A Framework for Malicious Workload Generation

Joel Sommers Vinod Yegneswaran Paul Barford
University of Wisconsin–Madison
jsommers,vinod,pb@cs.wisc.edu

ABSTRACT

Malicious traffic from self-propagating worms and denial-of-service attacks constantly threatens the everyday operation of Internet systems. Defending networks from these threats demands appropriate tools to conduct comprehensive vulnerability assessments of networked systems. This paper describes MACE, a unique environment for recreating a wide range of malicious packet traffic in laboratory testbeds. MACE defines a model for flexible composition of malicious traffic that enables both known attacks (such as the Welchia worm) and new attack variants to be created. We implement this model in an extensible library for attack traffic specification and generation. To demonstrate the capability of MACE, we provide an analysis of stress tests conducted on a popular firewall and two popular network intrusion detection systems. Our results expose potential weaknesses of these systems and reveal that modern firewalls and network intrusion detection systems could be easily overwhelmed by simple attacks launched from a small number of hosts.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General—*Security and protection (e.g., firewalls)*; C.4 [Performance of Systems]: Performance attributes

General Terms: Measurement, Performance, Security

Keywords: Traffic Generation, Network Intrusions

1. INTRODUCTION

Network outages due to self-propagating worms and denial-of-service attacks have been widely reported over the past few years. Despite efforts by the research and operational communities to mitigate these threats, many Internet systems remain vulnerable. One reason for this insecurity is that systems and protocols are often not designed with deliberate consideration of threat models. An example that has received recent attention is the TCP initial sequence number protocol vulnerability that has the potential to inflict chaos by disrupting BGP sessions in the backbone of the Internet [5]. Another reason is that system behavior under heavy load is often unpredictable. Although scaling properties of commercial software systems and routing hardware are quite impressive, they are typically not developed with malicious traffic conditions in mind. For example, the case study by Ogielsky *et al.* reports on how software bugs combined with load introduced by the unprecedented spatial diversity of Code-Red worm traffic and elevated BGP activity led to widespread cascading outages [10].

The potency of Internet worms and viruses has continued to evolve since the Code-Red outbreak. Examples include high-speed worms such as SQL-Slammer and more recently multi-modal worms/viruses such as Agobot which package exploits for many known vulnerabilities. We have also witnessed the Witty worm which exploits vulnerable network stacks in firewalls. Witty targets all destination ports equally, hence it cannot be neutralized by simple techniques like port blocking at network gateways. These attacks underscore the need for scalable intrusion detection systems to protect large networks by performing signature matching at Internet gateways. They also galvanize the need for better tools for evaluating the resiliency of routers, middleboxes and intrusion detection systems.

This work has two primary objectives. The first is to create a performance benchmarking tool that enables assessment of quality of service degradation (the effect of maltraffic on good traffic) and resilience of middleboxes and network intrusion detection systems (NIDS) over a range of malicious traffic volumes. The state of the art for benchmarking firewalls and NIDS is still considered nascent, and there are no universally accepted standards. Current best practice is to gather and use malware itself. This approach is limited in its ability to scale traffic volume and compose previously unseen attack traffic. A flexible research-oriented toolkit for producing maltraffic is therefore a critical component for the development and standardization of benchmarks for NIDS and firewalls. Our second objective is to recreate attack traffic scenarios in a laboratory setting for black-box testing of protocol suites, software, and hardware systems. This capability will enable us to move toward quantitative measurement and assessment of claims regarding the degree to which individual network devices contribute to large-scale correlated failures.

This paper describes the design and implementation of a flexible, extensible toolkit called the **M**alicious tr**A**ffic **C**omposition **E**nvironment (MACE). MACE provides the basic building blocks for recreating a large set of known attacks. MACE satisfies the following requirements:

1. Generate a large, diverse set of attacks,

2. Generate and control benign background traffic,
3. Compose attacks in a high-level language.

The MACE model decomposes attacks into three components: exploit, propagation and obfuscation. To our knowledge, MACE is the first tool to adopt an extensible approach to systematic attack synthesis through a modular attack composition framework. MACE enables resiliency of systems to resource exploits and traffic mix to be evaluated. MACE is distinguished from efforts like Thor which focus on obfuscation methods and individual session morphing [6]. Results from such work can be easily folded into MACE.

We discuss the MACE framework in greater detail and provide example attack traffic configurations in § 3. In § 4 we demonstrate MACE by providing a case study of a popular firewall (a Cisco PIX) and two NIDS (Snort and Bro) under varying traffic mixes. Our results reveal that modern firewalls and NIDS could be easily overwhelmed by simple attacks launched from a small number of hosts. In most instances, the resilience of the devices to particular attacks varies with respect to the degree of connection state maintained by each device. The case study also illustrates the capabilities of MACE as a performance benchmarking tool.

2. RELATED WORK

Internet traffic generation for measuring application and network device performance has been well-studied. Harpoon reproduces network traffic in an application-oblivious manner [17]. Numerous application-aware traffic generators like SURGE produce workloads to stress-test web servers [9]. These tools are complementary to MACE and can be used to generate legitimate (benign) background traffic.

Taxonomies of malicious traffic inform the design and development of the MACE attack types and propagation models. One such taxonomy of DDoS attack characteristics is provided in [15]. A similar taxonomy of Internets worms based upon target discovery, carriers, activations, payloads and attackers is proposed in [18]. Our study is also related to the work by Lippman *et al.* which provides a dataset for evaluation of NIDS [14].

Our laboratory measurement of the performance of a firewall middlebox is complementary to the work of Allman, who measured the performance of an operational middlebox infrastructure in [7].

MACE is intended for black-box evaluation of network infrastructure resiliency and is not tuned to exploit specific implementation features of NIDS or firewalls. Related efforts [11, 13] have focused on carefully designed attacks on known or suspected algorithms or implementations.

Two other analogs to our work are Mucus [16] and Nessus [3]. Mucus is a tool for black-box testing of NIDS systems. As a benchmarking tool for NIDS systems, Mucus' evaluation criteria are *alert correctness* and *alert quality*. Nessus is a tool for penetration testing of network hosts. MACE differs considerably from both of these tools in its objectives. The MACE framework for malicious traffic generation accommodates its use both for penetration testing or as a NIDS benchmarking tool. Our goals of performance benchmarking NIDS and middleboxes do not require a complete database or accurate replication of specific attacks. While it is unnecessary, for example, to faithfully replicate all variants of the Beagle worm, it is desirable to have representatives from all classes of attacks. Finally, since the MACE model isolates obfuscation and propagation from exploit signatures, it can generate attacks that are not present in current vulnerability databases. Building a library of known exploits, obfuscations, and propagation models enables attacks composed of any permutation of these subcomponents. The spectrum of attacks is limited by the set of available subcomponents.

3. MACE FRAMEWORK

We begin by defining the abstract model for MACE. We then describe implementation details of the MACE toolkit with examples of attack instances and show how they can be composed.

3.1 Abstract Model Definition

The conceptual model for MACE is illustrated in Figure 1. It provides flexibility in specifying the base characteristics of malicious traffic which we define as the following:

- Exploit Model — set of vulnerabilities that are part of the attack sequence.
- Obfuscation Model — morphs in the header or payload to enable the exploit to elude NIDS. These could either be at the network layer or at the application layer.
- Propagation Model — order in which victims are chosen to be attacked.
- Background Traffic Model — legitimate traffic flow in the network.

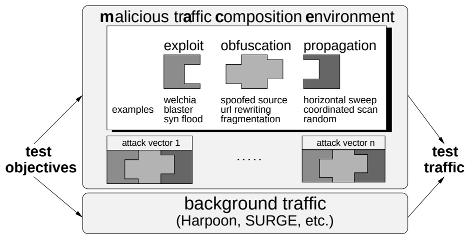

Figure 1: The MACE architecture. Test objectives inform the selection and composition of exploit, obfuscation, and propagation components into a series of attack vectors. Existing tools such as Harpoon produce the desired benign background traffic.

3.2 Model Realization

The modular MACE library consists of the exploit, obfuscation, and propagation components defined by the model, as well as a number of functions to support interpretation, execution, and exception handling of attack profiles. Our objective is not to provide a complete attack database, but rather to support a set of basic building blocks that can be used to create both known and custom attack vectors.

MACE is currently implemented in the Python programming language. The dynamicism of a language like Python is important for MACE as it enables sequences of exploits or obfuscations to be fabricated on-the-fly. Python has libraries that support many application layer protocols, such as HTTP, NNTP, and SMTP, allowing application-specific exploits to be easily created. For evolving scalability requirements, we plan to migrate key components of the library to C, extending the capabilities of the Python interpreter much in the way ns-2 [4] extends `Tcl/Tk`.

The building blocks of MACE include the following:
Payload Construction Elements - Payload elements in MACE are defined as character arrays. In practice, these are often payloads from various higher level protocols such as HTTP, NetBIOS/SMB or DCE RPC. The following example defines an HTTP GET request for the file index.html.

```
[ httprequest, ( 'method':'GET', 'absolute_path':'/index.html') ]
```

Header Construction Elements - Attack traffic often requires the use of raw sockets to construct custom TCP, UDP and IP headers. The header construction elements in MACE modify specific attributes without exposing the entire header to the user. The following example illustrates the definition of a TCP SYN packet. The source and destination IP addresses and ports are defined by the propagation elements. Other unspecified fields are appropriately filled with default or random values.

```
[ rawtcp, ( 'th_flags':'TH_SYN' ) ]
```

Obfuscation Elements - The obfuscation elements can be specified at the network layer or at the application layer. An example of network layer obfuscation is IP fragmentation. Examples of application layer obfuscation include HTTP URL encoding techniques such as Bare-byte or Unicode encoding used by worms such as Nimda [8]. The following example applies a Unicode obfuscator to the example HTTP request given above:

```
[ httprequest, {'method':'GET',
        'absolute_path':URLObfuscator.uencode('/index.html') } ]
```

Propagation Elements - MACE supports different propagation models via `AddressPool` objects. Each AddressPool is instantiated with a list of CIDR prefixes and port ranges, along with an indication of how to generate addresses from the given pool. Address pools may be traversed randomly, horizontally (sequentially across IP addresses, sequentially across ports), vertically (sequentially across ports, sequentially across IP addresses), or by a more complex methodology definable by the user. The following example illustrates an address pool defined to perform a vertical sweep of the target IP address `10.42.1.1`.

```
target_pool = AddressPool(AddressPool.Vertical, '10.42.1.1:1-65536')
```

3.3 Example Test Scenarios

We focus on four attack vectors:

- *SYN flood*: A standard denial-of-service attack that overwhelms the target system by sending a large number of TCP SYN packets.
- *Welchia*: A `ping` followed by a series of HTTP requests designed to exploit a buffer overflow in the WebDAV module of Microsoft's IIS web server [1].
- *Rose*: An attack that exploits poorly implemented handling of IP fragments. Two small fragments are sent, one with an offset of 0 and one with a large offset. Some network stacks reserve memory for the fragment hole, so a series of Rose packets can exhaust memory [12].
- *Blaster*: An attack that exploits a buffer overflow in Microsoft Windows RPC service (epmapper) [2].

The four attack profiles are realized in the example code fragment shown in Figure 2. Attack profiles are defined by their vulnerability exploit and propagation method. A full exploit is a sequence of *generator* and *validator* steps, along with parameters. A generator builds packet traffic using the payload and header construction and obfuscation building blocks. A validator collects and processes responses from the attack target, verifying that the generated traffic evoked an appropriate response. Each step in an attack vector is executed as long as they are successful. An exploit step may simply be "create a TCP packet with the SYN flag set" (line 1). Other steps might be as complicated as "create an HTTP GET request for the document '/' and validate that the HTTP response contained a 200 (success) code and that it was produced by a Microsoft web server" (lines 6–7). Some generator steps are not followed by validation, *e.g.*, the Rose attack where validation is unnecessary (lines 12–15). Since generators and validators are simply Python functions, it is easy to define new exploit steps. Propagation models for the attack profiles are defined by one or more address pools (lines 23–25). Finally, attacks are sent using the `send_once` or `send_periodic` functions (lines 27–28). `send_periodic` is a convenience function used to produce attack traffic for a given duration (delaying a specified amount of time between successive call to `send_once`).

```
synflood = [ [ rawtcp, { 'th_flags':TH_SYN} ] ]                                          1
                                                                                         2
bad_string = '...' # the buffer overflow string - not defined here                       3
welchia = [                                                                              4
    [ ping ],                                                                            5
    [ httprequest, { 'method':'GET' },                                                   6
       httpvalidate, { 'Server':'Microsoft-IIS/5.0' ],                                   7
    [ httprequest, { 'method':'SEARCH' },                                                8
       httpvalidate, { 'code':411, 'Server':'Microsoft-IIS/5.0'} ],                      9
    [ httprequest, { 'method':'SEARCH', 'absolute_path':bad_string } ] ]                10
                                                                                        11
rosepayload = '\0' * 32 # just a small fragment                                         12
rose = [                                                                                13
    [ rawudp, { 'frag_offset':0, 'frag_flag':IP_MF, 'payload':payload } ],              14
    [ rawudp, { 'frag_offset':8100, 'payload':payload } ] ] # 64800 byte offset         15
                                                                                        16
bindreq = '...' # the DCE RPC bind request - not defined here                           17
overflow = '...' # the buffer overflow exploit request - not defined here               18
blaster = [                                                                             19
    [ dcerpcbind, { 'bindreq':bindreq }, dcerpcbindack, { 'bindreq':bindreq } ],        20
    [ dcerpcreq, { 'payload':overflow } ] ]                                             21
                                                                                        22
src_pool = AddressPool(AddressPool.Vertical, '10.42.0.0/16:1-65536')                    23
dst_pool = AddressPool(AddressPool.Random, '10.52.128.0/20:1024-65536')                 24
http_pool = AddressPool(AddressPool.Horizontal, '10.52.128.0/20:80')                    25
                                                                                        26
send_once(src_pool, dst_pool, rose)                                                     27
send_periodic(src_pool, http_pool, welchia, duration=60, delay=0.001)                   28
```

Figure 2: A MACE code fragment, similar to what was used in our experiments.

4. SECURITY SYSTEM PERFORMANCE EVALUATION

To demonstrate MACE's capability, we examined performance characteristics of three standard network security systems: a firewall middlebox and two network intrusion detection systems running on commodity hardware.

4.1 Test Environment

The firewall we tested is a Cisco PIX 515e. It contains three Fast Ethernet interfaces, 64MB of RAM, a 433 MHz Intel Pentium II, and runs version 6.2(2) of the PIX Firewall operating system. It is a typical device deployed as the first line of defense for edge networks, implementing basic packet filtering and network address translation (NAT). The network intrusion detection systems were Bro (version 0.8a79) and Snort (version 2.1.1). Bro generally maintains significant connection state, while Snort does not. Each NIDS ran on a separate workstation with a 2 GHz Intel Pentium 4 processor and 1 GB of RAM. FreeBSD 5.1 was installed

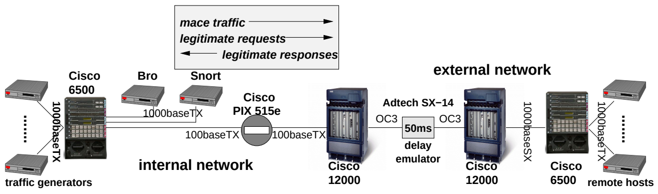

Figure 3: Experimental Environment. Legitimate and malicious traffic originate from the internal network and are directed toward the external network. The PIX firewall separates the inside and outside networks and performs network address translation.

on each machine[1]. For each NIDS, we used a default set of rules. Our Snort instance used the default `snort.conf` included with the software and Bro used the `mt` policy.

The PIX and NIDS were configured in a testbed as shown in Figure 3. The setup mimics an edge network connected to an ISP, with legitimate background traffic and attacks focused on a remote network. The PIX resided between the edge ("internal") network and the ISP ("external") network. The internal network contained traffic generators for MACE and background traffic, and the two NIDS. All network traffic received from or sent to the PIX was duplicated on the links connected to the NIDS. In the external network, we used a hardware propagation delay emulator (Adtech SX-14) between two backbone-class routers (Cisco 12000) to create a round-trip time of roughly 100 milliseconds between the traffic generators and the target servers. We used popular enterprise-scale switches (Cisco 6500) to aggregate traffic at the endpoints.

On each host in the internal network, we created 2^{12} alias addresses and on the remote hosts we created 2^8 aliases. The PIX performed network and port address translation between hosts on the internal network and a pool of 2^8 addresses routable across the external network. Also, the PIX performs an implicit packet filtering based on its NAT configuration: it only performs NAT or port address translation (PAT) for local addresses that are part of its configuration. Packets from any other source address are dropped.

Using two levels of benign background traffic, we generated attack traffic using a set of five exploits and six levels of offered load. Two levels of background traffic, "low" and "high", were generated using Harpoon [17] and were tuned to averages of 20 Mbps and 70 Mbps, respectively. Source and destination addresses for legitimate traffic were chosen randomly from the pool of 2^{12} source addresses and the pool of 2^8 destination addresses. The six levels of attack load were generated by using one to five hosts running MACE. For each exploit, the MACE processes on a single host were configured to generate roughly 1 Mbps of traffic, regardless of background traffic level.

The exploits we used were (1) Welchia worm traffic, (2) SYN flood denial-of-service attack, (3) a SYN flood with spoofed source addresses, (4) the Rose fragment attack, and (5) a multi-modal attack consisting of the previous four exploits plus Blaster worm traffic. Each host running MACE used a source address pool of 2^{12}, as described above. For the SYN floods and Rose attack, we horizontally traversed the source address pool. For the two worm exploits, we randomly (uniformly) sampled the source address pool. All attack traffic was directed toward a single address on the remote network.

4.2 Test Measurements

For the 52 distinct experiments, we measured CPU and memory utilization at all three systems every five seconds. We also measured packet counts in and out of the PIX every five seconds and the number of reported packet drops using SNMP, and took packet traces on either side of the PIX. At the two NIDS hosts, we verified and used the capabilities of each software package to report received packet volume and the number of dropped packets. Packets are dropped by each system due to overflow of the queue of incoming packets received by the packet filter (each NIDS uses the Berkeley Packet Filter and libpcap for packet capture). For packet drops at each NIDS, we did not discriminate between benign and malicious traffic. For the PIX, we used the packet traces to measure drops of benign packets. Each experiment was run for six minutes, including a one minute warm-up phase from which measurements are discarded.

4.3 Results

Figure 4 shows average CPU utilization and Figure 5 shows packet loss measurements for each experiment. The two columns in each figure correspond to low and high background traffic levels, and the three rows display results for each device. The first feature to notice in the plots is the diversity of responses of each system to distinct MACE attack profiles. Except for a few cases, there is also a noticeable, and sometimes very large, divergence between the first two data points. These points correspond to zero malicious traffic and a single MACE host. We discuss detailed results for each device class (firewall and NIDS) below.

4.3.1 Effects on the PIX firewall

For the PIX, the Rose attack has the least effect on performance. When processing fragmented packets, the PIX keeps a queue of (by default) 200 fragments in order to reassemble them before forwarding them to the remote network. If the missing fragments do not arrive in a configurable amount of time (the default is 5 seconds), the fragments are dropped. For the Rose attack, the fragment queue becomes full shortly after starting MACE. When fragments arrive that cannot be queued, they are dropped. Although there is no fragmented legitimate traffic in our setup, these packets would very likely be dropped even with an attack rate of just over 40 Rose packets per second. Also, queues for each interface and the fragment reassembly queue share a common buffer pool. Since there are fewer buffers available during a Rose attack, interface queues are more likely to fill, causing additional packet drops[2].

The non-spoofed SYN flood is the attack with the most impact on the PIX. Since the PIX is performing NAT, it must maintain state for each connection. Even with a single MACE host, all 64 MB of system memory is used after a

[1]On each host we modified the kernel parameter `debug.bpf_bufsiz` from 4096 bytes to 524,288 bytes, as suggested in the Bro documentation. Snort presumably can benefit from this change as well so we applied the change to each NIDS host.

[2]The PIX documentation notes that it is possible to set the fragment reassembly maximum queue length to be equal to the total number of buffers available. The documentation warns that such a configuration would enable fragment attacks to be effective denial-of-service attacks.

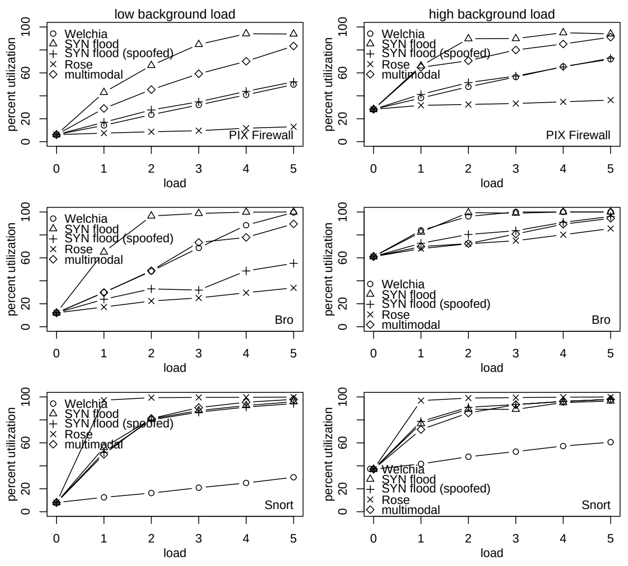

Figure 4: CPU utilization measurements. Results for the PIX, Bro and Snort are in top, middle, and bottom rows, respectively. Left and right columns show results for low and high background traffic loads, respectively. Load levels along the x-axis correspond to number of MACE hosts used in each test. Each MACE host generates roughly 1 Mbps of traffic regardless of attack or background traffic level.

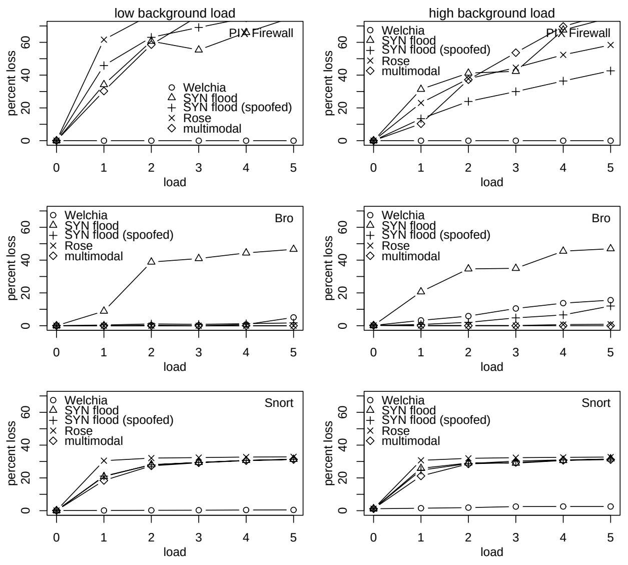

Figure 5: Packet loss measurements. Results for the PIX, Bro and Snort are in top, middle, and bottom rows, respectively. Left and right columns show results for low and high background traffic loads, respectively. Plots for the PIX show packet drops for benign traffic only, while plots for Bro and Snort show aggregate packet drops. Load levels along the x-axis correspond to number of MACE hosts used in each test. Each MACE host generates roughly 1 Mbps of traffic regardless of attack or background traffic level.

short while because of this state requirement[3]. While there is still some memory available for buffering packets as they flow through the system, this memory pool is now much smaller and consequently, the PIX has diminished ability to absorb bursts of packets. This situation does not occur with the spoofed SYN flood, since the source addresses do not conform to the NAT configuration at the PIX and are dropped. In our traces, we see persistent dropping of legitimate packets during the non-spoofed SYN flood and multimodal experiments (in both low and high background traffic regimes) and aggregate traffic rates through the PIX clearly show the well-known poor performance of TCP in the face of such high packet loss: for the low background SYN flood experiment using only one MACE host, the inbound (external to internal) packet rate through the PIX for the spoofed SYN flood is nearly twice that of the non-spoofed SYN flood. For experiments with more than one MACE process, the results are more dramatic. In summary, maintaining state and managing system resources under a low rate non-spoofed SYN flood is difficult even for a specialized device. Considering the rate at which resources consumed by the embryonic connections are reclaimed by the PIX, there is probably a "sweet spot" at which SYNs can be sent at a low enough rate to cause problems for good traffic, but are not at a high enough rate to easily detect.

The Welchia and Blaster worms, as with other worms, are typically short flows, so even with a low attack rate the primary effect on the PIX is an increased rate of connection initiations. To the PIX (without any special packet filters installed) these worms look like benign traffic and are treated the same as all other legitimate packets. In our experiments, the PIX appears to be sufficiently provisioned to handle the increased volume.

Finally, it is interesting to note that while all Rose and spoofed SYN flood packets are dropped by the PIX, these attacks have peculiar effects on CPU usage. Without detailed internal information of the PIX, we can only surmise that the path for handling IP fragments is significantly faster than the process of matching a source address to the NAT configuration at the PIX (though, as pointed out above, there is a potential denial-of-service problem related to fragment processing).
Summary: (1) Non-spoofed low-rate SYN floods are effective resource exploits leading to significant service degradation, and (2) Obfuscations via packet fragmentation are effective resource exploits against poorly configured systems.

4.3.2 Effects on Bro and Snort

There is a sharp contrast between behavior of Bro and Snort. For example, the Rose attack has little effect on Bro, but an enormous impact on Snort. Since Bro and Snort receive raw packets from the network interface, they must perform reassembly of fragments. Bro is clearly able to handle fragments more efficiently than Snort, even though Snort maintains much less overall state than Bro.

For Bro as with the PIX, the non-spoofed SYN flood has the greatest impact on CPU load and packet loss. For the

[3]There are timeouts defined in the PIX to reclaim resources used by idle and half-closed connections, and a feature called "floodguard" which essentially prioritizes which resources to reclaim in order to deal with resource shortages. Our experiments are shorter than either one of the default values for these timeouts (1 hour for idle connections, 10 minutes for half-closed connections).

non-spoofed SYN flood, Bro maintains state for all attack-induced connections. Although Bro periodically expires idle connection state, the rate of SYNs in our experiment was high enough that Bro eventually exhausted available memory[4]. Since the table of known connections continues to grow during this attack, connection state lookups are more costly. The resulting effect on packet drops is clearly shown in the center column graphs of Figure 5. The reason the spoofed SYN flood has relatively little impact on Bro is that the PIX silently blocks the spoofed SYNs so Bro will never see a SYN/ACK response. Apparently this lack of response allows Bro to flush the embryonic connection state in a more efficient manner.

Except for the Welchia attack, Snort performs similarly under all attacks. The SYN floods, Rose, and multimodal attacks each contain packet-level attacks in contrast to Welchia, which (at least from the perspectives of IP and TCP) looks like legitimate background traffic to Snort[5]. Efficiently processing ill-formed packets and pathological packet sequences is clearly a requirement and a challenge for NIDS.

With respect to packet loss, Bro and Snort again exhibit contrasting behavior. Except for the Welchia attack, Snort consistently drops roughly 20-30% of all packets once MACE traffic is introduced. Bro, despite maintaining significant connection-level state, drops a relatively small proportion of packets except for the non-spoofed SYN flood attack. For both NIDS, any significant level of packet dropping will affect the ability of the tool to detect ongoing attacks. Additionally, knowledge of packet dropping behavior could be exploited by an attacker to launch a relatively benign low-rate attack in order to mask one that is more insidious.

Summary: (1) Multiple attack vectors are effective resource exploits leading to packet loss, implying degraded detection rates, and (2) The marginal impact of resource exploits does not appear to be greater for NIDS maintaining connection state.

5. CONCLUSIONS AND FUTURE WORK

The escalation of malicious activity in the Internet motivates the need for better tools to measure the resiliency of routers and middleboxes to malicious traffic. To address this need, we propose MACE, a framework for malicious network traffic generation. The MACE architecture is composed of three building blocks: exploits, obfuscators, and propagation elements. These components define and create malicious traffic for use in laboratory testing of routers and network security infrastructure. We provided experimental results of measurements conducted on a popular firewall and two network intrusion detection systems to document the varying responses of these systems to malicious traffic. Our results show that relatively low rates of attack traffic can exploit the overheads of maintaining connection state or inefficient processing of certain packets.

A tool like MACE can be used for testing and refining the operation of network systems, but if in the wrong hands, could be used for generating malicious traffic in the live Internet. Our plan for making MACE available to a wider community is to supply the code only to legitimate researchers and, to the best of our ability, keep careful documentation regarding who has copies of the code. Despite these precautions, there remains the possibility that MACE could be misused. While MACE cannot self-propagate, it could be used for seeding new worms.

We plan to expand the list of exploit and obfuscation building blocks within MACE and to make improvements to the volume of exploit traffic that MACE is able to produce. We believe these enhancements will facilitate the laboratory emulation of large-scale failure scenarios using more elaborate physical and logical topologies and a greater diversity of network devices.

[4]We also experimented with the `reduce-memory` policy script with Bro, which caused an increase in CPU usage and higher packet loss rates for all attacks.

[5]The default configuration of Snort lacks specialized rules for processing HTTP traffic. Enabling HTTP-specific rules causes higher CPU load and packet drops for all attack profiles.

6. ACKNOWLEDGEMENTS

We thank Somesh Jha, Jon Giffin, Shai Rubin and the anonymous reviewers for helpful comments.

This material is based upon work supported by the National Science Foundation under Grant No. 0335234 and by support from Cisco Systems. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Cisco Systems.

7. REFERENCES

[1] Microsoft Security Bulletin MS03-007. http://www.microsoft-.com/technet/security/bulletin/MS03-007.mspx, 2003.

[2] Microsoft Security Bulletin MS03-026. http://www.microsoft-.com/technet/security/bulletin/MS03-026.mspx, 2003.

[3] Nessus. http://www.nessus.org, 2004.

[4] The Network Simulator – ns-2. http://www.isi.edu/nsnam/ns, 2004.

[5] NISCC Vulnerability Advisory 236929. http://www.uniras.gov.uk/vuls/2004/236929/, 2004.

[6] THOR: A Tool to Test Intrusion Detection Systems by Variations of Attacks. http://thor.cryptojail.net/, 2004.

[7] M. Allman. On the Performance of Middleboxes. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, 2003.

[8] E. J. Aronne. The Nimda worm: An overview. http://www.sans.org/rr/papers/36/95.pdf, 2001.

[9] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Perfromance Evaluation. In *Proceedings of ACM SIGMETRICS*, 1998.

[10] J. Cowie, A. Ogielsky, B. Premore, and Y. Yuan. Global Routing Instabilities Triggered by CodeRed II and Nimda Worm Attacks. http://www.renesys.com/projects/bgp_instability, 2001.

[11] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security*, 2003.

[12] Gandalf. IP Fragmentation $--$ > The Rose Attack. http://www.securityfocus.com/archive/1/359144, 2004.

[13] W. Lee, J. B. Cabrera, A. Thomas, N. Baliwalli, S. Saluja, and Y. Zhang. Performance Adaptation in Real-Time Intrusion Detection Systems. In *Proceedings of RAID*, 2002.

[14] R. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman. Evaluating Intrusion Detection Systems: 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of IEEE Security Symposium*, 1998.

[15] J. Mirkovic and P. Reiher. A Taxonomy of DDoS Attack and DDoS Defence Mechanisms. *ACM SIGCOMM Computer Communication Review*, 32(2), 2004.

[16] D. Mutz, G. Vigna, and R. Kemmerer. An Experience Developing an IDS Simulator for the Black-Box Testing of Network Intrusion Detection Systems. In *Proceedings of ACSAC*, 2003.

[17] J. Sommers and P. Barford. Self-Configuring Network Traffic Generation. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, 2004.

[18] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *Proceedings of CCS Worms*, 2003.