] *Snort Manual, Chapter 2.8.4*. http://www.snort.org/docs/snort_manual/node17.html. June 2004.

[14] *Snot V0.92 alpha*. http://www.stolenshoes.net/sniph/snot-0.92a-README.txt. June 2004.

[15] Stevens, W. Richard, *Advanced Programming in the UNIX Environment*. 1992. Addison-Wesley. Reading, MA.

[16] *TCPDUMP/LIBPCAP*. http://www.tcpdump.org/. June 2004.

[17] *TopLayer IDS Balancer*. http://www.toplayer.com/content/products/intrusion_detection/ids_balancer.jsp. June 2004.