SANS Institute
Information Security Reading Room

Mitigating Attacks on a Supercomputer with KRSI

Billy Wilson

Mitigating Attacks on a Supercomputer with KRSI

GIAC (GCUX) Gold Certification

Author: Billy Wilson, billy_wilson@byu.edu
Advisor: *Clay Risenhoover*

Accepted: 11/14/2020

Abstract

Kernel Runtime Security Instrumentation (KRSI) provides a new form of mandatory access control, starting in the 5.7 Linux kernel. It allows systems administrators to write modular programs that inject errors into unwanted systems operations. This research deploys KRSI on eight compute nodes in a high-performance computing (HPC) environment to determine whether KRSI can successfully thwart attacks on a supercomputer without degrading performance. Five programs are written to demonstrate KRSI's ability to target unwanted behavior related to filesystem permissions, process execution, network events, and signals. System performance and KRSI functionality are measured using various benchmarks and an adversary emulation script. The adversary emulation activities are logged and mitigated with minimal performance loss, but very extreme loads from stress testing tools can overload a ring buffer and cause logs to drop.

1. Introduction

Systems administrators of high-performance computing (HPC) sites face the daunting task of securing research data without sacrificing peak system performance. They facilitate cutting-edge research that contractually comes with tight deadlines and stringent data security requirements. Satisfying both time and security constraints is an ongoing challenge that often requires novel approaches to old problems. This paper describes and tests Kernel Runtime Security Instrumentation (KRSI), a new Mandatory Access Control (MAC) extension in Linux. It allows systems administrators to program very specific and targeted MAC policies that potentially avoid the performance impact of large MAC extensions.

This author will refer to the technology as KRSI because it is a distinctive acronym that its creator continues to use in his presentations (Singh, 2020 July). However, the reader should be aware that this technology has been referred to as LSM BPF Hooks by Linux kernel developers (Corbet, 2019 December; Corbet, 2020) and LSM Probes in user-space applications (Olsa, 2020).

KRSI is a Linux Security Module (LSM) that hooks into the same kernel security events as SELinux and AppArmor, but rather than provide a major MAC extension, it lets an administrator compile and attach small, modular programs that control whether an action is allowed or denied (Singh, 2020 March). An administrator can attach their own custom code that controls file access, network activity, process execution, and much more.

This technology can potentially be adopted as an LSM of choice in high-performance computing. The fact that LSMs are disabled at HPC sites is prevalent enough that NIST included in their 2016 Action Plan Draft for HPC Security, “Consider why tools like SELinux don’t get used” (National, 2016). Many systems administrators disable SELinux because of the negative performance impact it has on both synthetic benchmarks and real-world applications (Larabel, 2020).

Researching KRSI is a continuation of previous research on BPF Probes (Wilson, 2020 June). BPF Probes *detected* low-profile attacks against servers with little

performance impact; however, the probes were limited in their ability to *mitigate* the attacks. In contrast, KRSI can provide both detection and mitigation.

In this research, new tracing scripts were written that used KRSI to detect and mitigate low-profile attack techniques. An environment of eight compute nodes was configured, booted from the latest available stable Linux kernel as of 21 September 2020. Benchmarking tools were run on the compute nodes to measure their baseline performance. A series of low-profile attacks were then launched, along with the tracing scripts, during a second set of benchmarks. Performance was compared and the functionality of the scripts was analyzed. Five appendices have been included that provide a KRSI tutorial, various source code, and benchmark results.

2. Technology Review

Writing KRSI programs is an advanced topic. To make the subject more approachable, a brief review of the technologies that KRSI is built upon is provided.

2.1. BPF

KRSI is ultimately made possible by Berkeley Packet Filter, or BPF. Though traditionally recognized as a network filter tool, BPF is now a system-wide tracing subsystem for Linux.

Using BPF, systems administrators can write and attach small tracing programs to places of interest in the operating system. The programs can be attached to defined tracepoints or arbitrary functions, both in the kernel and user-space (Gregg, 2020). When a function is entered or exited, the BPF program can view the data passed to the function and data returned from the function. Though used primarily for performance analysis, BPF also serves as a valuable tool for security monitoring (Gregg, 2017).

A tracer called “bpftrace” was the tool of choice in previous research by Wilson (2020 June). It simplified the writing and attachment of BPF programs by providing an AWK-like syntax:

```
#!/usr/bin/bpftrace

probe1 /filter/ { action }
probe2, probe3 /filter/ { action }
```

Figure 1. Syntax of bpftrace

The three main components of bpftrace syntax are the probe, the filter, and the action. The probe specifies the tracepoint or function where the BPF program will be attached, the filter qualifies which events are processed, and the action defines the action to take when the event fires.

Security practitioners can leverage bpftrace to achieve a remarkable depth of visibility on Linux systems. Wilson (2020 June) provided bpftrace scripts to detect cryptocurrency software traffic, privilege escalation attempts, network pivot attempts, and SSH proxy creation

Despite its broad monitoring capabilities, bpftrace was limited in its ability to mitigate attacks. At best, the tool could respond to an event by sending a signal to a process or by unsafely spawning a shell to perform an action (Gregg, 2020 September).

Wilson (2020 June) concluded that future research could focus on KRSI, an up-and-coming MAC extension that was better positioned to mitigate attacks with BPF.

2.2. Linux Security Modules

Another essential prerequisite to writing KRSI programs is understanding how Linux Security Modules work. MAC extensions in Linux are implemented as LSMs, and this includes KRSI.

The LSM framework made it possible to extend the security model of Linux within the mainline kernel. Before its existence, Linux was limited to Discretionary Access Control, or DAC (Barkley, 1994). Projects that added MAC to Linux, such as Medusa, RSBAC, DTE, and the NSA's SELinux, had to maintain their own custom-patched kernels (Smalley, 2002 May).

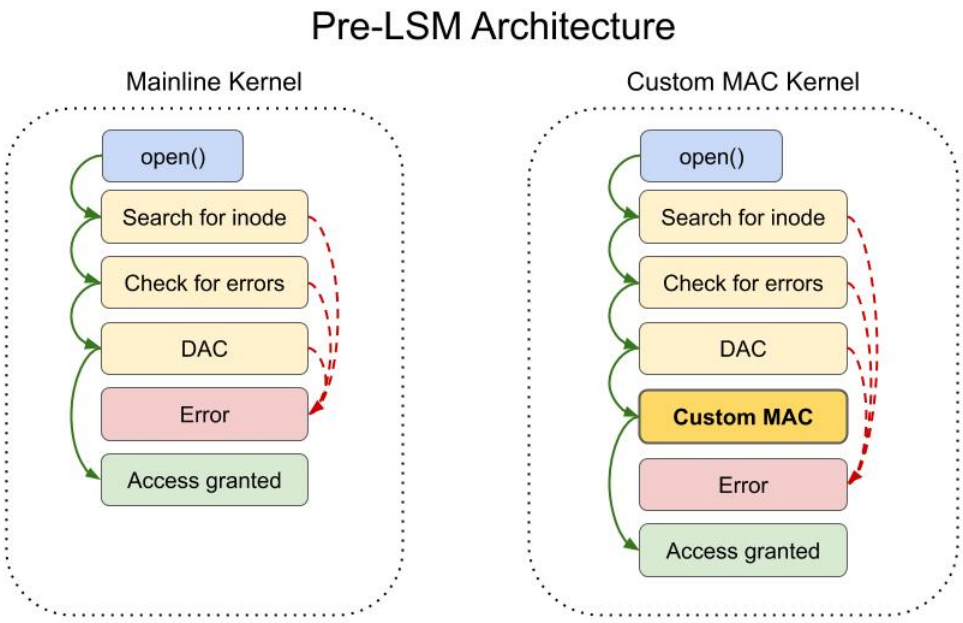

Figure 2. Pre-LSM Architecture that Required Custom Kernels for MAC

Linux kernel maintainers eventually created the LSM framework to provide a pathway for these custom security projects to be merged into the Linux mainline kernel (Smalley, n.d.). Multiple LSMs were eventually merged, but only one of them could be enabled at a time.

The two biggest players among Linux MAC extensions were SELinux for Red Hat-based distributions and AppArmor for Debian-based distributions (Smalley, 2002 June; Beattie, 2017). SELinux was known for type enforcement, which enforced how certain types of subjects could interact with certain types of objects. AppArmor took an alternate approach of basing its policies on filesystem paths.

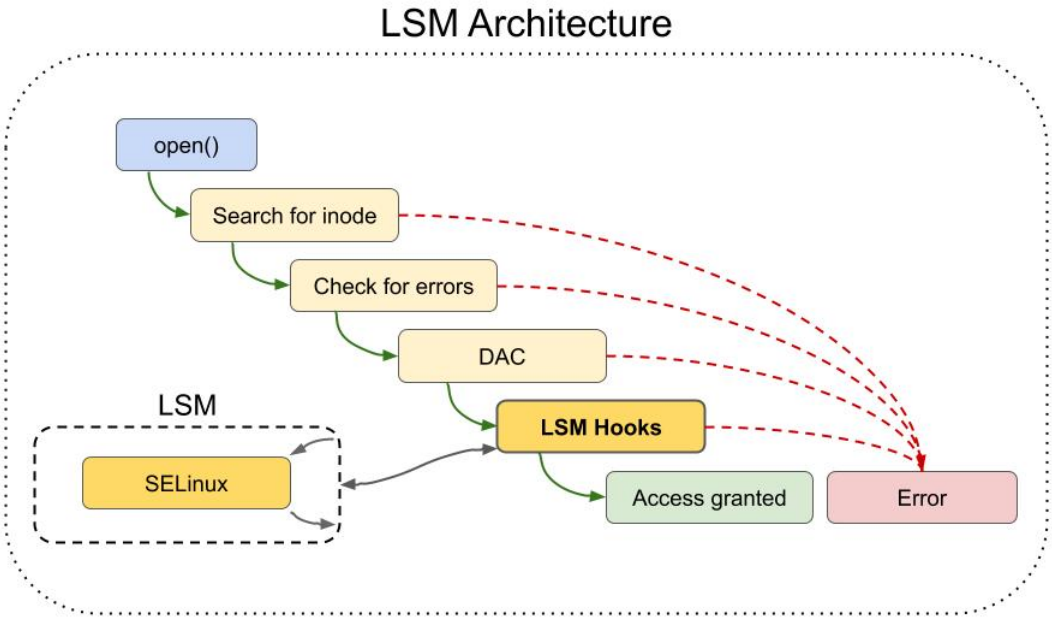

Figure 3. Linux Security Modules Architecture

There were several lesser-known MAC extensions as well, including Smack (Cook, 2017), TOMOYO (Takeda, 2009), and Yama (Cook, 2010). Because only one LSM could be enabled at a time, these smaller LSMs were often crowded out. However, starting in 2015, multiple LSMs could be loaded at the same time (Edge, 2015).

LSM hooks are still the common interface used by MAC extensions. They are listed in the Linux kernel source code file "/include/linux/lsm_hook_defs.h." The following are a few example entries:

```
LSM_HOOK(int, 0, inode_permission, struct inode *inode, int mask)
LSM_HOOK(int, 0, bprm_check_security, struct linux_binprm *bprm)
LSM_HOOK(int, 0, socket_listen, struct socket *sock, int backlog)
```

Figure 4. Excerpt of /include/linux/lsm_hook_defs.h

The LSM_HOOK() macro specifies the function return type, the default return value, the name of the security hook, and the list of arguments passed into the hook.

The first line of the excerpt above is for a hook named inode_permission. Its arguments are an inode structure (which contains the metadata for a file) and an integer

that represents the permission mask. When an LSM hook is triggered, control is passed to one or more LSMs which examine the arguments passed to the hook and then allow or deny access.

Another Linux kernel source file supplements information about these hooks. The following is an excerpt from "/include/linux/lsm_hooks.h" about the inode_permission hook, slightly modified for readability:

```
* @inode_permission:
*  Check permission before accessing an inode.
...
*  @inode contains the inode structure to check.
*  @mask contains the permission mask.
*  Return 0 if permission is granted.
```

Figure 5. Excerpt of /include/linux/lsm_hooks.h

This entry documents that the inode_permission hook can be used for additional permission checks before allowing access to an inode. Security practitioners can reference these two source files to understand the purpose and usage of every LSM hook in the Linux kernel. They can be viewed online with the Elixir Cross Referencer at https://elixir.bootlin.com.

3. Kernel Runtime Security Instrumentation

In September 2019, KP Singh proposed a set of kernel patches to the Linux Kernel Mailing List for a new LSM called "Kernel Runtime Security Instrumentation," or KRSI (Singh, 2019 September). It allowed an administrator to attach BPF programs to the various LSM hooks, and it could also inject an error to block the operation in question. This gave administrators the ability to define their own MAC policies with arbitrary code.

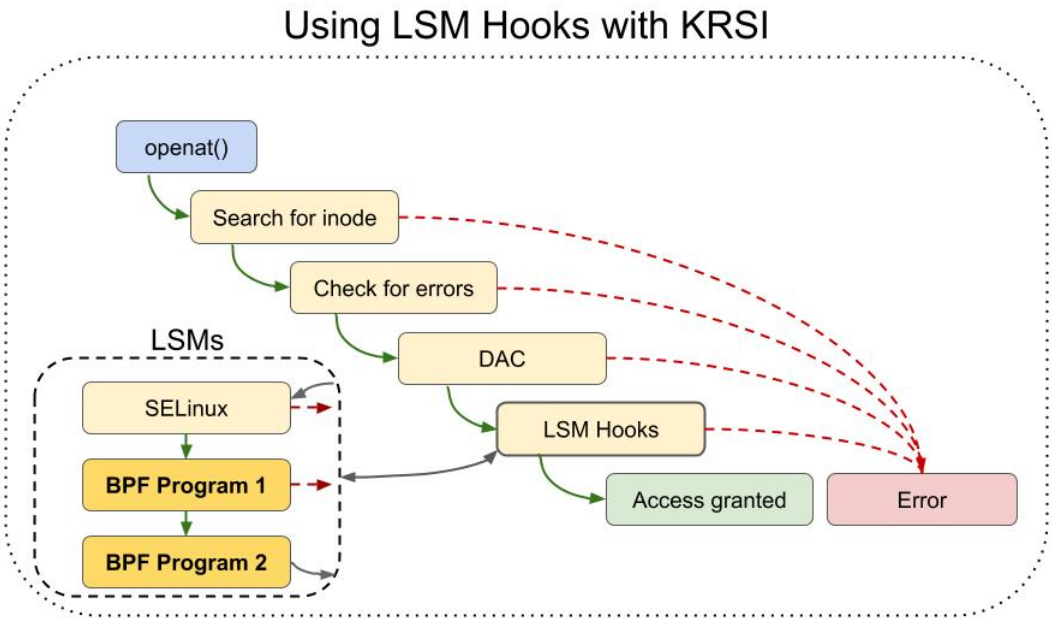

Figure 6. Using LSM Hooks with KRSI

The patches went through several revisions and were merged into the mainline Linux kernel in March 2020 (Borkmann, 2020). Two months later, Linux kernel version 5.7 was the first to include KRSI, released on 31 May 2020 (Corbet, 2020 June).

KRSI programs are being used in production at Google for mitigating various attacks, including LD_PRELOAD attacks (Singh, 2019 December).

Unfortunately, working examples of using KRSI in the mainline kernel are almost non-existent. The examples included in the KRSI patch set depended on special helper functions that were never merged into mainline. An open-source tool called Hawk demonstrated KRSI usage, but it only monitored process execution (Singh, 2020 September).

3.1. Tools for Writing KRSI Programs

There are three main toolsets for writing BPF programs: bpftrace, the BPF Compiler Collection (BCC), and the BPF-related tools provided in the Linux kernel source code.

The most approachable of the three tools is bpftrace, but it does not support KRSI yet. There is a pull request for this feature, but it has not been merged yet due to its

dependency on a missing kernel helper function (Olsa, 2020). Once the pull request is merged, writing KRSI programs will become much simpler. The following is a basic example of using bpftrace to load a KRSI program that prevents any other BPF programs from being loaded:

```
bpftrace -e 'lsm:bpf { return -1234; }'
```

Figure 7. Example of using KRSI with bpftrace

This program attaches to the "bpf" LSM hook, which performs the initial check for all bpf() syscalls. It overrides the return value with a non-zero integer, causing all future attempts to call bpf() to fail until the program is unloaded. In other words, this BPF program blocks other BPF programs from loading.

The second tool of choice is BCC. Fortunately, it has supported KRSI since its 0.15.0 release in June 2020 (Song, 2020). As such, this research will rely on BCC.

Writing BCC scripts is significantly more involved than writing bpftrace scripts. There are two halves to each BCC script; the first half is the BPF program that will be loaded into kernel-space. This portion is written in C. The second half is the user-space script that will load the BPF program, poll data from it, and facilitate various command-line options. This portion is written in Python. The Python script can either embed the C program within it or reference it as a separate file. A full tutorial for writing a basic BCC script can be found in Appendix A.

4. Analysis of KRSI in HPC

The remainder of this paper is dedicated to measuring the ability of KRSI programs to mitigate attacks in an HPC environment. A test environment was built to compare the functionality and performance of KRSI-disabled and KRSI-enabled systems.

4.1. KRSI Scripts

Five BCC scripts were written that used KRSI for mandatory access control. Their source code is available in a public GitHub repo (Wilson, 2020 October). The scripts were written to address the following questions:

- Can KRSI block and report users who create files with insecure permissions?
- Can it block and report users who run unauthorized executables?
- Can it block and report users who establish SSH proxies?
- Can it block and report users who pivot to unauthorized network segments?
- Can it block and report unauthorized attempts to terminate processes?

All the scripts logged event data regarding the processes that caused them to fire. This data included the timestamp, command name, UID, GID, and PID of the processes, as well as any actions taken (allow or deny). Each script also recorded additional data that was unique to the type of event that it handled. The scripts wrote logs in a key-value format, but for the following sections, the logs were adjusted to a header-column format with a truncated timestamp for readability.

4.1.1. mac_fileperms

The mac_fileperms script was written to restrict users from setting the SUID (Set UID) and WOTH (Writable by Others) permission bits of files.

The SUID and WOTH bits are legitimate components of the Linux permissions model that are known to be abused by attackers. Setting the SUID bit on a file will allow a user to execute it with the file owner's privileges. It is necessary for executables like "passwd" and "sudo," but malicious programs have also set the SUID bit on files to maintain persistent backdoors (MITRE, 2020 August). The WOTH bit allows anyone to write to a given file. It is often set incorrectly by users to ensure an application works (MITRE, 2020 March) or to intentionally share data with peers. Malicious actors can find and abuse files that are writable by anyone.

The mac_fileperms script attached programs to the "inode_create" and "path_chmod" LSM hooks. These hooks were triggered when a new file was created or an existing file's permissions were modified, respectively.

The attached programs prevented SUID and WOTH permission bits from being set on new files or added to existing files. It did so by examining the requested permissions mode of a file and the umask of the process. If it detected a SUID or WOTH

bit request, then it could prevent the file from being created or the file permissions from being updated.

Figure 8 below shows an invocation of the mac_fileperms script and its resultant effect on the user "billy," who has UID 1000.

Invocation of mac_fileperms:

```
# ./mac_fileperms -D -u billy
TIMESTAMP TYPE  COMM  UID  GID  PID   OLDMOD REQMOD UMASK  NEWMOD ACT
T13:09:34 chmod chmod 1000 1000 18064 100664 -      -      004664 deny
T13:09:59 creat touch 1000 1000 18073 000000 100666 000000 100666 deny
T13:11:10 chmod chmod 1002 1002 18163 100664 -      -      004664 allow
T13:11:42 creat touch 1002 1002 18168 000000 100666 000000 100666 allow
```

User terminal:

```
billy@linux1 ~ $ touch /tmp/suidfile
billy@linux1 ~ $ chmod u+s /tmp/suidfile
chmod: changing permissions of '/tmp/suidfile': Operation not permitted
billy@linux1 ~ $ umask
0002
billy@linux1 ~ $ umask 0000
billy@linux1 ~ $ touch /tmp/writable-by-others
touch: setting times of '/tmp/writable-by-others': No such file or directory
billy@linux1 ~ $
```

Figure 8. Invocation and Effect of "mac_fileperms"

This script was invoked in Deny Mode, specifying that the user "billy" should be blocked from adding SUID or WOTH permission bits, whether through file creation or file modification. Other users were not restricted.

The user "billy" attempted to add SUID permission bits to a file, but chmod returned an error. The user then changed their shell's umask to include the WOTH bit for newly created files. When the user ran the touch command to create a file with the WOTH bit set, the command failed to create the file.

The script logged data from each event. The “chmod” type indicated a request to add SUID or WOTH permission bits to an existing file, and the “create” type indicated a request to set them on a new file. Both actions by the user “billy” were logged as denied.

These steps were then repeated with a different user. Their operations were examined and allowed, as evidenced by the two log entries for UID 1002.

4.1.2. mac_suidexec

The mac_suidexec script restricted SUID files from being executed. This contrasted with the previous script, which prevented their creation. Dozens of legitimate SUID executables exist on Linux systems, but centrally restricting SUID execution to a subset of files or users is desirable.

The script attached a program to the “bprm_check_security” LSM hook. This hook fired when a nascent process was being prepared for execution via the exec() family of syscalls. The hook exposed the linux_binprm structure, which contained information about the program being invoked (Drysdale). The invoked program’s mode was examined for the SUID bit which, if detected, would result in the script logging the attempt and would possibly prevent the execution from occurring.

The following is an example of invoking mac_suidexec and its effect on a user session:

Invocation of mac_suidexec:

```
 # ./mac_suidexec -D -u billy -F /bin/passwd
TIMESTAMP TYPE   COMM UID  GID  PID   DEV INODE    MODE   ACT
T14:48:15 exsuid bash 1000 1000 21307 42  43091310 104755 allow
T14:48:22 exsuid bash 1000 1000 21317 42  43091340 104111 deny
T14:48:30 exsuid bash 1000 1000 21325 42  43091580 104755 deny
```

User terminal:

```
billy@linux1 ~ $ passwd
Changing password for user billy.
Current password: ^C
billy@linux1 ~ $ sudo -i
-bash: /bin/sudo: Operation not permitted
billy@linux1 ~ $ newgrp
-bash: /bin/newgrp: Operation not permitted
```

Figure 9. Invocation and Effect of "mac_suidexec"

The above invocation of mac_suidexec prevented the user "billy" from invoking any SUID binaries on the system *except* for /bin/passwd (signified by the capital -F option). When the user attempted to run the "passwd" binary, it succeeded. However, attempts to run "sudo" and "newgrp" were blocked and logged by the script. The script logged the event type as "exsuid" and recorded the file's inode number, device number, and mode. The script tracked files by inode and device numbers rather than paths.

4.1.3. mac_sshlisteners

The mac_sshlisteners script prevented the creation of SSH proxy tunnels and handled both IPv4 and IPv6 addresses.

In "Securing the Soft Underbelly of a Supercomputer with BPF Probes," Wilson (2020 June) briefly explained TCP port forwarding. It is a built-in SSH feature that allows someone to use a server as a proxy to reach external resources. This feature can be used to transfer data to and from a device that lacks direct access to the internet. Although the feature can be disabled on SSH servers, it is on by default and usually left

that way (Wilson, 2020 June). Whenever an SSH client attempts to listen on a socket, it indicates that the client is attempting to proxy traffic through a server that it can reach.

Unlike the BPF tracing script used by Wilson, which only *detected* SSH proxy tunnels, the mac_sshlisteners script both detected and prevented the proxy tunnels. It attached a program to the "socket_listen" LSM hook, which fired whenever a process attempted to change a socket to a LISTEN state. It examined IPv4 and IPv6 sockets to ensure that an SSH client was not attempting to listen on it.

The following was an invocation of the mac_sshlisteners script and its effect on a user who attempted to open an SSH proxy tunnel. Before invoking the script, the author slightly modified it to return the -NOLINK error upon denial instead of the -EPERM error. This was done to demonstrate that these scripts can return arbitrary error values to the user.

Invocation of mac_sshlisteners:

```
# ./mac_sshlisteners -D -U root
TIMESTAMP TYPE   COMM UID  GID  PID  PROTO LADDR     LPORT ACTION
T08:53:39 listen ssh  1000 1000 6602 6     [::1]     9999  deny
T08:53:39 listen ssh  1000 1000 6602 6     127.0.0.1 9999  deny
```

User terminal:

```
billy@linux1 ~ $ ssh -D 9999 10.7.7.6
listen: Link has been severed
listen [::1]:9999: Link has been severed
listen: Link has been severed
listen [127.0.0.1]:9999: Link has been severed
channel_setup_fwd_listener_tcpip: cannot listen to port: 9999
Could not request local forwarding.
Last login: Thu Oct  8 16:49:11 2020 from 192.168.100.15
billy@login1
```

Figure 10. Invocation and Effect of "mac_sshlisteners"

The script was invoked in Deny Mode with a capital "-U" to deny all users except root from opening proxy tunnels with SSH clients. The user then attempted to open an

SSH proxy tunnel with dynamic port forwarding (signified by the SSH client's -D option). When the client attempted to change an IPv6 socket to a listening state on local port 9999, it was given the error that a link was severed. It then attempted another listen operation with an IPv4 socket, which failed in the same manner. The SSH connection to the remote host still succeeded for the user, but the proxy tunnel was not successfully established. The script logged both the IPv4 and the IPv6 attempts to change a socket to a listening state.

4.1.4. mac_skconnections

The mac_skconnections script restricted socket connections to certain destinations. It had full visibility into any socket connection attempt, but it was written to only examine IPv4 socket connections for this research. This script protected against pivot attempts by adding per-user firewall restrictions, and it did so without modifying the host's central firewall configuration.

The script was very similar to a BPF tracing script used by Wilson (2020 June) called tcp_connectfilter.sh. This script only handled TCP connections via the tcp_connect() kernel function, and it was limited to detection. In contrast, the mac_skconnections script handled any IPv4 socket connections regardless of the underlying protocol, and it provided both detection and mitigation.

The mac_skconnections script attached a program to the "socket_connect" LSM hook, which fired when a process attempted to make a socket connection. The following was an example of invoking the mac_skconnections script and how it affected a user.

Invocation of mac_skconnections:

```
# ./mac_skconnections -D 10.100.0.0 -m 255.255.0.0 -u billy
TIMESTAMP TYPE   COMM UID  GID  PID   PROTO DADDR           DPORT ACT
T10:45:00 skconn ssh  1000 1000 10109 6     10.100.3.4      22    deny
T10:45:11 skconn ssh  1000 1000 10160 6     10.101.3.4      22    allow
T10:50:02 skconn dnf  0    0    10275 17    10.102.5.6      53    allow
T10:50:02 skconn dnf  0    0    10275 6     209.132.183.108 443   allow
T10:51:57 skconn nc   1002 1002 10357 6     10.100.10.11    80    allow
T10:57:20 skconn nc   1000 1000 10644 17    10.102.5.6      53    deny
```

User terminal:

```
billy@linux1 ~ $ ssh 10.100.3.4
ssh: connect to host 10.100.3.4 port 22: Operation not permitted
billy@linux1 ~ $ ssh 10.101.3.4
Last login: Fri Oct  9 16:46:37 2020 from 192.168.10.15
billy@login3 ~ $ ^C
billy@linux1 ~ $ nc -u 10.100.4.5 53
Ncat: Operation not permitted.
```

Figure 11. Invocation and Effect of “mac_skconnections”

The script was invoked to block the user “billy” from making IPv4 socket connections to the 10.100.0.0/16 subnet. When the user attempted to SSH to a destination in that subnet, the operation was not permitted. Trying another destination outside the restricted subnet was successful. The script then logged some socket connections by the server’s package manager, Dandified YUM (dnf). These connections were allowed. The user tried a final netcat to 10.100.4.5 on port 53, which was blocked by the script.

4.1.5. mac_killtasks

The mac_killtasks script restricted process signals. It was used to protect all the BCC scripts from early termination, whether by unprivileged users or root. It handled only SIGKILL and SIGTERM signals, but it could be extended to handle any signal.

Sending signals is a fundamental part of Linux and other POSIX operating systems. Signals can inform processes that an event has occurred (Kerrisk). They can also pause or terminate processes. The kernel generates them, but other system processes can

request that the kernel send signals on their behalf. Over thirty signals are available on Linux, but the two restricted by this script were SIGTERM (ask a process to terminate itself) and SIGKILL (kill the process immediately).

The following is an example of invoking mac_killtasks to protect an instance of mac_fileperms and its effect on the root user.

Invocation of mac_killtasks:

```
# ./mac_fileperms -A -u root &
[2] 18290
# mac_fileperms_pid=$!
# ./mac_killtasks -D -e -t $mac_fileperms_pid
TIMESTAMP TYPE   COMM UID GID PID   TARGETUID TARGETPID SIGNO ACT
T14:36:13 sgkill bash 0   0   11342 0         18290     15    deny
T14:36:17 sgkill bash 0   0   11342 0         18290     9     deny
T14:36:32 sgkill bash 0   0   11342 0         18316     9     deny
T14:36:44 sgkill bash 0   0   11342 0         18399     15    allow
```

A root terminal:

```
# pgrep -fl mac_
18290 mac_fileperms
18316 mac_killtasks
# kill 18290
-bash: kill: (18290) - Operation not permitted
# kill -9 18290
-bash: kill: (18290) - Operation not permitted
# kill -9 18316
-bash: kill: (18316) - Operation not permitted
# sleep 1000 &
[1] 18399
# kill 18399
```

Figure 12. Invocation and Effect of “mac_killtasks”

The script hooked into the “task_kill” LSM hook, which fired when a signal was about to be sent to a process. It examined the attributes of the source process and the target process to determine whether the signal should be allowed or not.

First, mac_fileperms was invoked to allow only the root user to create files with SUID or WOTH bits. The process ID of that script was saved. Then mac_killtasks was invoked so that no processes could send kill signals to the mac_fileperms process or to the mac_killtasks process that protected it.

In another terminal, the root user looked up the PIDs of the two scripts and attempted to send a SIGTERM signal to mac_fileperms. This attempt failed. The root user then unsuccessfully attempted to send SIGKILL signals to both BCC scripts. These also failed. Finally, root spawned a sleep process and sent it a SIGTERM signal. This succeeded and all activities were logged.

The script provided two new options: --kernel (-k) and --eternal (-e). The --kernel option modified the attached program to also control signals originating from the kernel itself in addition to those requested by user-space processes. This author did not test the implications of blocking signals from kernel-space processes and therefore marked this option as "dangerous" in the script's help text. The --eternal option ensured that the mac_killtasks executable itself was unkillable by any process except for its parent process. If the parent process exited, nothing could kill the process.

4.2. Test Environment

Two Linux kernels were compiled that differed only in whether KRSI was enabled or not. Version 5.8.10 of the source code was used, which was the most recent stable version available as of 21 September 2020.

The Linux kernels were configured as similarly as possible to the kernel that is included in Red Hat Enterprise Linux distributions. This was not done manually, as the 5.8.10 kernel configuration file contains nearly 7,000 lines of options. Rather, an RPM package for the 5.8.10 kernel version was downloaded from ELRepo.org, and the "config" file was extracted from it. Running `make oldconfig` with this configuration file returned no output, confirming that all options for the 5.8.10 kernel were defined.

The non-KRSI and KRSI kernels needed to be easily differentiated. One of the copies of the kernel source code was configured to append the string "-non-krsi" to its

version. The other was configured to append the string "-krsi." This allowed for quick identification of the kernel in use by running `uname -r`.

KRSI was enabled in the latter kernel with the following configuration changes:

```
CONFIG_BPF_LSM=y
CONFIG_LSM="yama,loadpin,safesetid,integrity,selinux,smack,tomoyo,apparmor,bpf"
CONFIG_DEBUG_INFO_BTF=y
```

Figure 13. Linux Kernel 5.8.10 Configurations to Enable KRSI

The first line enabled KRSI instrumentation. The second line provided a list of LSMs to initialize, with KRSI represented by the word "bpf" at the end of the string. The entries in the list besides "bpf" were included in the default configuration. The third line ensured that the kernel was compiled with BPF Type Format (BTF) symbols. Many BPF tools now depend on including BTF symbols in the kernel; the symbols help the in-kernel BPF Verifier perform memory access safety checks on the program before it is loaded. The pahole binary, part of the "dwarves" package, also needed to be installed to build the Linux kernel with BTF symbols.

The servers were likewise configured to be as similar as possible to each other. Eight compute nodes from an HPC cluster were reserved for testing. Kernel selection was handled with PXE boot. Each compute node mounted the same read-only root filesystem from a central NFS server.

4.3. Low-Profile Attack Script

A bash script simulated the activity that the BCC scripts were written to detect and mitigate. Every one-to-fifteen seconds, the script randomly performed one of the following actions:

- Created a new other-writable file
- Added other-writable permissions to an existing file
- Added the SUID bit to an existing file
- Ran a SUID executable file
- Attempted to open an SSH proxy connection

- Attempted to connect to a remote destination over TCP
- Attempted to connect to a remote destination over UDP
- Attempted to terminate the KRSI scripts (as root)

This script logged all actions taken, which allowed for comparison between the logs of the attack script and the logs of the detection scripts. The script is included in Appendix B.

4.4. Benchmarks

High-performance computing applications vary immensely in how they exercise a system. They can cause performance bottlenecks in processing, memory operations, filesystem operations, network communications, and more.

For this reason, three benchmarks were chosen for this research: xhpl for computational benchmarking, a C program named mdstress for IO benchmarking, and tcpkali for network benchmarking. These benchmarks are described below.

4.4.1. Computational Benchmark: xhpl

The High-Performance Linpack Benchmark, or xhpl, measures the computational performance of the largest supercomputers in the world. It solves a series of linear algebra equations to measure the maximum "flops," or floating-point operations per second, that a cluster is capable of. It can measure the flops of a single compute node or multiple compute nodes working in unison.

The xhpl benchmark was run on the eight test nodes individually. For each node, it ran thirty times on the non-KRSI kernel, thirty times on the KRSI kernel without the BCC scripts loaded, and thirty times with all five BCC scripts loaded.

4.4.2. Filesystem Benchmark: mdstress

The mdstress benchmark is a rudimentary C program written by this author. It created a new file, wrote the string "mdstress" to it, and deleted the file in a tight loop. When all loops completed, it printed the total elapsed time in seconds. The purpose of this benchmark was to determine how mac_fileperms behaved under extreme load. The source code of the program can be found in Appendix C.

Each mdstress benchmark was configured to complete in 10 seconds at optimal performance. Any overhead would cause its completion to exceed 10 seconds. For example, when creating 100,000 inodes, the stresser loop was rate limited to 10,000 loops per second, which at its fastest would complete in 10 seconds. Any slowdown beyond the rate limit would result in the benchmark taking longer than 10 seconds to complete. Each configuration was run thirty times on each node with no BCC scripts running, and the results were averaged. These tests were repeated with mac_fileperms running and then repeated one more time with the umask of the mdstress process set to 0000, which resulted in the WOTH bit being set on the inodes. File and directory caches were dropped before each run to minimize the effect of caching between runs.

4.4.3. Network Benchmark: tcpkali

The tcpkali benchmark can establish and tear down thousands and even millions of TCP connections in a short period of time. It is used to stress-test applications and networks. For this research, it helped identify the performance cost of attaching programs to the socket_connect LSM hook.

This benchmark used one compute node as a client. It used up to seven other computes nodes for destinations. The compute nodes were all tuned to allow up to 55,000 TCP connections per peer. These tunings can be found in Appendix D.

The client attempted 1,500 TCP connections per second per destination, scaling up to 10,500 TCP connections per second when all seven destinations were in use. Each run was given a thirty-second time limit, and the number of successful connections was recorded by the nodes acting as destinations. The destination nodes ran `socat` to listen for and record TCP connections. These tests were run without mac_skconnections loaded and with it loaded to compare performance.

5. Results

The benchmark results showed that the BCC scripts performed very favorably under intensive but realistic workloads. However, when mdstress and tcpkali pushed systems to very extreme levels, enough to degrade general system performance, then

running the BCC scripts worsened performance. Those extreme workloads also overloaded the "perf ring buffer," which was the buffer used by BPF to stream high volumes of kernel events to user-space. This did not impede the BCC scripts in preventing attacks, but it did affect their ability to log them.

5.1. Detection Results

During the xhpl benchmarks, the adversary emulation script and all the BCC scripts ran and logged their activities. These logs were aggregated to determine whether the actions by the adversarial script were successfully detected and mitigated.

All malicious file activities, including creating world-writable files and adding WOTH or SUID bits, were logged as "denied" except for one of the 2,964 "Add WOTH bit" actions. All attempts to run SUID executables were denied and logged. All attempts to open SSH proxies were logged as "denied" except for two of the 2,943 IPv4 attempts. All unauthorized socket connection attempts and all attempts to kill the BCC scripts were denied and logged.

It was difficult to determine the cause of the three missing logs due to a shortcoming of the attack script itself. The attack script logged what actions it would take, but it did not record whether the attempted action was executed or not. It was possible that a few malicious actions failed from regular system errors. This would have acted as a short-circuit prior to the LSM hook being triggered and the BCC script coming into play.

The results are summarized below in Figure 14.

Malicious Action	Count	KRSI Script	Deny Count	Deny Types	Type Count
Create WOTH	2997	mac_fileperms	8907	create	2997
Add WOTH	2964			chmod (0602)	2965
Add SUID	2946			chmod (4600)	2946
Run SUID	2989	mac_suidexec	2989	exsuid	2989
SSH Proxy	2945	mac_sshlisteners	5888	IPv4	2943
				IPv6	2945
Socket Connect	2918	mac_skconnect	2918	IPv4	2918
Kill BCC (fileperms)	586	killtasks (file)	1176	fileperms only	586
Kill BCC (suidexec)	602	killtasks (suid)	1192	suidexec only	602
Kill BCC (ssh…)	600	killtasks (ssh…)	1190	sshlisteners only	600
Kill BCC (skconnect)	621	killtasks (skc…)	1211	skconnect only	621
Kill BCC (killtasks)	590				

Figure 14. Table of Malicious Action Counts and MAC Denial Counts

While the xhpl detection results were very favorable, the tcpkali and mdstress benchmarks demonstrated that an excessive amount of MAC events could result in dropped logs. These details will be discussed as part of the performance results.

5.2. Performance Results

Three tools measured the performance impact of KRSI: xhpl, mdstress, and tcpkali. While xhpl was configured to use a large but reasonable portion of each system's computational resources, mdstress and tcpkali scaled IO and network loads past reasonable system capabilities. Stress testing IO and network resources in this manner revealed the behavior of KRSI on systems suffering from extreme loads.

This section presents the benchmark results as charts; the numeric tables used to generate these charts can be found in Appendix E.

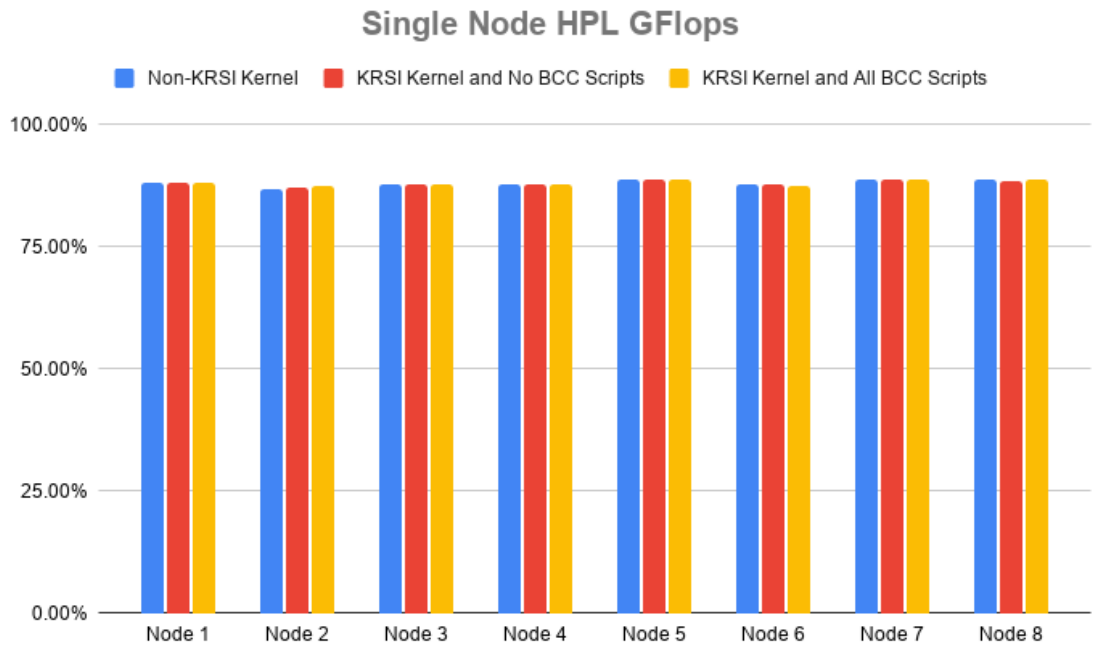

Figure 15. Chart of "xhpl" Results

KRSI had a less than 1% computational impact on xhpl benchmarks on the compute nodes. The worst case was Node 6, which exhibited a 0.14% performance loss when using a KRSI-enabled kernel with all BCC scripts loaded. Oddly, Node 2 showed a 0.51% gain in performance with BCC scripts loaded, but this would not be explained by KRSI. It was likely due to other factors in the test environment, such as network contention from unrelated compute nodes or system jitter. The rest of the nodes showed a performance change of 0.05% or less between the non-KRSI kernel and the KRSI kernel with BCC scripts running.

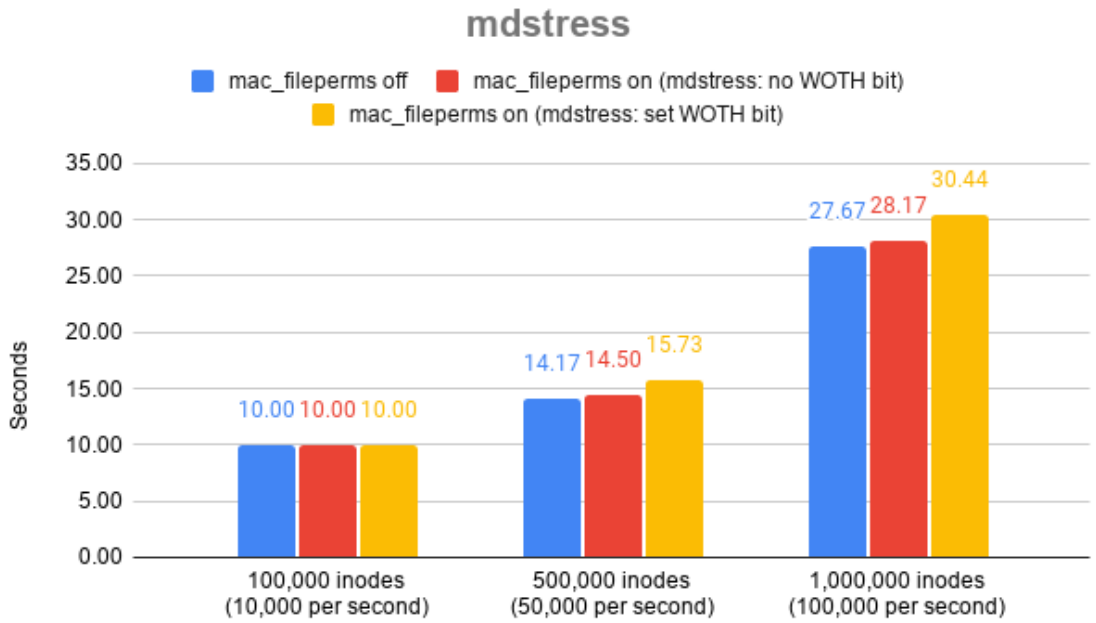

Figure 16. Chart of "mdstress" Results

The mdstress benchmark revealed two issues when under extreme filesystem load. First, if a compute node was stressed with more file creation operations than it could handle, then running BCC scripts that monitored file creation would make the problem worse. Second, when events were generated by the kernel-space program too quickly, the perf ring buffer would overwrite the oldest event data before user-space scripts logged it. This was indicated by messages from the BCC script stating, "Possibly lost N samples," with N ranging from 1 to over a million, depending on the load that mdstress placed on the script.

When handling 10,000 inode creation events per second, the mac_fileperms script did not cause performance loss issues, but it did drop logs due to the overwhelmed ring buffer.

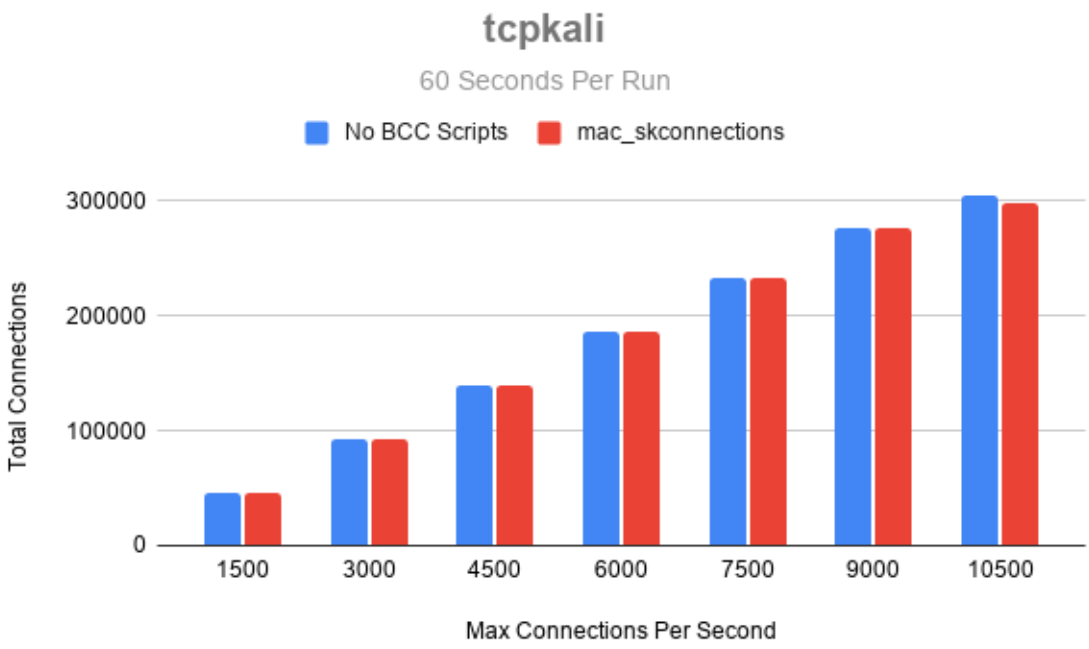

Figure 17. Chart of "tcpkali" Results

The tcpkali benchmark stressed the system beyond its ability to maintain its connection rate when set to 9,000 TCP connections per second. This performance degradation became worse at 10,500 TCP connections per second. The mac_skconnections script began to lose logs from the perf_ring_buffer at this point. In both cases, attaching mac_skconnections caused the system to perform slightly worse.

6. Conclusions and Future Work

Running BCC scripts with KRSI did not cause performance loss unless the system was already suffering from degraded performance. It did not impact xhpl benchmarks. It only impacted mdstress and tcpkali benchmarks after extreme loads were already causing general system performance problems. Therefore, companies and organizations should continue investigating KRSI for adoption in HPC and information security.

The BCC scripts appeared to still mitigate attacks even under extreme load, but they did not guarantee zero loss of logs. When the filesystem event rate approached the order of 100,000 file creations per second, the mac_fileperms script dropped millions of

incoming logs. This is not a shortcoming of KRSI, but rather is the expected consequence of overloading a ring buffer. It is possible to increase the perf ring buffer's size, but the default is already 64 pages per CPU (Drayton). Any load that overwhelms the perf ring buffer has already pushed the system well beyond reasonable limits.

KRSI will become much more approachable once bpftrace supports it. This tool provides an AWK-like syntax that is much more intuitive to systems administrators and information security practitioners than the complex BCC scripts written for this research.

Those interested in KRSI can now experiment with it using the stock kernel of a major Linux distribution. Canonical released the 20.10 version of Ubuntu on 22 October 2020, and it runs on the 5.8 Linux kernel. Users can enable KRSI by specifying "bpf" in the "lsm" kernel parameter (e.g., "lsm=lockdown,yama,integrity,apparmor,bpf").

This author used a kernel configuration baseline that had many LSMs initialized, which may have drowned out the true performance impact of enabling KRSI. Future research can measure the performance of systems with either no LSMs initialized or only KRSI initialized.

The scripts written for this research used only six LSM hooks, but there are 233 LSM hooks available in the 5.8.10 Linux kernel. It would be valuable for those with kernel development and information security experience to write additional BCC scripts that leverage these LSM hooks in interesting new ways.

Once KRSI is generally available, HPC environments can potentially couple it with their schedulers to launch custom MAC policies per compute node, based on the users' jobs dispatched to those nodes. This would result in special security measures that follow users to whichever nodes their research is running on.

The ultimate benefit of KRSI is the freedom it gives systems administrators to build creative new MAC policies. Those who work on Linux systems with recent kernels are encouraged to try KRSI, be creative in its usage, and share their innovations with the community.

Billy Wilson, billy_wilson@byu.edu

References

Barkley, J. (1994). Discretionary Access Control. *NIST Special Publication 800-7*. Retrieved 26 September 2020 from http://ftp.gnome.org/mirror/archive/ftp.sunet.se/pub/security/docs/nistpubs/800-7/main.html

Beattie, S. (2017). About AppArmor. *AppArmor Security Project Wiki*. Retrieved 26 September 2020 from https://gitlab.com/apparmor/apparmor/-/wikis/About

Borkmann, D. (2020). Merge branch 'bpf-lsm' [Computer Software]. *Kernel.org git repositories.* Retrieved 26 September 2020 from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=641cd7b06c911c5935c34f24850ea18690649917

Cook, K. (2010). security: Yama LSM [Computer Software]. *LWN*. Retrieved 26 September 2020 from https://lwn.net/Articles/393012/

Cook, K. (2017). Smack. *The Linux kernel user's and administrator's guide.* Retrieved 26 September 2020 from https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/Smack.html

Corbet, J. (2019, October). BPF at Facebook (and beyond). *LWN.* Retrieved 26 September 2020 from https://lwn.net/Articles/801871/

Corbet, J. (2019, December). KRSI – the other BPF security module. *LWN.* Retrieved 26 September 2020 from https://lwn.net/Articles/808048/

Corbet, J. (2020, June). The 5.7 kernel is out. *LWN.* Retrieved 26 September 2020 from https://lwn.net/Articles/821829/

Billy Wilson, billy_wilson@byu.edu

Drayton, M. (2017, February). Make perf ring buffer size configurable [Computer software]. *Github*. Retrieved 21 October 2020 from https://github.com/iovisor/bcc/pull/997

Drysdale, D. (2015). How programs get run. *LWN.* Retrieved 17 October 2020 from https://lwn.net/Articles/630727/

Edge, J. (2015). Progress in security module stacking. *LWN.* Retrieved 26 September 2020 from https://lwn.net/Articles/635771/

Gregg, B., et al. (2016). Bcc Python Developer Tutorial. *Github*. Retrieved 1 October 2020 from https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md

Gregg, B. (2018, September). gethostlatency.bt [Computer software]. *Github.* Retrieved 26 September 2020 from https://github.com/iovisor/bpftrace/blob/master/tools/gethostlatency.bt

Gregg, B. (2020, January). *BPF Performance Tools: Linux System and Application Observability*. United States: Addison-Wesley.

Gregg, B., et al. (2020, August). bcc Reference Guide. *Github*. Retrieved 26 September 2020 from https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md

Gregg, B., et al. (2020, September). bpftrace Reference Guide. *GitHub.* Retrieved 26 September 2020 from https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md

Gregg, B., & Maestretti, A. (2017, February). Security Monitoring with eBPF. In *BSidesSF 2017*, San Francisco, CA. Retrieved 26 September 2020 from https://www.youtube.com/watch?v=44nV6Mj11uw

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX system programming handbook.* San Francisco: No Starch Press.

Larabel, M. (2020). The Performance Cost to SELinux on Fedora 31. *Phoronix*. Retrieved 14 November 2020 from https://www.phoronix.com/vr.php?view=28798

National Institute of Standards and Technology. (2016). An Action Plan for High Performance Computing Security, Working Draft. Gaithersburg, MD. Retrieved 26 September 2020 from https://www.nist.gov/system/files/documents/2018/03/15/working_draft_actionplanhpc.pdf

Olsa, J. (2020). Add lsm probe support [Compute Software]. Retrieved 26 September 2020 from https://github.com/iovisor/bpftrace/pull/1347

Singh, K. (2019, September). Kernel Runtime Security Instrumentation. *LWN*. Retrieved 26 September 2020 from https://lwn.net/Articles/798918/

Singh, K. (2019, December). lsm_audit_env.c [Computer Software]. *Github*. Retrieved 1 October 2020 from https://github.com/sinkap/linux-krsi/blob/patch/v1/examples/samples/bpf/lsm_audit_env.c

Singh, K. (2020, March). MAC and Audit Policy using eBPF (KRSI). *LWN.* Retrieved 26 September 2020 from https://lwn.net/Articles/815826/

Singh, K. (2020, July). KRSI (BPF + LSM) – Updates and Progress. In *Linux Security Summit North America*, Virtual Conference. Retrieved 26 September 2020 from https://ossna2020.sched.com/event/ckpL/krsi-bpf-lsm-updates-and-progress-kp-singh-google

Singh, K., et al. (2020, September). hawk [Computer Software]. *Github*. Retrieved 17 October 2020 from https://github.com/googleinterns/hawk

Smalley, S., et al. (n.d.). Linux Security Modules: General Security Hooks for Linux. *The Linux Kernel Archives.* Retrieved 26 September 2020 from https://www.kernel.org/doc/html/latest/security/lsm.html

Smalley, S., et al. (2002, June). Linux Security Module Framework. In *Ottawa Linux Symposium*, Ottawa, Ontario, Canada. Retrieved 26 September 2020 from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.1604&rep=rep1&type=pdf

Smalley, S., et al. (2002, May). Implementing SELinux as a Linux Security Module. Retrieved 26 September 2020 from https://www.nsa.gov/Portals/70/images/resources/everyone/digital-media-center/publications/research-papers/implementing-selinux-as-linux-security-module-report.pdf

Song, Y. (2020). prepare for release v0.15.0 [Computer Software]. *GitHub.* Retrieved 26 September 2020 from https://github.com/iovisor/bcc/commit/e41f7a3be5c8114ef6a0990e50c2fbabea0e928e#diff-45c17c0a080fbe29e2b8ded8940aa1e8

Takeda, K. (2009). TOMOYO Linux Overview. Presented at *linux conf au*, Hobart, Australia. Retrieved 26 September 2020 from https://osdn.net/projects/tomoyo/docs/lca2009-takeda.pdf

MITRE. (2020, March). Keydnap. *MITRE ATT&CK*. Retrieved 17 October 2020 from https://attack.mitre.org/software/S0276/

MITRE. (2020, August) CWE-732: Incorrect Permission Assignment for Critical Resource. *Common Weakness Enumeration.* Retrieved 17 October 2020 from https://cwe.mitre.org/data/definitions/732.html

Wilson, B. (2020, June). Securing the Soft Underbelly of a Supercomputer with BPF Probes. *SANS Institute.* Retrieved 26 September 2020 from https://www.sans.org/reading-room/whitepapers/linux/securing-soft-underbelly-supercomputer-bpf-probes-39635

Wilson, B. (2020, October). bcc-lsm-scripts [Computer Software]. *Github*. Retrieved 17 October 2020 from https://github.com/wilsonwr/bcc-lsm-scripts

Appendix A: Tutorial of Using KRSI with BCC

A.1 Kernel-Space C Program

The BPF program in C is covered first. Its role is to examine the operation that caused the LSM hook to fire, send event data to user-space, and determine whether to allow or deny the operation that caused the LSM hook to fire.

First, the correct header files need to be included to use the appropriate structures and variables.

```
#include <linux/fs.h>
#include <linux/errno.h>
```

Figure A-1. Headers included in example BPF program

Then a data structure is defined that will store some context from each event. In this case, it will store the UID, PID, timestamp, inode number, and command name associated with the event. Then the BPF_PERF_OUTPUT() macro is called, which will set up the necessary components, including a structure named "events," for the program to send data to user-space.

```
struct data_t {
  u32 uid;
  u32 pid;
  u64 ts;
  unsigned long inode_num;
  char comm[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(events);
```

Figure A-2. Data Structure and Buffer to Pass Events in Example BPF Program

Next is the LSM_PROBE macro, which specifies which LSM hook this program will be attached to. The correct hook name and arguments are determined by looking at the corresponding LSM hook entry in "/include/linux/lsm_hook_defs.h", which was

discussed earlier. This program will attach to the inode_permission LSM hook, and it will be able to examine the inode structure and the permission mask.

```
LSM_PROBE(inode_permission, struct inode *inode, int mask) {
```

Figure A-3. Specifying the LSM Hook to Use in Example BPF Program

Next is the code that runs when this LSM hook fires. It initializes a structure for event data and then invokes the BPF helper function bpf_get_current_uid_gid(), storing only the lower bits that contain the UID. This is one of many BPF helper functions that the kernel makes available to BPF programs like this one (Gregg, August 2020).

Following this is an odd string, UID_FILTER, which is not C. It is a placeholder that will be used by Python to substitute logic into the program, depending on options passed to the Python script.

```
struct data_t data = {};
u32 uid = bpf_get_current_uid_gid();
UID_FILTER
```

Figure A-4. Setting up the UID Filter in Example BPF Program

The program then prepares to ship data to user-space by querying and storing information about the current UID, PID, timestamp, inode number, and command. It has access to the inode structure and any other arguments that are a part of the LSM hook.

```
data.uid = uid;
data.pid = bpf_get_current_pid_tgid();
data.ts = bpf_ktime_get_ns();
data.inode_num = inode->i_ino;
bpf_get_current_comm(&data.comm, sizeof(data.comm));
```

Figure A-5. Collecting Event Data in Example BPF Program

The data is placed in an “events” buffer that user-space can access. The method is called “perf_submit” because it depends on a special buffer called a “perf ring buffer,”

which is commonly used to gather Linux performance metrics from the kernel. Finally, the program returns a permission error, resulting in the process being denied access to the inode.

```
events.perf_submit(ctx, &data, sizeof(data));

return -EPERM;
```

Figure A-6. Event Data Submission and Access Denial in Example BPF Program

A.2 User-Space Python Script

The Python portion of the program is what the user executes. Only key portions of the Python script are covered here; the full program can be reviewed at the end of this appendix.

The C program outlined earlier is embedded into the Python script, being assigned to the variable "bpf_text" as a multi-line string.

```
bpf_text = """
#include <linux/fs.h>
#include <linux/errno.h>
...
```

Figure A-7. Embedding the BPF Program in Example BCC Script

The Python script directly modifies the C program before it is compiled and loaded. In the snippet below, Python replaces the "UID_FILTER" string in the C program with a branching statement that will allow inode access for all UIDs *except* the one passed to the Python script as an argument.

```
bpf_text = bpf_text.replace('UID_FILTER',
    'if (uid != %s) { return 0; }' % args.uid)
```

Figure A-8. Substituting Filters in Example BCC Script

The C program is then compiled and loaded into the kernel by invoking the BPF() function that is part of the "bcc" Python library.

```
b = BPF(text=bpf_text)
```

Figure A-9. Compiling and Loading BPF Program in Example BCC Script

The Python script can then access the event data by opening the "events" buffer that was defined in the C program.

```
b["events"].open_perf_buffer(print_event)
while 1:
  try:
    b.perf_buffer_poll();
  except KeyboardInterrupt:
    exit()
```

Figure A-10. Polling for Events in Example BCC Script

The following is an example of the script in Appendix A being invoked to deny inode access from UID 1234. When the user attempts to SSH to the server where this program is running, the output is as follows (modified for readability):

```
# ./mac_deny_inode_access.py -u 1000
TIME(s)     COMM      UID   PID   INODE     MESSAGE
0.000000000 (systemd) 1000  5330  42508571  Deny: inode_permission
hook
0.000038826 (systemd) 1000  5330  42508571  Deny: inode_permission
hook
0.000046959 (systemd) 1000  5330  42508571  Deny: inode_permission
hook
0.023633858 sshd      1000  5338  42508571  Deny: inode_permission
hook
0.023823824 sshd      1000  5338  42508571  Deny: inode_permission
hook
0.023833802 sshd      1000  5338  42508571  Deny: inode_permission
hook
```

Figure A-11. Output of Example BCC Script

A.3 Full Example BCC Program

```
#!/usr/bin/python

from bcc import BPF
from bcc.utils import printb
import argparse
import pwd

def parse_uid(user):
    try:
        result = int(user)
    except ValueError:
        try:
            user_info = pwd.getpwnam(user)
        except KeyError:
            raise argparse.ArgumentTypeError(
                "{0!r} is not valid UID or user entry".format(user))
        else:
            return user_info.pw_uid
    else:
        # Maybe validate if UID < 0 ?
        return result


examples = """examples:
    ./deny_inode_permission -u 1000   # deny all inode permissions for
UID 1000
"""

parser = argparse.ArgumentParser(
        description="Deny the specified user of any inode
interactions",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=examples)
parser.add_argument("-u", "--uid", type=parse_uid, metavar='USER',
        required=True, help="trace this UID")
args = parser.parse_args()

bpf_text = """
#include <linux/fs.h>
#include <linux/errno.h>

struct data_t {
u32 uid;
u32 pid;
u64 ts;
unsigned long inode_num;
char comm[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(events);

LSM_PROBE(inode_permission, struct inode *inode, int mask) {

struct data_t data = {};
```

```
u32 uid = bpf_get_current_uid_gid();

UID_FILTER

data.uid = uid;
data.pid = bpf_get_current_pid_tgid();
data.ts = bpf_ktime_get_ns();
data.inode_num = inode->i_ino;
bpf_get_current_comm(&data.comm, sizeof(data.comm));
events.perf_submit(ctx, &data, sizeof(data));
return -EPERM;
}
"""

bpf_text = bpf_text.replace('UID_FILTER',
        'if (uid != %s) { return 0; }' % args.uid)

b = BPF(text=bpf_text)
#b = BPF(src_file="mac_deny_inode_access.c")

# header
print("%-18s %-16s %-6s %-6s %-12s %-6s" % ("TIME(s)", "COMM", "UID",
"PID", "INODE", "MESSAGE"))

# process event
start = 0
def print_event(cpu, data, size):
    global start
    event = b["events"].event(data)
    if start == 0:
        start = event.ts
    time_s = (float(event.ts - start)) / 1000000000
    printb(b"%-18.9f %-16s %-6d %-6d %-12d %s" % (time_s, event.comm,
event.uid, event.pid,
        event.inode_num, b"Deny during inode_permission hook"))

# loop with callback to print_event
b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

Appendix B: Quiet Activity Bash Script

quiet_activity.sh

```
#!/usr/bin/bash

MIN_SLEEP=1
MAX_SLEEP=15
NONROOT_USER=billy
DMZ_IP=10.1.1.1

create_world_writable_file() {

  local tmpfile="/tmp/${RANDOM:-world_writable}"

  date +"%Y-%m-%d %H:%M:%S type=createwoth comm=touch
user=$NONROOT_USER"
  su - $NONROOT_USER -c "umask 0000 && touch $tmpfile >/dev/null 2>&1"
&& rm -f $tmpfile >/dev/null 2>&1

}

add_world_writable_bit() {

  local tmpfile="$(mktemp)"

  date +"%Y-%m-%d %H:%M:%S type=addwoth comm=chmod user=$NONROOT_USER"
  su - $NONROOT_USER -c "chmod o+w $tmpfile >/dev/null 2>&1"
  rm -f $tmpfile >/dev/null 2>&1

}

add_suid_bit() {

  local tmpfile="$(mktemp)"

  echo 'whoami' > $tmpfile
  chown $NONROOT_USER $tmpfile
  date +"%Y-%m-%d %H:%M:%S type=addsuid comm=chmod user=$NONROOT_USER"
  sudo -u $NONROOT_USER chmod u+s $tmpfile >/dev/null 2>&1
  rm -f $tmpfile >/dev/null 2>&1

}

run_suid() {

  local choice=$(shuf -i 0-1 -n 1)

  case $choice in
    0)
      date +"%Y-%m-%d %H:%M:%S type=runsuid comm=sudo
user=$NONROOT_USER"
      sudo -u $NONROOT_USER sudo id >/dev/null 2>&1
      ;;
    1)
```

Billy Wilson, billy_wilson@byu.edu

```
      date +"%Y-%m-%d %H:%M:%S type=runsuid comm=passwd
user=$NONROOT_USER"
      sudo -u $NONROOT_USER passwd -S >/dev/null 2>&1
      ;;
  esac

}

ipv4_connect_attempt() {

  local o1=192
  local o2=168
  local o3=10
  local o4=$(shuf -i 8-23 -n 1)
  local port=$(shuf -i 1-61000 -n 1)

  local choice=$(shuf -i 0-1 -n 1)

  case $choice in
    0)
      # tcp
      date +"%Y-%m-%d %H:%M:%S type=ip4_connect comm=bash
ip=$o1.$o2.$o3.$o4 proto=6 port=$port"
      timeout 15 sudo -u $NONROOT_USER bash -c "echo
>/dev/tcp/$o1.$o2.$o3.$o4/$port 2>/dev/null"
      ;;
    1)
      # udp
      date +"%Y-%m-%d %H:%M:%S type=ip4_connect comm=bash
ip=$o1.$o2.$o3.$o4 proto=17 port=$port"
      timeout 15 sudo -u $NONROOT_USER bash -c "echo
>/dev/udp/$o1.$o2.$o3.$o4/$port 2>/dev/null"
      ;;
  esac
}


ssh_proxy() {

  local choice=$(shuf -i 0-1 -n 1)
  local port=$(shuf -i 10000-60000 -n 1)

  case $choice in
    0)
      local port_fwd_opt="-L $port:ubuntu.com:443"
      ;;
    1)
      local port_fwd_opt="-D $port"
      ;;
  esac

  date +"%Y-%m-%d %H:%M:%S type=ssh_proxy comm=ssh
port_fwd_options=$port_fwd_opt"
  sudo -u $NONROOT_USER ssh -n -o ConnectTimeout=5 -o
ConnectionAttempts=1 $port_fwd_opt $DMZ_IP 'id' >/dev/null 2>&1
```

```
}


kill_bcc_script() {

  local signo_choice=$(shuf -i 0-1 -n 1)
  local bcc_choice=$(shuf -i 0-4 -n 1)

  case $signo_choice in
    0)
      signo=9
      ;;
    1)
      signo=15
      ;;
  esac

  case $bcc_choice in
    0)
      bcc_script="mac_fileperms"
      ;;
    1)
      bcc_script="mac_suidexec"
      ;;
    2)
      bcc_script="mac_sshlisteners"
      ;;
    3)
      bcc_script="mac_socketconnections"
      ;;
    4)
      bcc_script="mac_killtasks"
      ;;
  esac

  date +"%Y-%m-%d %H:%M:%S type=kill comm=pkill user=root signo=$signo
target=$bcc_script"
  pkill -f -$signo $bcc_script >/dev/null 2>&1

}

while :
do

  interval=$(shuf -i $MIN_SLEEP-$MAX_SLEEP -n 1)
  sleep $interval || exit 1

  choice=$(shuf -i 0-6 -n 1)
  case $choice in
  0)
    create_world_writable_file
    ;;
  1)
    add_world_writable_bit
    ;;
  2)
```

```
    add_suid_bit
    ;;
  3)
    run_suid
    ;;
  4)
    ipv4_connect_attempt
    ;;
  5)
    ssh_proxy
    ;;
  6)
    kill_bcc_script
    ;;
  esac

done

exit 0
```

Appendix C: Source Code of mdstress

mdstress.c

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

/*
 * mdstress: Rudimentary metadata stress tester
 *
 * Creates a file, writes a small string to it,
 * then deletes it in a tight loop.
 *
 * *** This program does not validate input! ***
 *
 * Arguments:
 *   @file:             The file to create and delete
 *   @total-inodes:     How many times the inode should be
created/deleted
 *   @rate-per-second:  How many times per second it should be
created/deleted
 */

int main(int argc, char *argv[]) {

        int fd;
        long loops;
        long delay_ns;
        double begin;
        double end;
        struct timeval tv;
        struct timespec next;
        ssize_t num_written;
        char data[9] = "mdstress";

        if (argc < 4 || strcmp(argv[1], "--help") == 0) {
                printf("usage: %s file total-inodes rate-per-
second\n",
                        argv[0]);
                return 0;
        }

        loops = strtol(argv[2], NULL, 10);
        delay_ns = 1000000000 / strtol(argv[3], NULL, 10);

        /* For calculating sleep time per loop */
        clock_gettime(CLOCK_MONOTONIC, &next);

        /* For measuring elapsed wall time at end of execution */
        gettimeofday(&tv, NULL);
        begin = tv.tv_sec * 1000000 + tv.tv_usec;
```

Billy Wilson, billy_wilson@byu.edu

```
        while (loops-- > 0) {
                next.tv_nsec += delay_ns;
                next.tv_sec += next.tv_nsec / 1000000000;
                next.tv_nsec %= 1000000000;

                fd = open(argv[1], O_RDWR | O_CREAT,
                                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP
|
                                S_IROTH | S_IWOTH);
                if (fd == -1)
                        perror("open");

                num_written = write(fd, data, strlen(data));
                if (num_written == -1)
                        perror("write");

                if (close(fd) == -1)
                        perror("close");

                if (unlink(argv[1]) == -1)
                        perror("unlink");

                clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next,
NULL);
        }

        gettimeofday(&tv, NULL);
        end = tv.tv_sec * 1000000 + tv.tv_usec;

        printf("Total inodes: %s | Inodes per second: %s | Time
elapsed: %f\n",
                argv[2], argv[3], (end - begin) / 1000000.0);

        return 0;

}
```

Appendix D: Network Tunings Prior to tcpkali Benchmark

network_tunings.sh

```
#!/bin/bash

# tcpkali has a client-server model. Some of these tunings
# are intended for a server, and others for a client. However,
# for simplicity's sake all, the settings were applied to every
# device participating in the benchmark, whether server or
# client.

N=100000

sysctl fs.file-max=$((10000+2*N))
sysctl net.ipv4.tcp_max_orphans=$((N))

# For load-generating clients.
sysctl net.ipv4.ip_local_port_range="10000 65535"
sysctl net.ipv4.tcp_tw_reuse=1

#For server
sysctl net.core.somaxconn=16384

#For NIC
ifconfig em1 txqueuelen 5000

sysctl net.core.netdev_max_backlog=2000
sysctl net.ipv4.tcp_max_syn_backlog=8192

# Do this manually so it applies to the bash session
# ulimit -n 65536
```

Appendix E: Benchmark Results

Each xhpl result is an average of thirty runs on the node for the given configuration.

xhpl Time Elapsed (Seconds)			
Name	Non-KRSI Kernel	KRSI kernel, No Scripts	KRSI kernel, All BCC Scripts
Node 1	882.26	882.32	882.21
Node 2	896.69	895.10	891.45
Node 3	888.22	888.60	887.67
Node 4	886.10	885.93	886.01
Node 5	878.59	878.67	878.37
Node 6	887.85	888.08	889.28
Node 7	876.81	876.30	876.44
Node 8	878.12	878.79	878.47

xhpl Gigaflops			
Name	Non-KRSI Kernel	KRSI kernel, No Scripts	KRSI kernel, All BCC Scripts
Node 1	711.22	711.17	711.26
Node 2	699.77	701.02	703.89
Node 3	706.44	706.15	706.88
Node 4	708.14	708.27	708.21
Node 5	714.19	714.12	714.37
Node 6	706.74	706.56	705.60
Node 7	715.65	716.06	715.95
Node 8	714.59	714.06	714.32

Each mdstress result is an average of thirty runs on the node for the given configuration.

mdstress Time Elapsed (100,000 inodes @ 10,000 per second)			
Name	0664 with no scripts	0664 w/ mac_fileperms	0666 w/ mac_fileperms
Node 1	10.00	10.00	10.00
Node 2	10.00	10.00	10.00
Node 3	10.00	10.00	10.00
Node 4	10.00	10.00	10.00
Node 5	10.00	10.00	10.00
Node 6	10.00	10.00	10.00
Node 7	10.00	10.00	10.00
Node 8	10.00	10.00	10.00

mdstress Time Elapsed (500,000 inodes @ 50,000 per second)			
Name	0664 with no scripts	0664 w/ mac_fileperms	0666 w/ mac_fileperms
Node 1	13.61	13.88	15.19
Node 2	13.40	13.72	15.46
Node 3	13.66	14.14	15.67
Node 4	14.87	15.76	16.28
Node 5	14.85	14.93	15.69
Node 6	13.06	13.82	15.35
Node 7	13.98	15.05	16.16
Node 8	15.92	14.72	16.07

mdstress Time Elapsed (1,000,000 inodes @ 100,000 per second)			
Name	0664 with no scripts	0664 w/ mac_fileperms	0666 w/ mac_fileperms
Node 1	27.18	27.06	29.54
Node 2	26.93	28.24	29.98
Node 3	26.77	26.39	30.80
Node 4	28.67	29.19	30.21
Node 5	28.76	29.36	31.15
Node 6	26.05	26.13	29.45
Node 7	27.38	28.79	30.57
Node 8	29.63	30.23	31.81

tcpkali (mac_skconnections not loaded)					
Client	Servers	Time	Connections Per Second	Successful Connections	Ideal Max Connections
Node 1	Node 2	30	1500	46422	45000
Node 1	Nodes [2-3]	30	3000	92843	90000
Node 1	Nodes [2-4]	30	4500	139255	135000
Node 1	Nodes [2-5]	30	6000	185594	180000
Node 1	Nodes [2-6]	30	7500	232106	225000
Node 1	Nodes [2-7]	30	9000	276026	270000
Node 1	Nodes [2-8]	30	10500	305430	315000

tcpkali (mac_skconnections loaded)					
Client	Servers	Time	Connections Per Second	Successful Connections	Ideal Max Connections
Node 1	Node 2	30	1500	46415	45000
Node 1	Nodes [2-3]	30	3000	92819	90000
Node 1	Nodes [2-4]	30	4500	139230	135000
Node 1	Nodes [2-5]	30	6000	185608	180000
Node 1	Nodes [2-6]	30	7500	232063	225000
Node 1	Nodes [2-7]	30	9000	275750	270000
Node 1	Nodes [2-8]	30	10500	297864	315000