CamFlow Network Provenance

Tray Keller
University of North Carolina at Charlotte

Austin Waddell
University of North Carolina at Charlotte

Jonathan Hurtado
University of North Carolina at Charlotte

Abstract

CamFlow and the Linux Provenance Module (LPM) framework are both an excellent step in the right direction for whole-system provenance, but each have weaknesses. For LPM, the most significant downside is the difficulty in maintaining code across updates to the Linux kernel, as LPM had to duplicate the Linux Security Modules framework in order to allow stacking provenance collection with other functionality. In contrast, CamFlow makes use of Netfilter hooks and advances in LSM modules to greatly reduce the amount of code which must be changed between kernel releases, making the design more modular by implementing the provenance feature using kernel modules. However, CamFlow's network provenance collection has room for improvement, as its method of packet labeling is not scalable, in contrast with LPM's packet labeling methods. To address this scalability issue, we have ported selected parts of LPM's networking code to CamFlow.

1 Introduction

The word provenance means "the place of origin or earliest known history of something". This key word is important in terms of secured systems because of the fact that most operating modern systems cannot trace back the origin of a process or files on a system. This is where the provenance-aware systems comes in handy. There are multiple approaches to collecting data provenance but for the most part they are broken into 3 general categories: provenance-aware applications, provenance-aware file systems, and provenance-aware operating systems. The reason why they are broken into these sections is due to the fact that every operating system are comprised of user-space applications, a file-system, and a kernel. The provenance methods collect data in one of or multiple one of these spaces. Every system has it's pro's and con's however for numerous reasons that will be discussed in the later sections in more detail our team has been assigned modify a provenance-aware operating system technology called CamFlow.

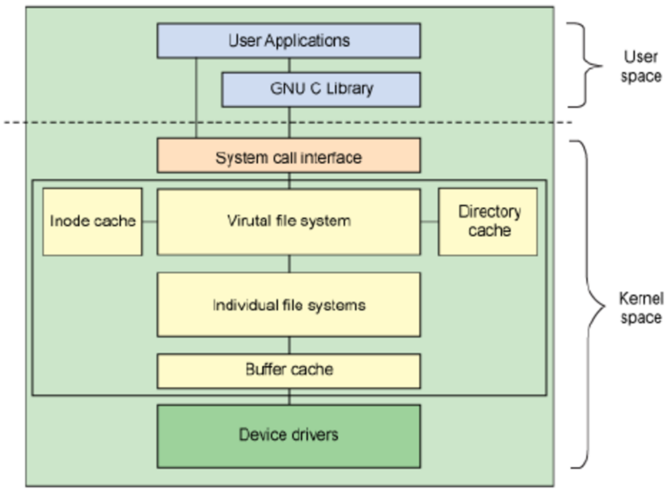


Figure 1: Basic layer of an operating system

2 Background

As discussed previously, there are different layers at which provenance can be collected, including provenance-aware applications, provenance-aware file systems, and provenance-aware operating systems.

First, provenance-aware applications are created by adding a provenance functionality to an already existing program. This could be done by editing the source code of the program or by adding code to a dependency that a application depends on. This in terms would make it so that when the application is in use it would collect data on what the application is doing, such as when a file is being downloaded or when a application is editing something on the file-system. The problem with this approach however is the fact that it is limited to a specific application and that other applications are not being monitored. Another problem is that if a file related to the application once moved or renamed loses the ability to collect provenance data on it. For example, if a provenance aware text editor was created and it was modifying a file if such file was moved or renamed the provenance application would not be able to resume provenance collection due to the fact that it lost the connection to the file.

Provenance-aware file systems track provenance based on the data of the files on the provenance file system that is in use. Examples of what data that could be collected from this type of provenance model could be data related to when a file was was modified or who modified a a file on the system, to what application put the file on the file-system such as a browser when it downloads a file. A limitation of provenance-aware file systems is that it is practically impossible to figure out more information related to the origin of a certain file unless it was file-system specific. For example, if a browser were to download a file then the provenance would be able to keep track of the file; however, the origin of where the file came from such as a website would not be kept. This drawback would prevent investigation of the file and prevention of the file from being placed on the system again in the future.

Lastly, provenance-aware operating systems have the benefits of the prior system and more. Provenance is collected by keeping track of system calls to the operating system. This in turn collects a very large range of information such as file manipulation, application usage, even when a person logs onto a system. However this system comes with it's draw back as well. One draw back is the limitation of the system call. System calls in general are a great way to track information on the system however the specifics of what the application does is not guaranteed. For example, if a web browser is opened in user space on the system and downloads a file, the entire process would be tracked clearly; however what particularly is happening on the web browser may or not be tracked such if the browser in question downloads a file.

By layering application provenance capabilities on top of operating system-level provenance awareness, CamFlow allows the target and scope of provenance collection to be configured with a high level of granularity, but can still produce useful provenance data without needing to modify application source code.

3 Design

To implement more efficient packet labeling in CamFlow, two features were ported from LPM. The first of these was LPM's ingress packet labeling, which was ported to `security/provenance/netfilter.c` in CamFlow. Instead of re-implementing the LSM framework and duplicating features as LPM had done, the equivalent packet labeling functionality was incorporated into CamFlow as a Netfilter hook. As CamFlow currently had a Netfilter hook for outbound IPv4 packet labeling, the additional code only needed to cover inbound IPv4 packet labeling.

The second feature needed for more efficient packet labeling related to provenance hooks for two socket functions `socket_setsockopt` and `socket_getsockopt` inside of `security/provenance/hooks.c`. While not responsible for labeling network traffic, these additional hooks feign ignorance to prevent socket options from being overwritten and are crucial for proper functionality.

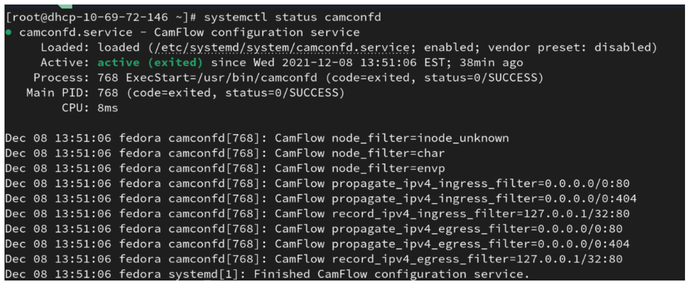
After implementing the network provenance feature the next step was to modify the makefile that compiles the original CamFlow code to compile our code. This is due to the fact that the original CamFlow makefile pulls together multiple files from the original creators repository instead of our own.

4 Evaluation

To test the new network provenance functionality of CamFlow, two virtual machines were created on a university Proxmox cluster. Both virtual machines ran Fedora 35 Workstation (kernel version 5.15.5) and were given 16 GB of RAM and 8 vCPUs. One virtual machine had a vanilla version of CamFlow while the other virtual machine utilized a version of CamFlow that had our code implemented onto it. To begin testing the ipv4 and ipv6 filters were turned on to enable the network provenance functionality of Camflow. Wireshark was then used to capture and compare network traffic from both of the virtual machines, with network captures saved as `.pcap` files. To get the results we used Google Chrome to search for the website "www.google.com" while running Wireshark on both machines. The network capture from the vanilla CamFlow virtual machine was used as a baseline of modifications made by CamFlow, and was compared to the network capture from the modified CamFlow virtual machine to search for differences in packet characteristics caused by our modifications.

5 Related Work

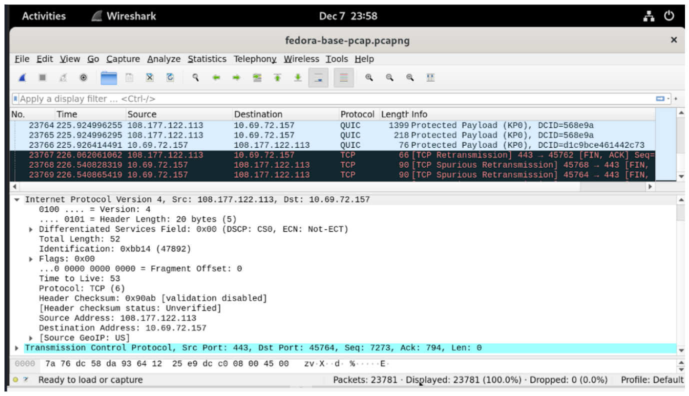
Our work improves the functionality of CamFlow, which incorporated features and advancements from systems such as



```
[root@dhcp-10-69-72-146 ~]# systemctl status camconfd
● camconfd.service - CamFlow configuration service
     Loaded: loaded (/etc/systemd/system/camconfd.service; enabled; vendor preset: disabled)
     Active: active (exited) since Wed 2021-12-08 13:51:06 EST; 38min ago
    Process: 768 ExecStart=/usr/bin/camconfd (code=exited, status=0/SUCCESS)
   Main PID: 768 (code=exited, status=0/SUCCESS)
        CPU: 8ms

Dec 08 13:51:06 fedora camconfd[768]: CamFlow node_filter=inode_unknown
Dec 08 13:51:06 fedora camconfd[768]: CamFlow node_filter=char
Dec 08 13:51:06 fedora camconfd[768]: CamFlow node_filter=envp
Dec 08 13:51:06 fedora camconfd[768]: CamFlow propagate_ipv4_ingress_filter=0.0.0.0/0:80
Dec 08 13:51:06 fedora camconfd[768]: CamFlow propagate_ipv4_ingress_filter=0.0.0.0/0:404
Dec 08 13:51:06 fedora camconfd[768]: CamFlow record_ipv4_ingress_filter=127.0.0.1/32:80
Dec 08 13:51:06 fedora camconfd[768]: CamFlow propagate_ipv4_egress_filter=0.0.0.0/0:80
Dec 08 13:51:06 fedora camconfd[768]: CamFlow propagate_ipv4_egress_filter=0.0.0.0/0:404
Dec 08 13:51:06 fedora camconfd[768]: CamFlow record_ipv4_egress_filter=127.0.0.1/32:80
Dec 08 13:51:06 fedora systemd[1]: Finished CamFlow configuration service.
```

Figure 2: Camflow network filters turned on


Activities
Wireshark
Dec 7 23:58
fedora-base-pcap.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time          Source           Destination      Protocol Length Info
 23764 225.924996255 108.177.122.113 10.69.72.157     QUIC     1399 Protected Payload (KP0), DCID=568e9a
 23765 225.924996295 108.177.122.113 10.69.72.157     QUIC      218 Protected Payload (KP0), DCID=568e9a
 23766 225.926414491 10.69.72.157    108.177.122.113  QUIC       76 Protected Payload (KP0), DCID=d1c9bce461442c73
 23767 226.062061062 108.177.122.113 10.69.72.157     TCP        66 [TCP Retransmission] 443 → 45762 [FIN, ACK] Seq=
 23768 226.540828319 10.69.72.157    108.177.122.113  TCP        90 [TCP Spurious Retransmission] 45768 → 443 [FIN,
 23769 226.540865419 10.69.72.157    108.177.122.113  TCP        90 [TCP Spurious Retransmission] 45764 → 443 [FIN,

▾ Internet Protocol Version 4, Src: 108.177.122.113, Dst: 10.69.72.157
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0xbb14 (47892)
  ▸ Flags: 0x00
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 53
    Protocol: TCP (6)
    Header Checksum: 0x90ab [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 108.177.122.113
    Destination Address: 10.69.72.157
  ▸ [Source GeoIP: US]
▸ Transmission Control Protocol, Src Port: 443, Dst Port: 45764, Seq: 7273, Ack: 794, Len: 0

0000  7a 76 dc 58 da 93 64 12  25 e9 dc c0 08 00 45 00   zv·X··d· %·····E·

Ready to load or capture        Packets: 23781 · Displayed: 23781 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

Figure 3: Network packet data from modified CamFlow

PASS, Hi-Fi, and LPM. PASS, Hi-Fi, and LPM all made contributions to whole-system provenance collection, but were difficult to maintain due to the way their codebases were set up. CamFlow was able to avoid the code maintenance issues related to kernel updates that had plagued these prior systems by taking advantage of advances in Linux features such as Netfilter and the Linux Security Module framework.

CamFlow made whole-system provenance collection practical, but it did not make network packet labeling scalable. Our work extends the scalability of LPM's network packet labeling scheme to CamFlow.

6 Future Work

Due to time contraints we were not able to test the full functionality of Camflow. However since the Camflow kernel compiled completely it is assumed that the network provenance functionality has been successfully implemented and in the future we will perform more testing.We also realised that in order for more developers to help implement new functionality to CamFlow there needs to be a more seamless way to compile the code for testing purposes. We are planning to completely rewrite the build script for CamFlow so that instead of it reaching out to multiple repositories to gather the necessary dependencies to compile, it will instead be placed in one repository so that when an individual clones the repository they will be able to start implementing their changes. We also plan to edit the way CamFlow implements the change code into it. As of right now CamFlow uses a patch file that is generated by git to port over the code to the modified code. This is used to keep track of changes. However this means that for a person to successfully compile the code that an individual has written they must set up git in a specific way which could lead to a slower development process.

7 Conclusion

CamFlow is a whole-system provenance capture mechanism which is easy to maintain since it is implemented using Linux features such as Linux Security Modules, Netfilters, and existing kernel functionality. However, CamFlow tracks network packets in a manner that is not easily scalable. By porting existing Netfilter code from LPM, we were able to bring scalable ingress and egress packet labeling to CamFlow.

References

[1] Adam Bates, Dave Tian, Kevin Butler, and Thomas Moyer. Linux provenance modules. https://linuxprovenance.org/.

[2] Thomas Pasquier. Camflow: Project. https://camflow.org/.

[1, 2].