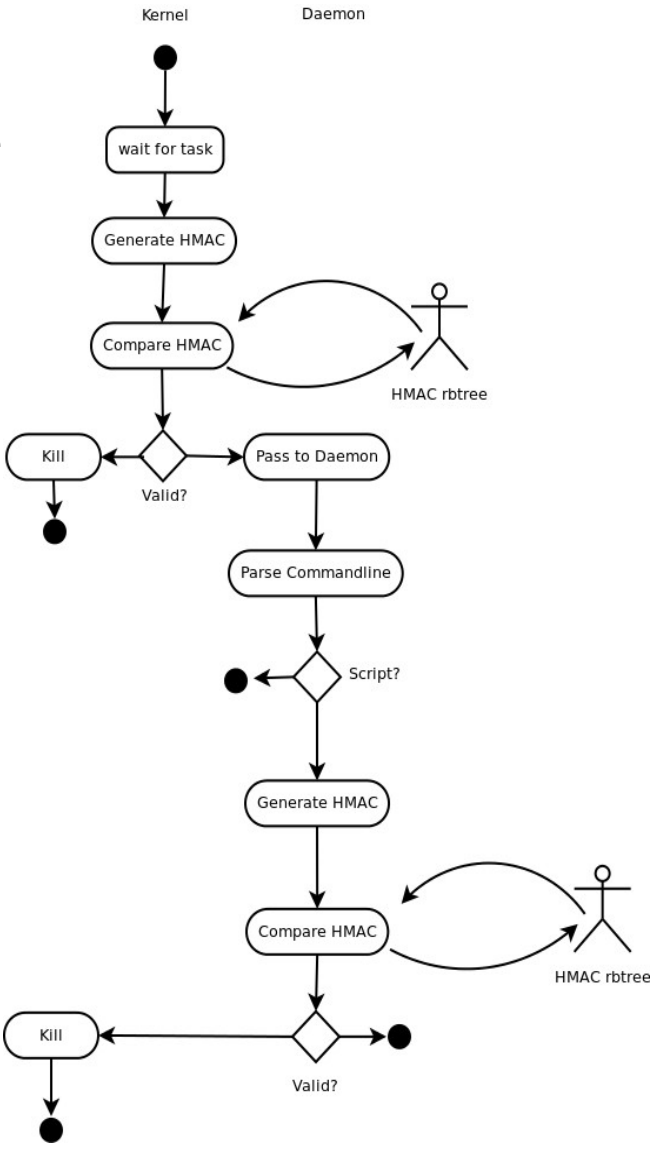
Aerolock Alpha Documentation

Aerolock is an Linux embedded system protection suite based on the white-list paradigm and implemented as a loadable kernel module/daemon or netlink/daemon combination, depending on what's available. The kernel driver depends on `kretprobe` and if that's not enabled, the system falls back to `netlink`; which should always be available. The system is reactive rather than proactive and relies on the kernel task queue for program launch requests. Both systems trap everything that the kernel forks or execs. The current status is Alpha and both the kernel version and netlink support are installable and effective. Testing and tweaking continue with the goal of complete stability and consistent performance. Internally the system uses the stock Linux rb-tree [*O(log n)*] implementation for storing HMACS, NTRU HMAC generation and Berkley DB for HMAC persistence. HMACs are loaded into the driver kernel at startup.



The current test environments are Ubuntu 12.04 (kernel 3.09) for both the development environment and the BeagleBone Black test ARM embedded system. Performance on both systems is adequate to stop executables but there are latency issues with the Daemon while testing scripts; so some get through still. The current focus of development is on reducing this latency and gaining 100% effectiveness against Java, Python, Perl, and any other interpreter/byte-code systems.

Reactive vs. Proactive

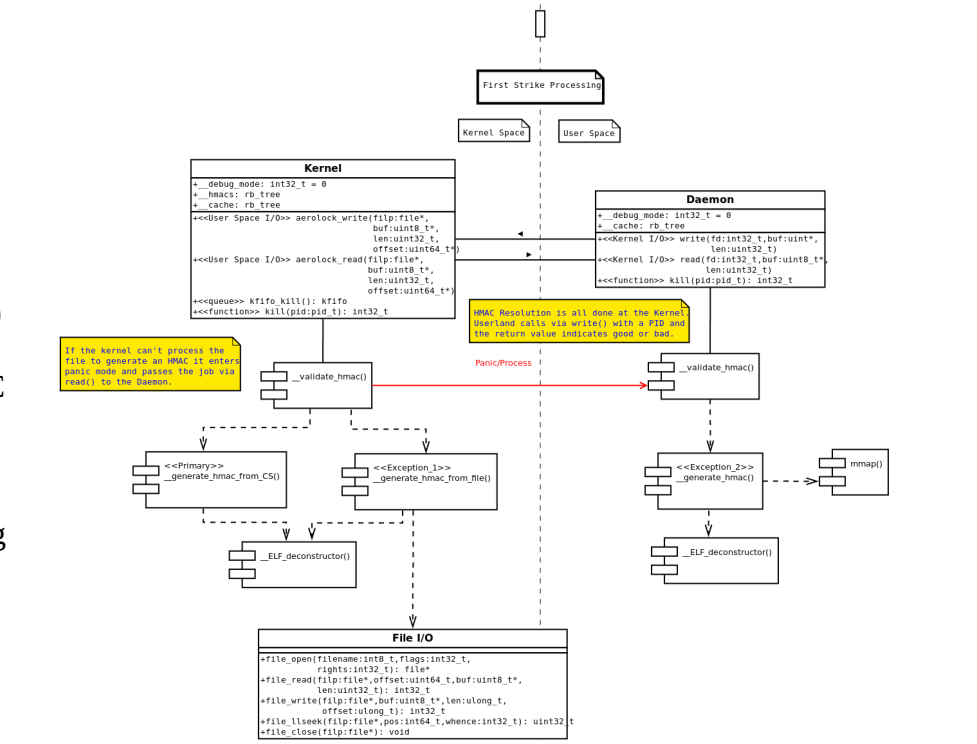
A proactive system stops the process the second it's launched and processes it while a reactive system allows the process to be launched and run. We've tested both and found that the proactive system implemented as an LKM coupled with a reactive Daemon to be the best solution.

The goal is to provide a single, portable mechanism that can be used by all Linux and Linux based systems including Android, RT Linux, QNX, etcetera. Latency observations in the reactive model are as follows. Note all tests were performed on debug code with logging turned on.

	BeagleBone Black	Ubuntu 12.04 32 bit
Processor	ARM 1GHz/512MB	X86x4 2.7GHz/2GB
Kernel	3.8.13	3.8.0-39 Generic
Average Userland Latency	2.6ms	.27ms
Average Kernel Latency	.35ms	.12ms
Memory usage (driver)	38.6K + (32*n of HMACS)	44.5K + (32*n of HMACS)
Memory usage (daemon)	Real(rss) 37.4K, Virtual(vsz) 1K	Real(rss) 18.9K Virtual(vsz) 93.7K
File Count	~4,500	~191,000

Architecture and Flow

The system leverages both a loadable kernel module (LKM) implemented as a character driver; and a daemon. The kernel driver works with the scheduler and intercepts tasks started by the fork() and execve() functions using kretprobe. Each task has an HMAC generated for it on the fly which is then searched for in the rb-tree of HMACs. Should there be a match no action is taken other than placing the PID in a read queue for additional processing by the daemon. If there is no match the PID is placed in a "kill" queue that results in its termination.



There are three methods for generating HMACs:

1) Using the code segment (CS) start and length entries in the mm structure in the task.
2) Using the filename contained in the vma structure.
3) Using the daemon by passing out the process ID (PID).

Method 1 is the primary while 2 &

3 are exception handlers. Using the mm structure CS reference has several problems with it. The first is that the CS points to user space and its difficult to get the data reliably. The second is that the kernel has to allocate a buffer for the CS which is some cases can be quite large and `kmalloc()` and its exception function, `__get_free_pages()` fail. The first exception handler extracts the filename from the virtual memory area (vma) and uses the virtual file system (vfs) to open the file and extract its CS, generates an HMAC and tries to match it in the rb-tree. This method normally works but the use of files from the kernel is frowned on, and sometimes it fails. The second exception handler is a panic function that passes the PID to the user land daemon for processing. This function always succeeds if the PID is valid and the file is available. It opens the file, extracts the code segment, generates an HMAC and then calls the kernel to validate it. While this function always succeeds, the latency involved makes it a last ditch solution.

Userland Processing

Once the file has been validated, the kernel passes the daemon the PID for testing as a scripting engine. Should the program be identified as a scripting engine, the arguments to it are parsed, an HMAC is generated for each argument identified as a file, and the kernel is called to validate the HMAC. Should the argument prove valid, no action is taken. Should the argument prove invalid, the daemon sends the kernel a “kill” command for the scripting engine and the item is queued for termination.

Threading & Queues

The kernel driver runs on two (2) threads. The first is the process context thread and the second is driver managed killer thread, where the process termination code runs. There are two (2) queues implemented to provide data to either the kill or read routines. Kernel queues are used to both bridge the thread gap to the killer and to allow for multiple operations to be handled sequentially should that be required. The daemon runs up to 16 threads, each of which block on the character driver's read() function and allows for handling most everything the kernel demands. Note that each thread is rather large, which is bad in an embedded environment, so the system is typically run with two (2) to four (4) threads, which generally proves adequate for typical systems.

Caching

Both the kernel and the daemon maintain caches for items that have been processed and proved benign. The caches are implemented as rb-trees [*O(log n)*] and reduce latency considerably (TODO: Invalidate caches periodically), generally improving overall performance.

Open Source

There is one piece of open source software in the system and that's a userland implementation of the kernel based rb-tree. This may prove to be a problem both in porting and in the automotive market where open source appears not to be allowed.

Java, Python, and other Scripting Systems

<<write this Pete>>

Components

The system is separated into configuration and runtime components. The runtime components are the device driver, `aerolock.ko`, and the daemon, `aerolockd`. The setup components are `rmprofile`, `rmsetup`, `rmaddhmac`, `rmdepends`, `rmdumpdb`, and `rmverify`.

aerolockd

The aerolock daemon. It may be launched directly from the command line or auto-launched using upstart or init. In a live environment one would naturally choose one of the automatic mechanisms. Switches:

--driver, -m Use the device driver interface. In this mode the system attempts to communicate with the loadable kernel driver (LKD) and use the read/write interface to control the analysis and termination process.

--debug, -d Run in debug mode, non-daemonized. In this mode kill messages are sent but not executed, allowing one to watch the systems operation without effecting normal operations. When using a debugger this mode should be selected.

--test, -t Run in debug mode, daemonized. In this mode kill messages are sent but not executed, allowing one to watch the systems operation without effecting normal operations.

--threads,-c: Sets the maximum number of threads to use during the run (max/default: 16).

--delay: , -u: Delay *(n)* microseconds while parsing `/proc/pid/exe`. On some systems the time needed to write the symlink to the actual executable is takes longer than it does on higher performance systems, hence its necessary to slow things down a bit to allow the kernel to generate the proper entry. It is important to keep this value as low as possible to avoid unnecessary latency. Example; for a BeagleBone Black 1GHz board: aerolockd –driver -- delay 25000

--verbose,-v Enable verbose messages

```
Example:
           ; Load daemon as a daemon in debug mode
           sudo aerolockd --driver --threads 8 --delay 1500 --test

           ; Load the daemon as a regular program in debug mode
            sudo aerolockd --driver --threads 8 --delay 25000 --debug

Using upstart manually:
           sudo start aerolock
           sudo stop aerolock

Using init.d manually:
      sudo /etc/init.d/aerolock start
      sudo /etc/init.d/aerolock stop
```

In an upstart environment one edits the /etc/init/aerolock.conf file, adding or removing command line switches as required.

```
Example (upstart):
/etc/init/aerolock.conf
           start on runlevel [2345]
```

```
stop on runlevel [!2345]
respawn
expect fork
exec /usr/local/bin/aerolockd —-driver  --delay 1500
```

aerolock.ko

The loadable kernel module that intercepts do_fork() and do_execve(), where all processes are launched from. The module interacts directly with the kernel scheduler tasks and pulls the process ID's (pid) from the task structures and queues them for the read() function from user-land to pick up. Rogue processes are sent back to the the kernel via the write() function where the pids to be killed are queued and destroyed.

The module is automatically installed at boot time but may be removed and reinstalled as follows:

```
sudo rmmod aerolock
sudo insmod ${PATHtoModule}/aerolock.ko
```

You should restart the driver should you rescan the system using rmsetup or rmprofile so the driver loads the new HMAC database.

If the driver is uninstalled and `aerolockd` started, the system will default to the `netlink` interface.

Automatic driver execution

Assuming you've executed "`make deploy`" installing the driver so it will boot automatically is accomplished using the following steps:

1. sudo mkdir /lib/modules/`uname -r`/misc
2. sudo cp aerolock.ko /lib/modules/`uname -r`/misc
3. sudo depmod -a
4. sudo vi /etc/modules
5. add aerolock as a line item (not aerolock.ko!)
6. Save and exit

Automatic daemon execution

Assuming you've executed "`make deploy`" installing the daemon so it will boot automatically with upstart is accomplished using the following steps:

```
For upstart:
   1. sudo vi /etc/init/aerolock.conf
   2. uncomment the line that reads:  start on runlevel [2345]
   3. Save and exit

For init.d:
   No action required
```

rmprofile

The profiler is a process monitor that watches and records all system activity. The tool should be run for as long as possible to catch all of the foreground and background operating system activities plus the primary programs the computer is designed for. When complete the profiler will have generated

and stored a collection of HMACs that will be used by `aerolockd` to limit the actual tasks that can be run on the system. The net result of profiling is to turn a general use computer into a completely secure, single purpose system. Should a process be missed, it can be added to the database using `rmaddhmac` which will generate and store an HMAC for the specified program.

--driver, -m Use the device driver interface. In this mode the system attempts to communicate with the loadable kernel driver (LKD) and use the read/write interface to control the analysis and termination process.

--debug, -d Run in debug mode, non-daemonized. In this mode kill messages are sent but not executed, allowing one to watch the systems operation without effecting normal operations. When using a debugger this mode should be selected.

--test, -t Run in debug mode, daemonized. In this mode kill messages are sent but not executed, allowing one to watch the systems operation without effecting normal operations.

--delay: , -u: Delay *(n)* microseconds while parsing `/proc/pid/exe.` On some systems the time needed to write the symlink to the actual executable is takes longer than it does on higher performance systems, hence its necessary to slow things down a bit to allow the kernel to generate the proper entry. It is important to keep this value as low as possible to avoid unnecessary latency. Example; for a BeagleBone Black 1GHz board: aerolockd -- delay 25000

--threads,-c: Sets the maximum number of threads to use during the run (max/default: 16).

--verbose,-v Enable verbose messages

```
Example:
        sudo rmprofiler —-threads 4 —-driver —delay 10000
```

rmsetup

The setup program is used to add bulk items to the database including entire systems or directories. It is non-destructive meaning that if items already exist in the database they will be retained and updated as necessary. Setup can potentially take a very log time to run as it examines and processes every file on the system or specified directory.

--filesystem, -w Examine and white-list all executable files on the system. In this mode the system will generate a new encryption key and update all the HMACs in the database

--directory, -d: Examine and white-list all executable files in the specified directory. In this mode the system uses the existing encryption key and uses it to generate HMACS for all new files

```
Example:
          sudo rmsetup --filesystem
          sudo rmsetup --directory /opt/dev/programs
```

rmaddhmac

Rmaddhmac allows the addition or deletion of individual items in the database.

--add, -a: Add the following path/filename to the database

--delete, -d: Remove the following filename from the database

--replace, -r: Replace the following path/filename

```
Example:
          sudo rmaddhmac --add /opt/dev/new_program
          sudo rmaddhmac --delete program_name
          sudo rmaddhmac --replace /opt/share/replacement_file
```

rmverify

Verify tests the contents of the HMAC database against the items in the filesystem. Any anomalies are reported to the logfile in {cwd}/errors.log. Verify also optionally tests lists and other data structures, again, with errors being reported to stderr, or in the case of a massive failure, a segfault.

```
usage: rmverify -cdrg:hv
          --cache -c,    Only test cache/list
          --database -d, Only test database
          --driver -k,   Test aerolock driver
          --generate, -g,Generate and print HMAC for file
          --verbose  -v, Verbose mode, all file information displayed
          --help -h,     This message

Example:
      rmverify
      rmverify --cache
      rmverify --database --verbose
      rmverify --driver
```

rmdepends

Generates a dependency list for an executable or library using the same depth specification as the primary system. Outputs are the path/filename and the time in milliseconds required to process each individual component and the sum of the times, indicating the overall time required to process the top level file. Note that times are not recorded using a floating point number and is up-converted from nanoseconds, so everything is rounded up or down and therefor the total may not equal to the sum of its parts.

```
sudo ./rmdepends ./aerolockd

TEST: GetSignature(aerolockd), TID(0xb73f26c0)
= libpthread.so.0 [54684 bytes]
== libc.so.6 [50216 bytes]
=== ld-linux.so.2 [1267004 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libc.so.6 processing time: 13ms
== ld-linux.so.2 [50216 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libpthread.so.0 processing time: 14ms
= librt.so.1 [54684 bytes]
== libc.so.6 [15688 bytes]
=== ld-linux.so.2 [1267004 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libc.so.6 processing time: 12ms
== libpthread.so.0 [15688 bytes]
=== libc.so.6 [50216 bytes]
==== ld-linux.so.2 [1267004 bytes]
```

```
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libc.so.6 processing time: 11ms
=== ld-linux.so.2 [50216 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libpthread.so.0 processing time: 12ms
filename: /lib/i386-linux-gnu/librt.so.1 processing time: 25ms
= libdb-5.1.so [54684 bytes]
== libpthread.so.0 [1277192 bytes]
=== libc.so.6 [50216 bytes]
==== ld-linux.so.2 [1267004 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libc.so.6 processing time: 11ms
=== ld-linux.so.2 [50216 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libpthread.so.0 processing time: 12ms
== libc.so.6 [1277192 bytes]
=== ld-linux.so.2 [1267004 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libc.so.6 processing time: 11ms
filename: /usr/lib/i386-linux-gnu/libdb-5.1.so processing time: 49ms
= libc.so.6 [54684 bytes]
== ld-linux.so.2 [1267004 bytes]
filename: /lib/i386-linux-gnu/ld-linux.so.2 processing time: 0ms
filename: /lib/i386-linux-gnu/libc.so.6 processing time: 12ms
filename: rmprofiler processing time: 104ms
TEST: Time - Generate Signature(104ms)
```

rmdumpdb

Allows the user to query the backing Berkley DB or to dump its contents.
Switches:

--database, -d – The name of the database to work on (names or hmacs)
--find, -f – The executable to find. Returns full path
--dump, -r – Dumps the entire name database to stdout. (use 'less' to limit output)

```
./sudo ./aerolock_dumpdb --database names --find  appLoader
```

Building the System

The code has been built and tested on x86 and ARM based platforms running Ubuntu 12.04 (32 & 64). Other systems will be tested as needed and appropriate adjustments as needed. The `Makefile` supports straight build and management functions on both and will support cross compiling of the main code if desired, but the device driver needs to be compiled on the target machine. There are two make files, one for the device driver and one for the main code base.

```
aerolock
      | Makefile
      |------------rmdriver
      |------------rmenforcer
      |------------rmprofiler
      |------------rmsetup
      |------------rmverify
      |------------rmaddhmac
```

```
|------------rmtools
|------------NTRUEncrypt
|------------common
|------------scripts
|------------install
|------------x86
|------------arm
```

Build step 1 – `cd ./aerolock/rmdriver` ; Builds the loadable kernel module (driver)

```
make clean
make
```

Build step 2 – `cd ..`

```
make clean   ; deletes the kernel specific shared files
make         ; Builds the main code body
make release ; Populates the install directory with the production files
sudo make deploy ; Puts all the files in the right places, installs the
                 ; driver and sets up upstart or init.d
```

Running the System

If you rescan the system using rmsetup, you must restart the driver so it will re-read the HMAC database into memory.

```
rmsetup --filesystem or rmprofile --driver --threads 4 --delay 25000
cd /{work-path}/aerolock/rmdriver
sudo make unload
sudo make load
```

The system can be manually started and stopped using the automated controls for starting and stopping daemons or be done manually. By default the system will start automatically on boot which can be changed by editing `/etc/init/aerolock.conf`

```
Start using upstart/init:    sudo start aerolock
Stop using upstart/init:     sudo stop aerolock

Start using init.d:           sudo /etc/init.d/aerolock start
Stop using init.d:           sudo /etc/init.d/aerolock stop

Start using command line:  sudo ./aerolockd --driver --threads 8 --delay 1500
Stop using command line:
ps aux | grep aero
root      2885  2.8  0.5  86020 10556 ? Ssl 11:08   5:05 ./aerolockd --driver ...
sudo kill -SIGTERM 2885
```