Linux security modules and whole-system provenance capture

Aarti Kashyap
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
kaarti.sr@gmail.com

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

ABSTRACT

Data provenance describes how data came to be in its present form. It includes data sources and the transformations that have been applied to them. There have been several different OS provenance capture tools in the past. However, only CamFlow and its predecessor Linux Provenance Modules use Linux Security Modules which is a framework that allows the Linux kernel to support a variety of computer security models while avoiding favouritism toward any single security implementation. This paper examines the relationship between LSM and CamFlow. the two key research questions that are addressed: 1) Does the latest version of Linux LSM capture all security related information flows? 2) Given the results of (1) and the data captured by CamFlow in its LSM hooks, can we prove that an intrusion will be reflected as a different in the provenance graphs. This is an important question that either validates or refutes the use of kernel provenance for intrusion detection.

I. INTRODUCTION

Provenance is the chronology of ownership, custody or location of a historical object. These documents are used to guide the authenticity and quality of an item. The term is used in a wide range of fields including science and computing. In computing...

Data provenance can help detect such intrusions in the kernel. Data provenance only provides with the capability to detect intrusions not prevent them. However, in order to make sure that all the intrusions are getting detected, we need a way to capture the complete data flow in the system which is why we chose to work with Camflow.

Camflow is a practical implementation of whole-system provenance capture that can be easily maintained and deployed. They use Linux security modules as the underlying framework to capture the data flows which provides us the ability to...

In this paper, we examine if the whole system provenance capture mechanism developed by Camflow can be utilized for intrusion detection.

Data provenance has wide range of applications ranging from dependibility (reliability and security) of the system to reproducibility of computational experiments.

Security is a major concern since there is no permanent fix to detect intrusions. Its a race between attackers and defenders. An example to show that the kernel is still under threat Xioo et al (22) showed that attackers are able to manipulate the running behaviours of operating systems without injecting any malicious code. This type of an attack is called as kernel data attack. With the power of tampering data, the attackers can stealthily subvert various kernel security mechanisms. The

We first focus on a methodology proposed by Georget et al. to verify if indeed every security related flow goes through an LSM hook. The methodology proposed by Georget et al. had been designed for linux kernel v4.3 to ensure that all security flows are passing through the hooks. We gaurantee the same for v4.20 which is the last formal release.

A second challenge in determining if the use of kernel provenance can help in intrusion detection, we prove that a security breach is reflected in the provenance graph it produces.

Key contributions

The key contributions of our work include: 1) analysing the callgraphs obtained from static analysis of the linux kernel to see if all paths are being tracked 2) a formalism to automate the process of analysing the callgraphs for the future versions 3)prove or disprove if indeed

II. BACKGROUND AND OBJECTIVES

To provide context for the rest of the paper, we first introduce a few concepts. We introduce the notion of whole-system provenance, provenance graphs, linux security modules and complete mediation property.

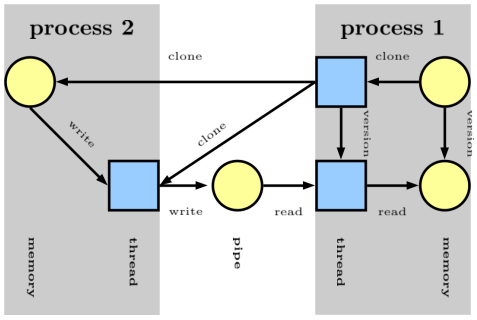

Fig. 1. Process 1 clones process 2. Process 2 writes to a pipe. Process 1 read from the same pipe

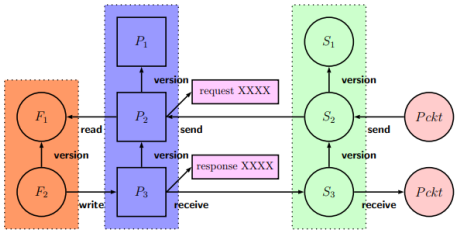

Fig. 2. Annotated provenance graph

A. Whole-system provenance capture

Data provenance was originally introduced to understand the origin of data in a database. (17,18) According to the W3C (19) standards provenance is defined as a directed acyclic graph (DAG). The vertices in the DAG represent entities(data), activities(transformations of data) and agents(persons or organizations). The edges represent the relationships between these elements. Mapping the graph definition to our system, which is the OS level entities are kernel objects, such as messages, network packets, files etc., but also xattributes, inode attributes, exec call parameters, network addresses, etc. Activities are the tasks or processes carrying out manipulations on entities causing data flows. The agents are the persons or the organizations, who control the activities on different entities. These are the users and the groups at the OS level. the agents are users and groups. Fig 1 illustrates these concepts. In the example in Fig 1, process 1 clones process 2. Process 2 writes to a pipe and finally process 1 read from the same pipe.

Processes exchange information via system calls. Some system calls represent information exchange at some discrete point in time e.g, read, write; others may create shared states, e.g, nmap.

There have been multiple whole system provenance capture systems proposed in the past such as HiFi, PASS. However, they had a couple of problems: 1) struggled to keep abreast with current OS releases 2) Did not have whole system provenance capture guarantees 3) generated too much data 4)imposed too much overhead. Learning from the lessons from the past the whole system provenance capture systems, Camflow was introduced which promised to resolve all the above mentioned issues. Camflow addressed the above mentioned shortcomings 1) by leveraging the latest kernel defining to achieve efficiency 2) using a self-contained, easily maintainable implementation relying on Linux Security Modules, Netfilter, and other existing kernel facilities.

B. Linux Security Modules

Since the kernel version 2.6, Linux added support for a framework to implement security extensions for the kernel called Linux Kernel Modules(LSM)(14). This framework provides a set of hooks strategically placed in the kernel code associated with felds in internal data structures for exclusive use by security extensions. The hooks are functions which can be used by security extensions to (1), allocate, free, and maintain the security state of various internal data structures having a dedicated security field, and (2), implement security checks at specific points of execution, based on the security state and a policy. Security modules have a chance to apply security restrictions anywhere a hook is present, but only at these places. LSMs original design is the access control and this has dictated the placement of hooks in the code. . It is thus necessary to verify the correctness of this placement for the purpose of information flow tracking to ensure that information flow trackers.

Since, Camflow uses LSMs and Net-Filters to capture the system provenance, the need to verify the correct placement of hooks for Camflow becomes necessary.

C. Complete Mediation Property

The first goal of our contribution is to verify the property called ”Complete Mediation property”. According to this property for any execution path in the kernel starting with a system call and leading to an information flow, there is at least one LSM hook in the path which is reached before the flow is performed. It’s important to verify this property. The reason is because if there exists a path generating a flow but not going through any LSM hooks, then there exists an opening for a malicious program to perform illegal actions without triggering any alarms. This is because information flow monitor can only react when one of the hooks is reached.

In order to identify all the paths which lead to information flows requires solving two common problems in static analysis. The first problem is that the number of execution paths is infinite because of loops and recursions in the code. In order to finitize the code a subset is selected which should be sufficient to draw a conclusion for all the paths. In other words constructing an abstraction of the code which can be mapped to the concrete code after the analysis is required. The second problem is that many execution paths that appear in the control flow graph (CFG) cannot actually be taken. These are called as the infeasible paths(20).

For our analysis we consider all the system calls in the kernel version 4.20. Flow control requires knowing precisely when an information flow starts and when it stops. If this information is not available, it is not possible to maintain a

correct representation of all flows currently taking place in the system at any given time. Since LSM was designed with access control in mind, some hooks might be missing to perform information flow tracking in every kernel update that comes. However, using static analysis we can find if some hooks are missing to perform information flow tracking.

The purpose of information flow control is to monitor the way in which information is disseminated in the system once it is out of its original container. This is unlike access control which can only enforce rules on how whose containers are accessed. Several scientific and technical challenges exist in ensuring complete information flow. One of them being the large Linux kernel code base. Georget tackles this issue in his work.

Camflow which utilizes LSM for the whole-system provenance capture. It collects the provenance data and constructs provenance graphs from the collected data. Now that we are aware that Georget's methodology ensures the placement of hooks such that complete information flow is possible, we prove/disprove that the violations are reflected in the provenance graphs.

III. LSMs and LSM hooks

Though we have introduced what a Linux Security Module(LSM) means, we will discuss it in a little more detail. The Linux Security Module (LSM) framework provides a mechanism for various security checks to be hooked by new kernel extensions. The name module is a bit of a misnomer since these extensions are not actually loadable kernel modules. Instead, they are selectable at build-time via CONFIG_DEFAULT_SECURITY and can be overridden at boot-time via the "security=..." kernel command line argument, in the case where multiple LSMs were built into a given kernel.

A. LSMs and policies

The primary users of the LSM interface are Mandatory Access Control (MAC) extensions which provide a comprehensive security policy. Examples include SELinux, Smack, Tomoyo, and AppArmor. In addition to the larger MAC extensions, other extensions can be built using the LSM to provide specific changes to system operation when these tweaks are not available in the core functionality of Linux itself.

B. Types of LSMs

Without a specific LSM built into the kernel, the default LSM will be the Linux capabilities system. Most LSMs choose to extend the capabilities system, building their checks on top of the defined capability hooks.

The different types of Linux Security Modules are listed in Table 1. The entire list of the LSMs can be found by reading */sys/kernel/security/lsm*. The Table 1 reflects the order in which checks are made. The capability module is always first, followed by "minor" e.g. Yama) and then the one major module (e.g. SELinux) if there is one configured. This information will help us in understanding the next part of our research.

C. LSM Hooks

In order to explain how a LSM hook works, we show an example in Fig 4. We take system call open as an example. We can see in Fig 4 that just before the kernel addresses an internal object, a check function provided by LSM is called. Hence, LSM allows the modules to understand wheather subject S is allowed to perform an action OP over kernel's internal object OBJ.

TABLE I
LINUX SECURITY MODULES

Different LSMs
AppArmor
LoadPin
SELinux
Smack
TOMOYO
Yama

IV. Modelling the system calls causing flows

The analysis proposed relies on the C compiler from the Gnu Compilers Collection [23], used to compile the Linux kernel

A. Control flow graphs

The analysis technique we use if has been proposed by Georget et al. () for a subset of the system calls. It's a four step methodology which relies on the C compiler from the Gnu Compilers Collection(21).

1) The model designed by Georget to represent system calls and their execution paths does not describe the C source code. They instead use an internal representation called GIMPLE[].
2) Each system call is represented by a control flow graph (CFG).
3) The paths in these graphs model the execution paths in the program as defined by the classical graph theory[].
4) The system calls are analysed one at a time.
5) Each system call contains multiple functions. These functions are inlined into the system calls to reduce the analysis to intra-procedural case.
6) Finally, in the CFGs, two kinds of nodes are marked: the nodes which correspond to the LSM hooks and the nodes which correspond to operation which generate the flows.

B. Constraints in modelling

In the CFGs which we construct , a node is not a basic block but a simple GIMPLE instruction. The analysis methodology does not deal with all expressions and variables of the language. Another reason for the same is that usage of floating-point values is explicitly prohibited in the Linux code. Variables representing structures or unions are also not handled when they involve pointer arithmetic. Global or volatile variables are also not handled, since they can have an arbitrary value at any point in the execution. Ignoring some

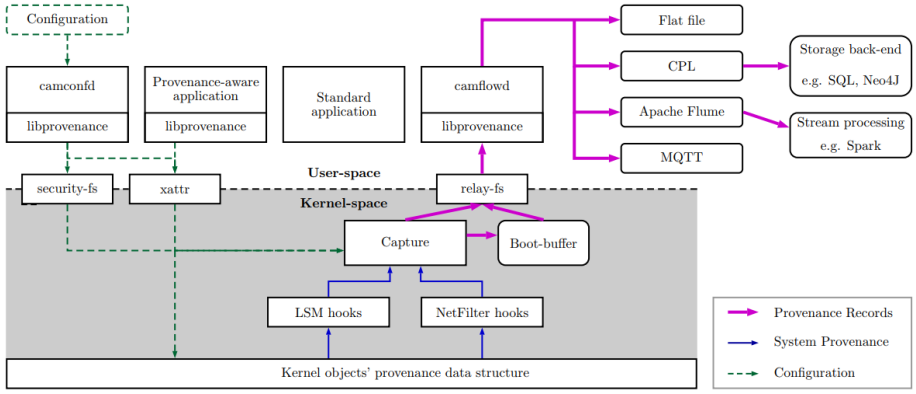

Fig. 3. Camflow architure

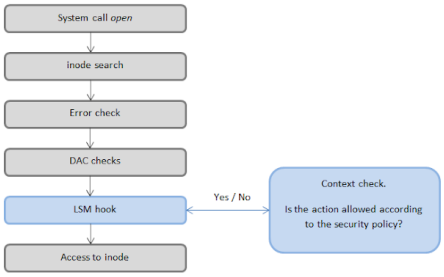

Fig. 4. LSM hook working for system call "open"

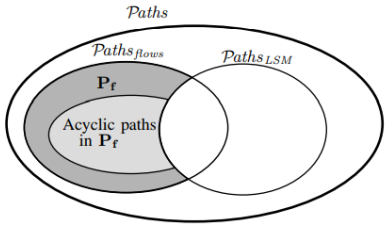

Fig. 5. Sets involved in analysis

variables does not hinder the soundness of our approach: less impossible paths might be detected as such but we never declare as impossible a possible path. A path in the CFG is said to be impossible when any execution that would follow it would enter in an impossible state. For example, a path including two conditional branching with incompatible conditions would require a Boolean expression to be both true and false at the same time.

There are powerful static analysers available such as Blast(24) available. However, for our need to ensure the complete mediation property, we don't need a framework which deals with complex types and data structures. The mediation property which we have described in the previous section despite introducing the above mentioned constraints provides us with precise modelling.

V. Static analysis on paths

The goal of static analysis is to verify that information flow goes through a LSM hook or that it is impossible. In order to do so, we need to find the information flows for any CFG representing a system call.

A. Verifying complete mediation

We consider the set *Paths* of all paths in a CFG.We introduce two particular subsets in this:(1) the set Paths_{flows} of paths starting at the initial node of the CFG and ending at one node generating an information flow; and (2) the set Paths_{LSM} of paths having a node corresponding to a LSM hook. The placement of the hooks would be obviously correct if we could prove that $\text{Paths}_{flows} \subseteq \text{Paths}_{LSM}$. However, as we will see, this is not the case. Pf = Paths_{flows} Paths_{LSM}. The sets involved in the analysis are shown diagrammatically in Fig. 5. represents the set of paths that may be problematic.

As explained earlier, some paths in Paths_{flows} are actually impossible, and therefore even if there are no LSM hooks in them, they are not actually problematic. Recall that an impossible path is a path that does not correspond to a possible execution. The objective of our analysis is thus to verify that $P_f \subseteq I$ where $I \subseteq$ Paths is the set of impossible paths in the CFG.

The property which we are looking to verify is the complete mediation property. Explaining the entire proof is out of scope of this paper. Hence, we state the entire property here.

1) Property 1 (Complete Mediation):: Complete mediation holds iff: $P_f \subseteq I$, i.e. all the execution paths that perform an information flow and are not controlled by the information flow monitor since they do not contain a LSM hook are impossible according to the static analysis.

B. Soundness of proofs

The soundness of our analysis is stated as follows.

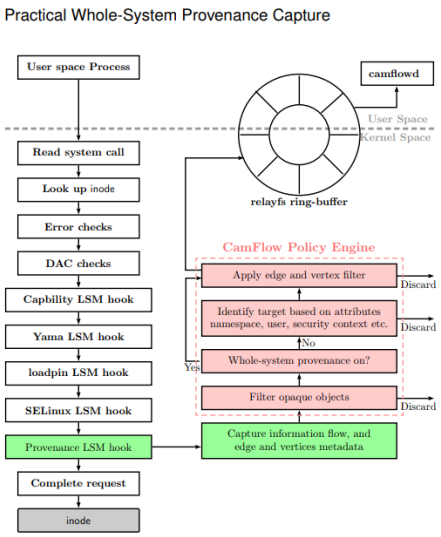

Fig. 6. Executing an open system call. In green is the capture mechanism. The pink is the provenance tailoring mechanism.

1) Proposition 1 (Soundness): For each path p in a CFG, for all concrete configurations θ_1 and θ_2, and all abstract configurations k_1 and k_2 such that $\theta 1$ p $\theta 2$ and k1 p k2, we have 1 k1 = 2 k2.

VI. Results

VII. Limitations

VIII. Related Work

IX. Conclusion

X. Discussion

References

[1] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, Andy Hopper, an "A Primer on Provenance," acmqueue, 2014.

[2] Daniel Crawl and Ilkay Altintas , A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows.

[3] Laurent Georget, Mathieu Jaume, Guillaume Piolle, "Verifying the reliability of operating system-level information flow control systems in linux," Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering, FormaliSE 17, pages 1016, Piscataway, NJ, USA, 2017. IEEE Press.

[4] GERWIN KLEIN, "Operating system verificationAn overview,"Sadhan a Vol. 34, Part 1, February 2009, pp. 2769. Printed in India

[5] Jonathan Pincus and Brandon Baker , "Mitigations for Low-Level Coding Vulnerabilities: Incomparability and Limitations ," 2004.

[6] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, Margo Seltzer, "Runtime Analysis of Whole-System Provenance ,"16 pages, 12 figures, 25th ACM Conference on Computer and Communications Security 2018.

[7] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, Jean Bacon, Practical Whole-System Provenance Capture , SoCC '17 Proceedings of the 2017 Symposium on Cloud Computing.

[8] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer, Harvard University
, "FRAPpuccino: Fault-detection through Runtime Analysis of Provenance ," HotCloud'17.

[9] PASQUIER, T. F.-M., SINGH, J., BACON, J., AND EYERS, D. , "Information flow audit for paas clouds. Cloud Engineering (IC2E), 2016 IEEE International Conference on (2016), IEEE, pp. 4251.

[10] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. , " Provenance-aware storage systems. ," In USENIX Annual Technical Conference, General Track (2006), pp. 4356..

[11] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K , "Hi-fi: collecting high-fidelity whole-system provenance ," In Proceedings of the 28th Annual Computer Security Applications Conference (2012), ACM, pp. 259268.

[12] Adam Bates, Dave (Jing) Tian, and Kevin R.B. Butler , "Trustworthy Whole-System Provenance for the Linux Kernel. 24th USENIX Security Symposium, 2015.

[13] PASQUIER, T. , "Camflow information flow patch. In https://github. com/CamFlow/information- flow- patch. .

[14] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman. , "Linux Security Modules: General Security Support for the Linux Kernel. Proceeding Proceedings of the 11th USENIX Security Symposium Pages 17-31
August 05 - 09, 2002 .

[15] PASQUIER, T. , "CamFlow development. In https://github.com/CamFlow/camflow-dev.

[16] INRIA , "The Kayrebt Toolset In http://kayrebt.gforge.inria.fr/

17 Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In International Conference on Database Theory. Springer, 316330.

18 Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In International Conference on Data Engineering. IEEE, 91102.

19 Khalid Belhajjame, Reza BFar, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, Luc Moreau, and Paolo et al. Missier. 2013. Prov-DM: The PROV Data Model. Technical Report. World Wide Web Consortium (W3C). https://www.w3.org/TR/prov-dm/

20 Rastislav Bodk, Rajiv Gupta, and Mary Lou Soa. 1997. Refining Data Flow Information Using Infeasible Paths. SIGSOFT Software Engineering Notes 22, 6 (Nov. 1997).

21 Richard Matthew Stallman and the GCC developer community. 2013. Using the GNU Compiler Collection (GCC). Technical Report. https://gcc.gnu.org/ onlinedocs/gcc-4.8.4/gcc/

22 author="Xiao, Jidong and Huang, Hai and Wang, Haining", editor="Thuraisingham, Bhavani and Wang, XiaoFeng and Yegneswaran, Vinod", title="Kernel Data Attack Is a Realistic Security Threat", booktitle="Security and Privacy in Communication Networks", year="2015", publisher="Springer International Publishing"/

23 R. M. Stallman and the GCC developer community, Using the GNU Compiler Collection (GCC), Tech. Rep., 2013. [Online]. https://gcc.gnu.org/onlinedocs/gcc-4.8.4/gcc/

24 D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, The software model checker Blast, International Journal on Software Tools for Technology Transfer, vol. 9, no. 5, 2007.