Netgroups: Per-process IP Filtering for Unprivileged Users in Linux

{donegan, jugonz97, webbhorn}@mit.edu

http://github.com/webbhorn/linux-stable/tree/netgroups

1. INTRODUCTION

Netgroups is a set of Linux tools that provides granular firewalls for individual processes. Currently, it is easy to block network access for the entire system; modern operating systems like Linux provide this functionality on install. However, this can be an extreme measure, as often times the goal is not to isolate the entire system from an untrusted server but to prevent a specific process or group of processes from such access. On Linux, this can be done, but superuser access is required. Another goal may be to provide only specific network access to a process; this is also currently difficult to do and requires superuser access.

Netgroups provides both functionalities by introducing a new kernel primitive: the network ID (NID). Similar to the existing group ID (GIDs) primitive, every process runs with a list of NIDs that collectively determine its network access rights. Each NID is tied inside the kernel to a policy that cannot be altered, and when a process attempts to send a packet over the network, those policies are checked, effectively 'jailing' processes to the network access determined by its NIDs. Setting these policies is also not restricted to the root user; any user can create a new NID with a custom policy and launch a process using it.

By introducing NIDs as a new primitive, it becomes much easier to control individual processes' access to the internet. An untrusted binary can now be sandboxed from the network entirely by simply creating an NID with a policy that filters out all IP packets. Network privilege isolation for servers can also be achieved by setting a new NID that allows access to only specified IP addresses. Processes that "phone home" by sending usage data to a developer's server can also be prevented from doing so without losing total access to the network.

Our work is inspired in part by UserFS. UserFS gives users access to privilege isolation tools in Linux typically reserved for system administrators, like chroot and chown. It also permits unprivileged users to set firewall rules on a per-UID basis. In the UserFS paper, the authors "...would like users to be able to run a process with a set of firewall rules attached to it... Unfortunately, this would require changing the core Linux kernel...we compromised, and associated firewall rules with UIDs instead" (6). Netgroups implements the kernel changes necessary to attach a set of firewall rules to individual processes, and it also provides the modules and userspace helpers necessary to enforce and define these rules.

2. DESIGN

The ability to provide a composable, expressive and easy to use policy system rests heavily on the specifics of our new kernel level primitive. We propose an ID-based system because it allows a high number of policies that can be referenced easily while also passed from a process to any processes it creates. This means that system calls like `fork()` and `exec()` must preserve all

currently set NIDs in the same way that UIDs are preserved. Without the additional restriction that processes must not be able to unset any NIDs however our rules can simply be ignored by any process aware of our module; therefore, our interface to add an NID to the existing process does not remove any NIDs.

Our main goal for the policy schema is to allow a high degree of expressivity, therefore we allow policies to act on IP addresses. More specifically, a policy is defined by the 4-tuple:

$$policy = (nid, uid, mode, < ip\ address\ list >)$$
$$mode \in \{whitelist, blacklist\}$$

While this is a simple design, its structure satisfies all of our goals. First, policies are restricted to the user that created them. This prevents users from locking out other user's processes from the network and prevents users from having to coordinate with each other to not overwrite policies. After the user ID comes a policy mode, which is either a blacklist or whitelist; this allows many different use cases to be expressed at once. The final field is a list of IP addresses that the policy mode applies to

By design, an $(nid, policy)$ pair is immutable because any modifications may allow privilege escalation. We do not feel that this poses a problem for the user, as the NID type is large enough to allow a sizable number of policies. It is also possible to query the list of set policies for a given UID and NID pair, so there is no need for the user to remember a great deal of policies.

3. IMPLEMENTATION

The implementation of Netgroups comprises three segments: the core kernel modifications, the kernel module, and the user programs. All of our changes are in the Linux kernel source tree, which is forked from kernel version 3.12.0.

Core Kernel Modifications

Each process in Linux is represented by a `task` structure, which further contains a set of security credentials contained in a `cred` structure. We extend the `cred` structure associated with each task, or process, to include an extra field for NIDs. Because new processes inherit the security credentials of their parent, NIDs are copied whenever a process calls `fork()` or `exec()`. This design prevents processes from escaping their sandbox by getting rid of NIDs.

We also add three new system calls that let userspace applications set and read process NIDs. `addnid(nid)` appends `nid` to the list of NIDs contained by the calling process. `getnids(nidsetsize, netgrouplist)` copies at most `nidsetsize` NIDs into the array represented by `netgrouplist`, and `getrnid()` returns the calling processes' "real NID", which is always 0 and is included to enable a future extension by which the administrator can create policies that apply to every process on the system.

Kernel Module

Our kernel module serves two primary purposes: (1) enforce the existing IP filtering rules,

and (2) permit userspace applications to create new policy rules. To enforce policies, our module uses the Netfilter API to register a hook on local outgoing IP packets. When our hook is called, it checks the packet to make sure its source and destination are not violating our policies. If a policy is violated, the packet is dropped.

To enable userspace applications to create policies, the module creates a virtual file in /sys. Users can read and write policies through this world-accessible file, but as we discussed in section two, some special exceptions apply.

User permissions are stored in a hash table that maps *(uid, nid)* tuples to policies that contain a filtering mode (whitelist or blacklist) and a list of IP addresses to be filtered.

User Programs

The user programs in linux-stable/ngtools are not strictly necessary, since access to policies and NIDs is provided via the virtual file and our system calls respectively. That said, they provide a convenient layer for users to quickly and easily interact with our system. They are responsible for creating policies (chngp), viewing existing policies (lsngp), adding NIDs to a process (nidlaunch), and viewing existing NIDs of a process (lsnid).

4. ANALYSIS

Network Performance

We did not have time for extremely comprehensive performance testing, but we did run basic network tests using the iperf program (https://en.wikipedia.org/wiki/Iperf). We did not see a noticeable difference in network speed when our module was installed. The additional computation performed on each packet is insignificant compared to other potential bottlenecks, such as network bandwidth and latency.

Known Issues

One problem with using the Netfilter API is that its hooks are not always executed in the process context, making it difficult to determine the corresponding UID and NIDs of the corresponding process. Some protocols such as UDP and ICMP always execute in a process context, but others like TCP do not. For example, TCP packets are occasionally handled in interrupt context, where there is no easy way to determine its source. Our code provides a relatively unsatisfying method of obtaining credentials—in a more mature version of this project, we would find a polished way of getting user credentials. The credentials we get for the interrupt context are the credentials associated with the related socket, not the credential of the process or user writing to that socket. For the purposes of finishing a complete system, this approximates the correct behavior. This does mean that occasionally we drop a few more packets than necessary, but we favored this over allowing user-prohibited traffic through. A better solution to this problem would probably be to use the Netfilter connection tracking system, known as conntrack, which

tracks network connections and stores information about each packet in memory. Using this system would allow our module to get credentials for a packet even if the call was made in the interrupt context.

Incoming Packet Filtering

A main goal of our project is to stop malicious processes from sending undesirable packets over the network, and we achieve this by inspecting all outgoing packets. Although not strictly necessary for our stated goals, filtering incoming packets would provide a more complete network sandbox. For example, a malicious user might be able to exploit a buffer overflow in a server, even if the server could not reply. Filtering outgoing packets, however, is sufficient for stopping communication with a malicious node in the network. If we were to implement incoming packet filtering, we would run into a similar packet-capturing problem as above—our Netfilter hook would execute in the interrupt context. Again, the likely solution is to use the Netfilter connection tracking subsystem.

5. CONCLUSION

Our hope for Netgroups is to provide a working system that allows control of the network rights of processes, and we achieve this goal through our NID security primitive and Netfilter packet capturing hook. While not without room for improvement, our system is complete, functional and available to the public.

We also hope to set the foundation for more targeted use cases. For example, we envision an extension that enables support for rate-limiting individual processes. The ability to quickly and easily set bandwidth restrictions for processes enables users to easily tune system performance and isolate processes from malicious network access. NIDs would be a useful abstraction for implementing such a system.

We chose this topic for our final project in part to learn about Linux kernel development. Because none of us had any prior experience working in the kernel, it was a very instructive experience. With a constructive team dynamic, we really enjoyed working together on this project and plan to follow through with this and other systems-related projects.

APPENDIX A: Run Our Demo on Your System.

Follow these brief instructions to build, install, and run our system:

1. Run Debian in a VM, and compile and install our custom kernel by following our project notes posted at http://web.mit.edu/~webbhorn/www/netgroups/kernelhacking.
2. Build and load our module by changing directories to the kernel source tree base and running:

```
$ make modules SUBDIRS=net/nfilter
$ cd net/nfilter
$ sudo insmod nfilter.ko
```

3. Build our userspace helper tools by changing directories to the kernel source tree base and running:

```
$ cd ngtools && make
```

APPENDIX B: Examples

With our tools built, you can try out some sample use cases, which we have broken into Scenario 1 and Scenario 2.

Scenario 1

You have just downloaded an untrusted binary (ping) from the Internet, and want to restrict its access to the MIT's web server (18.9.22.69):

```
$ ping 18.9.22.69
$ ./chngp 500 w 18.9.22.69
$ ./lsnid
$ ./nidlaunch 500 /bin/bash
$ ./lsnid
$ ping facebook.com
$ ping 18.9.22.69
```

Scenario 2

Now suppose you notice that an application is contacting a 'spy' or ad server, and you want to continue using the application but not let it access that server:

```
$ ./lsnid
$ ./chngp 501 b 18.9.22.69
$ ./lsnid
$ ./nidlaunch 501 /bin/bash
$ ./lsnid
$ ping facebook.com
$ ping 18.9.22.69
```

Here is another example, with output included:

```
tim@debian:~$ chngp 402 b 68.71.245.5
set 402 b 68.71.245.5

tim@debian:~$ nidlaunch 402 ping 68.71.245.5
PING 68.71.245.5 (68.71.245.5) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
--- 68.71.245.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

tim@debian:~$ nidlaunch 402 ping 18.9.22.69
PING 18.9.22.69 (18.9.22.69) 56(84) bytes of data.
64 bytes from 18.9.22.69: icmp_req=1 ttl=128 time=6.26 ms
--- 18.9.22.69 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 6.265/6.265/6.265/0.000 ms
```