Runtime Verification of Linux Kernel Security Module

Denis Efremov and Ilya Shchepetkov

ISP RAS, Moscow, Russia
{efremov,shchepetkov}@ispras.ru

Abstract. The Linux kernel is one of the most important Free/Libre Open Source Software (FLOSS) projects. It is installed on billions of devices all over the world, which process various sensitive, confidential or simply private data. It is crucial to establish and prove its security properties. This work-in-progress paper presents a method to verify the Linux kernel for conformance with an abstract security policy model written in the Event-B specification language. The method is based on system call tracing and aims at checking that the results of system call execution do not lead to accesses that violate security policy requirements. As a basis for it, we use an additional Event-B specification of the Linux system call interface that is formally proved to satisfy all the requirements of the security policy model. In order to perform the conformance checks we use it to reproduce intercepted system calls and verify accesses.

Keywords: runtime verification, operating system kernel, security policy model, event-b, Linux security modules

1 Introduction

Access control mechanisms in operating systems are usually implemented based on a security policy model, which contains description of the security properties to be enforced by these mechanisms. A security policy model may be a simple text document, but for a certain level of assurance it should be formalized and verified, as stated by the Common Criteria standard [17, 18]. An additional level of assurance may be achieved by demonstrating that the implementation of an access control mechanism indeed conforms to its formal specification.

Access control mechanisms in Linux are implemented in the kernel. We propose to intercept system calls to the kernel while performing various actions like creating, reading, writing, deleting files, spawning processes, etc, and check that the results of their execution do not lead to accesses that are forbidden by the security policy model. It is difficult to check directly because of the abstraction gap. Security policy models are often too high-level comparing to concrete data structures and functions of the Linux kernel. To overcome the difference between the specification and the implementation we develop an Event-B [1] specification of the Linux system call interface, formally prove that it satisfies all requirements of the security policy model (which is also formalized in Event-B), and then translate it to an executable form which is more suitable for checking correctness of intercepted system calls.

The following section briefly describes the security policy model in use. Section 3 depicts the Event-B language in which the model was formalized and verified. Section 4 briefly describes the formal specification of the security model. Section 5 provides a description of an additional Event-B specification required to perform a conformance verification. Section 6 describes the Linux security modules framework, which is used to implement security policy models inside the kernel. Section 7 presents the runtime verification method itself. Related work is observed in Section 8. The final section concludes the paper and considers future work.

2 Security Policy Model

A security policy is a high-level specification of the security properties that a given system should possess, and of security mechanisms that enforce those properties. Security policies are described in the form of security policy models as *state transition systems*, where each possible state transition from a *secure* state must preserve security properties and produce another secure state. The state is declared secure if all current accesses and permissions are in accordance with a security policy.

Operating system (OS) security policy models define the rules for controlling accesses of subjects (users and programs running on their behalf) to various objects (files, directories, devices) and other subjects. They define state transitions as transition functions that model usual OS actions, like creating and deleting files, processes, requesting accesses, etc. Examples of such models would be the classic Bell-LaPadula [5, 4] and Biba [7] models, which were first to describe semantics and security properties of multilevel security and mandatory integrity control respectively.

In this paper we use a Hierarchical Integrated Model of Access Control and information Flows (the HIMACF model, previously known as the MROSL DP-model [8, 9]). It describes means to enforce the separation of information based on confidentiality and integrity requirements. It combines several security mechanisms:

- Role Based Access Control (RBAC). In RBAC, permissions to perform various actions are grouped intro roles and are assigned to a user by an administrator or obtained through special administrative roles. RBAC is often used as a replacement for more simple discretionary access control;
- Mandatory Integrity Control (MIC). In MIC, an integrity level is assigned to all users, processes and files. That level represents their level of trustworthiness, so the higher the level — the more trusted and important a user, a process or a file. MIC controls accesses of subjects to objects according to their integrity levels. MIC is implemented in Windows and macOS to protect system files from modification by users or malicious software;

- Multilevel Security based on Mandatory Access Control (MLS, MAC). It was designed to deal with classified documents in military computer systems. MLS controls accesses according to the user's clearance and the file's classification.

These mechanisms are integrated into a linear hierarchy, where each next level is based on the previous ones. Also since the sequence of perfectly normal and secure accesses may lead to insecure information flows, there is an additional level that contains proofs of their absence (see Fig. 1).

The HIMACF model is implemented in the certified distribution Astra Linux Special Edition [24] using the Linux Security Modules framework [23]. The model is written in the plain text with extensive use of math and consists of approximately 300 pages. We have formalized and verified [10] it using the Event-B specification language. It took us 4 years, and during this process we have found and fixed a number of issues and inconsistencies in the HIMACF model.

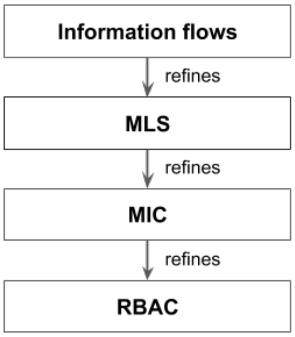

Fig. 1. The hierarchy of levels in the HIMACF model.

3 Event-B

Event-B is a formal method based on set theory and predicate logic. It has a simple notation and comes with a tool support in the form of the Rodin Platform [2]. It is mainly used for developing models of various control systems, but it is also particularly well suited for security policy modeling.

An Event-B specification is a *discrete transition system* and consists of *contexts* and *machines*. Contexts contain the static, or unchanged parts of the specification: definitions of *carrier sets*, *constants*, *axioms*. Machines contain the dynamic or behavioral parts of the specification: *variables*, *invariants* and *events*.

Event-B is a state-based method, so values of variables form the current state of the specification. Events represent the way the state changes over time — the transition. Events may contain *parameters*, *guard conditions* that are necessary for the event to be enabled (or preconditions), and *actions*, that change variables' values. Invariants describe important properties of the system and are supposed to hold whenever variable values change, so such changes need to be explicitly proven to be correct. For each case that requires a proof the Rodin platform generates a corresponding *proof obligation*, that can be discharged automatically using various provers and solvers or interactively. Interactive proofs are also automatically checked for the soundness.

4 Event-B Specification of the HIMACF Model

The HIMACF model uses set theory and predicate logic for defining the state and the properties that the state must satisfy, and it also contains several atomic state transition rules which describe events taking place in the operating system. It makes its structure very similar to the structure of a typical Event-B specification, so its formalization in Event-B was quite straightforward[1].

The state variables of the Event-B specification are expressed as sets and functions (set of ordered pairs with additional restrictions):

- Sets:
  - user accounts;
  - entities (objects and containers);
  - subjects;
  - roles (administrative, ordinary, negative).
- Functions:
  - integrity and security levels (in the form of lattice);
  - current accesses and access rights (or permissions) to entities and roles;
  - hierarchies of roles, entities and subjects;
  - some additional relations between elements of the specification;
  - various flags.

These variables describe the usual operating system elements like user accounts, subjects (which are processes), entities (files, directories, sockets, etc.), and roles. Each of these elements have integrity and security labels that are mapped to them by a number of corresponding functions. Some additional things are also modelled as functions, like current accesses, permissions, hierarchies, and so on.

In total the specification contains 65 state variables. There are also 80 events that describe possible state transitions typical for an OS:

- Create or delete entities, user accounts, subjects, roles;
  - create or delete hard links for entities and roles;
  - rename entities or roles;
- Get or delete accesses, access rights to roles, entities;
- Change security, integrity labels, various flags;
- Additional events for analysis of information flows;
  - example: if an entity x have write access to a subject y, which have write access to a subject z, then there can be an information flow from x to z.

Finally, the specification contains 260 invariants divided into three groups. First one are type invariants: they describe types of all state variables. For example, the type of the variable that contains accesses of subjects to entities is expressed in Event-B like this: $SubjectAccesses \in Subjects \rightarrow (Entities \leftrightarrow Accesses)$. Another group is consistency invariants: they impose correctness constraints on the system state. For instance, if we have a variable that describes

[1] Publicly available part of the specification: https://github.com/17451k/base-model

filesystem (hierarchy of files and folders), then it must not contain cycles, i.e., a folder cannot contain itself, even indirectly.

The last group of invariants is the most important one: it contains all security properties of corresponding security mechanisms. For example, there is the following security property: if a subject has write access to an entity, then its integrity label must be bigger or equal then the integrity label of this entity. It is expressed in Event-B like this: $\forall s, e \cdot s \in Subjects \wedge e \mapsto WriteA \in SubjectAccesses(s) \implies EntityInt(e) \leqslant SubjectInt(s)$.

5 Event-B Specification of the System Call Interface

The HIMACF model and its Event-B specification, however, is quite abstract and different from the concrete data structures and functions of the Linux kernel, which contain the security policy implementation as the Linux Security Module. To prove their conformance it is necessary to reduce this gap. Event-B supports *the refinement technique* [3] to represent systems at different abstraction levels that can be used to resolve this issue.

We have used refinement to develop an additional Event-B specification of the Linux kernel system call interface. Using Rodin we have formally proved that the additional specification correctly refines the Event-B specification of the HIMACF model and thus satisfies its properties. Hence, if we will show the conformance between the additional specification of the system call interface and the Linux kernel, then the desired conformance between the Linux kernel and the security policy model will be derived automatically.

The additional specification, however, has quite an unusual structure. The difference lies in the nature of system calls: the exact sequence of actions that will be performed as the result of the system call depends on the current state of the OS and on the arguments of the call. Because of this variability it is impossible to model them as single atomic events. Instead, we used a different approach.

To overcome this issue we have decided to represent each system call as a *graph of events* connected together with the special state variable called *Next*. *Next* is used to specify the order in which normally independent events should occur. This is achieved as follows: each event in the graph of events have a guard condition specifying that it can only occur if the current value of the *Next* variable is the name of this event. Depending on other guards the event also changes the value of the *Next* variable with the name of the event that should follow next.

Each graph of events representing a system call have a single entry node (the "initial" event), a single exit node (the "final" event) and a large amount of paths in between. Each path is a series of events and the next event in the path is specified by the current value of the *Next* state variable. The path (concrete series of events representing a particular execution of the system call) is defined by the parameters of the "initial" event and the current state of the specification in a way that for each event in the path there is no more than one possible next event.

Let's consider the `open()` system call to open or create a file. This system call has the following declaration[2]: `int open(const char *pathname, int flags)`. The `open()` call has two arguments: `pathname` specifies the file to open, and `flags` determines its access mode: read-only, write-only, or read/write. These access modes are expressed by corresponding flags `O_RDONLY`, `O_WRONLY` and `O_RDWR`. `flags` may also contain additional file creation and status flags. The return value of `open()` is a file descriptor, which can be later used in subsequent system calls (`read()`, `write()`, etc.).

Now let's consider a specific case of `open()` system call in which the file from the `pathname` argument does not exist, and the `flags` argument contains `O_WRONLY` (open file to write) and `O_CREAT` (create file if it does not exist) flags. If the process which calls `open()` has all necessary permissions, then `open()` performs the following sequence of actions:

- parse and validate values of it arguments;
- check that the process has all necessary permissions. In this case the check is successfull;
- get the process write access to the directory where the file will be created;
- create the file;
- get the process permission to write to the created file;
- get the process write access to the created file;
- return file descriptor of created and opened to write file.

This case can be formalized in the Event-B specification of the system call interface as the sequence of 8 events: `open_start, open_check_p, open_write_p, open_create, open_grant, open_check, open_write, open_finish`, where:

- `open_start` contains preconditions (guards) that analyze arguments of the (open) call and decide which event should occur next. In the given case, the file being opened does not exist, so the next event is `open_check_p`. If the file existed, the next event would be `open_check`, and the sequence of events would be different;
- `open_check_p` checks that the process has all necessary permissions. In this case the check is successfull, so the next event is `open_write_p`;
- `open_write_p` is a refinement of the `access_write_entity` event of the Event-B specification of the HIMACF model. This event grants the process write access to the directory where the file will be created;
- `open_create` is a refinement of the `create_object` event of the Event-B specification of the HIMACF model. This event creates the file;
- `open_grant` is a refinement of the `grant_rights` event of the Event-B specification of the HIMACF model. This event grants the process permission to write to the created file;

[2] According to the Linux manual page
http://man7.org/linux/man-pages/man2/open.2.html

- `open_check` checks that the process has necessary permissions to obtain access to the created file and decides which event should occur next. In this case the process opens file to write, so the next event is `open_write`;
- `open_write` is a refinement of the `access_write_entity` event of the Event-B specification of the HIMACF model. This event grants the process write access to created file;
- `open_finish` returns the requested file descriptor.

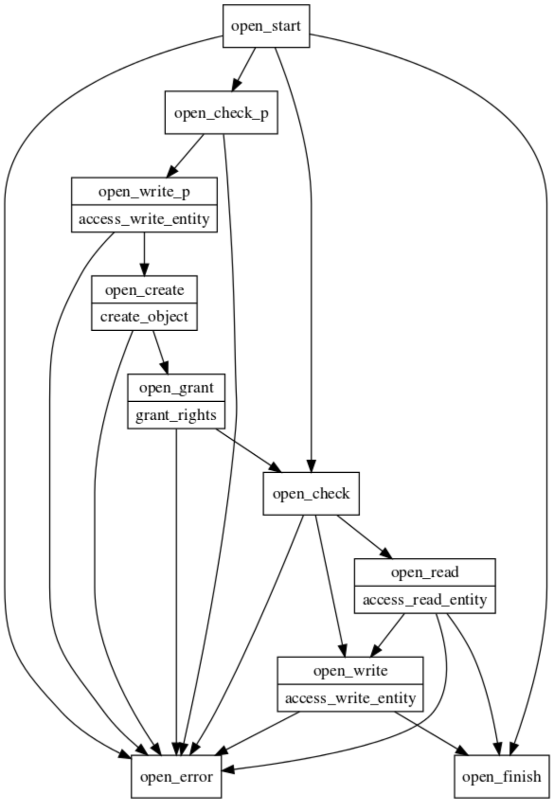

Fig. 2. Graph of events corresponding to several special cases of open() system call.

This sequence of events corresponds to one specific case of `open()` system call. To demonstrate our approach we have formalized a few more cases[3](see Fig. 2). You can see that the graph consists mostly from the same events, but there are more possible paths between them.

[3] Code can be found here: https://github.com/17451k/base-model/tree/open

If we formalize all the remaining cases, the resulting graph will be a formal specification of the behavior of the `open()` system call. Due to the use of refinement, this specification will be correct by construction and fully conform to the rules and events of the HIMACF model. In turn, this will mean that for any combination of parameters and the state of the system, executing the `open()` system call will hold all the security properties of the HIMACF model.

All system calls can be formalized in a similar way, resulting in the specification of the system call interface that is proved to be consistent and complete. But such specification can turn out to be extremely large (several times more than the Event-B specification of the HIMACF model) and difficult to write and prove, mainly from the complicated refinement relation between them (see Fig. 3).

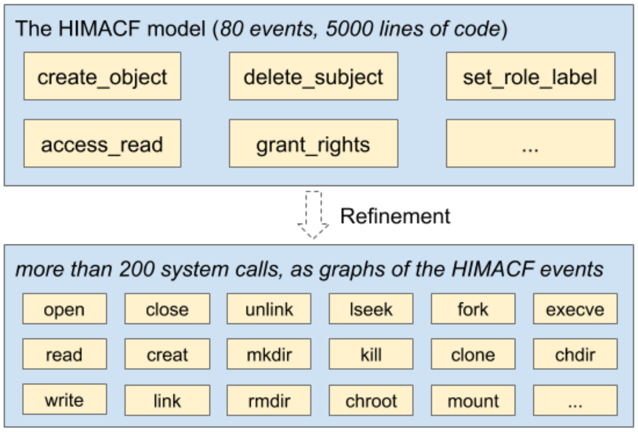

Fig. 3. Refinement between Event-B specifications of the HIMACF model and the system call interface.

6 Linux Security Modules

In Linux, userspace programs work with external resources via the kernel, and make requests for accesses through system calls. When a program executes a system call to, for example, open a file, the kernel performs a number of checks. It verifies the correctness of the passed arguments, checks the possibility of their allocation, and also checks that the program has the permission to obtain the requested resource by the discretionary access control. If more advanced access control mechanisms are used, then the kernel also checks that the access request satisfies their security policy. Such mechanisms are called security modules and based on the Linux Security Modules (LSM) framework. LSM adds a call to a security module after the discretionary access checks in a control flow of a system call handling. These calls are placed across the kernel and called LSM hooks (see Fig. 4).

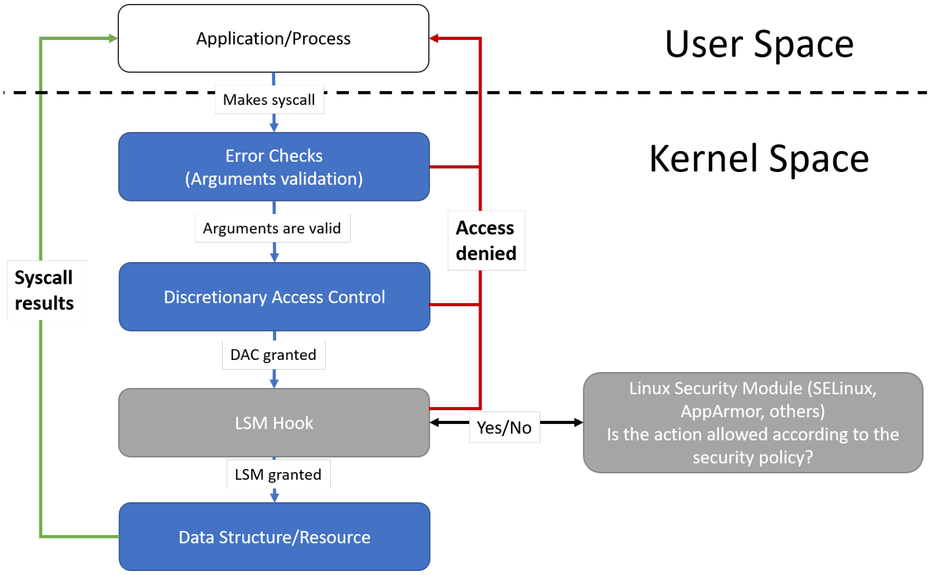

Fig. 4. Linux Security Modules (LSM) hooks.

There are several potential cases when the kernel manages permissions and accesses incorrectly. First, it is possible that the control flow does not reach a security module [27, 12, 6]. The LSM interface may not be complete enough, so it may lack hooks to check certain situations [20, 14]. A security module can also be implemented incorrectly and grant accesses that should not be granted, or deny accesses that should be granted. There is always place for errors due to the abstraction gap and specifics of kernel — module interactions. Thus, we want to verify that the kernel of Astra Linux distribution with the security module indeed conforms the Event-B specification of the HIMACF model and this includes all enumerated errors.

It is worth to note that the Event-B specification of the system call interface does not model such things as the availability of resources (number of processes, virtual memory), and does not contain description of the discretionary access control mechanism. So, for example, the kernel could deny an access due to the lack of physical resources of the machine, but at the same time the specification grants it assuming that the resources are unlimited. Thus the divergence between the specified behavior and the real one should be treated as an error only in case the security policy model denies the access, but the security module grants it.

7 Runtime Verification Method

We propose to demonstrate the absence of such divergences by means of runtime verification, which require a test suite. The test suite should cover various patterns of access requests. In this paper we do not consider the issue of constructing tests and instead use special tests for our model and whole system tests such as Spruce [25], ltp [22] fuzzing with syzkaller [26]. These test suites allow us to achieve relatively good line coverage (more than 80%) on our security module and to cover all LSM hooks in target subsystems of the Linux kernel.

The runtime part of the method is divided into two consecutive steps: gathering of information about the kernel behavior (monitoring) and its analysis.

At the first step, the execution traces of the Linux kernel are collected. It is performed while a test suite is run. In order to reproduce such traces on the specification we also need to record a global state of the kernel, which is performed at the very beginning of this step. This includes, for example, information about running processes, opened files, shared resources, etc.

Traces contain arguments of the system call and the result of its processing by the kernel (output arguments and the result code). Along with this, each trace contains an additional information that is necessary for mapping the global kernel state to the state of the specification, such as inodes and dentries for files, user ids, etc.

We use SystemTap [19] tool to gather the traces from the kernel. It allows one to describe desired probe points in the kernel, such as system calls, with a special language and log the state of in-kernel data structures to a journal.

Algorithm 1 Replay of single system call on the Event-B specification

```
procedure Replay_Syscall(spec, syscall)
    $syscall\_graph := spec[syscall[name]]$
    $params := syscall[args]$
    $event := syscall\_graph[initial]$
    while $event \neq syscall\_graph[final]$ do
        if $guards\_hold(spec[state], event, params)$ then
            $spec[state] \leftarrow event(spec[state], params)$        ▷ update
        else
            return Denied
        end if
        $event := next(spec[state], event, params)$
    end while
    return Granted
end procedure
```

At the second step, we initialize the state of the Event-B specification with the state of the kernel and replay system calls from the trace on the Event-B specification.

The replay algorithm consists of the following steps (see Alg. 1):

1. Pass the arguments of the system call as parameters to the “initial” event of its specification;
2. Check that all guards of the current event are satisfied. If they are not satisfied, then report that the access is denied according to the security policy rules (lines 6, 9);
3. If current event is not “final”, then compute the “next” event, apply event to the current state to change it (line 7), mark the “next” event as current and return to step 2;
4. If current event is “final”, then apply it to the current state and report that the access is granted according to the security policy rules;

Algorithm 2 Replay of kernel traces on the spec

```
 1: procedure REPLAY_TRACE(trace, spec, journal)
 2:     $spec[state] \leftarrow trace[init\_state]$            ▷ initial state for the specification
 3:     while $syscall := shift(trace[syscalls])$ do
 4:         $real\_result := syscall[result]$
 5:         $spec\_result := Replay\_Syscall(spec, syscall)$
 6:         switch $(spec\_result, real\_result)$ of
 7:             case $(Denied, Granted)$ :
 8:                 $journal \leftarrow (CRIT, syscall)$          ▷ An error with its level
 9:                 return$(Failure, journal)$
10:             case $(Granted, Denied)$ :
11:                 $journal \leftarrow Check\_ErrCode(syscall)$
12:                 $spec[state] \leftarrow revert(spec[state], syscall)$      ▷ rollback update
13:         end switch
14:     end while
15:     $journal \leftarrow compare\_states(trace[final\_state], spec[state])$
16:     return $(Success, journal)$
17: end procedure
```

We need to check that the result of the replaying conforms the result of the real system trace execution (see Alg. 2):

1. If access is granted or denied on both the specification and the real system, then we should proceed to the next system call (lines 3, 14);
2. If access is granted on the real system, but it is denied on the Event-B specification (line 7), this clearly signals about an error in the kernel or in the security module. It is not possible to proceed further after this kind of error, the analysis is stopped;
3. If access is denied on the real system, but it is granted on the Event-B specification (line 10), the return code of the system call is investigated (see Alg. 3). For example, if the return code signals about[4]:

[4] The listed error codes are taken from the Linux kernel file `include/uapi/asm-generic/errno-base.h`

   - not enough memory in the system, then no additional actions are taken;
   - invalid values in the system call's arguments, then with high probability this means that the specification is not complete. This divergence is recorded to the anomaly journal;
   - not enough permissions. That means there is an error in the kernel or in the security module. This kind of divergence is recorded to the journal.

   After this we restore the previous state of the specification (line 12) and proceed to the next system call from the trace.

4. If the replay reaches the end of the trace the global states of the kernel and the specification are compared. The divergences are logged to the anomaly journal.

Algorithm 3 Investigation of the error code of a system call

```
 1: procedure CHECK_ERRCODE(syscall)
 2:     $err\_code := syscall[result][err\_code]$
 3:     switch $err\_code$ of
 4:         case ENOMEM :                               ▷ Out of memory
 5:             return $\varnothing$
 6:         case EINVAL :                               ▷ Invalid argument
 7:             return $(WARN, syscall)$
 8:         case EACCES :                               ▷ Permission denied
 9:             return $(CRIT, syscall)$
10:         case . . . :
11:             . . .
12:     end switch
13: end procedure
```

The replay analysis outputs the journal of the divergences between the behavior of the real system and the modelled behavior of the Event-B specification. The journal records need to be analyzed manually to reveal flaws in the specification or the implementation. However, if no divergences were found then with a certain level of certainty based on obtained sources and specification coverage, we can claim that we successfully demonstrated conformance between the implementation and its specification.

We measure the coverage by lines of code of the security module and the number of covered LSM hooks across the kernel. The specification allows more behaviors (states) than it is possible to observe on the real system, thus the specification coverage consists of covered global invariants and different conjuncts of guards conditions. To evaluate the proposed algorithms we have manually translated a part of the Event-B specification of the HIMACF model to an executable program and tested it on the system call traces gathered with SystemTap.

8 Related Work

In the paper [28] Zanin and Mancini present a formal model for analyzing an arbitrary security policy configuration for SELinux. At the end of the paper the authors propose an algorithm based on their model for verifying whether, given an arbitrary security policy configuration, a given subject can access a given object in a given mode. However, they don't go down to the SELinux implementation.

Guttman et al [15] present a formalization of the access control mechanism of the SELinux security server together with a labeled transition system representing an SELinux configuration. Linear temporal logic is used to describe the desired security objectives. The authors use model checking to determine whether security goals hold in a given system.

There are other examples of using formal methods such as B and TLA+ to formalize and prove correctness of various access control mechanisms or security policy models [16, 21], but they also do not consider the implementation.

In the work [11] the correctness of LSM hooks placement in the Linux kernel is analyzed. The proposed runtime verification method leverages the fact that most of the LSM hooks are correctly placed to find the misplaced ones.

The authors of the paper [13] analyze the information flows in the LSM framework. They verify that 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 before the flow is performed. The analysis statically checks the control flow graphs of kernel functions, which are obtained by a compiler plugin during the kernel build, for existence of feasible paths without mediation of the LSM framework.

9 Conclusion and Future Work

We have outlined a method for verification of the access control mechanisms implemented as a module inside the Linux kernel for conformance with its abstract specification. The method consists of several steps. First, one need to formalize the specification of the access control mechanisms in the Event-B language and prove its correctness. Then, since the resulted Event-B specification is high-level and too different from the concrete data structures and functions of the Linux kernel, we propose to develop an additional specification of the Linux system call interface and prove that it conforms to the Event-B specification of access control mechanisms. Next, we trace system calls to the kernel while performing a series of typical user actions and tests. Finally, we replay them on the Event-B specification of the system call interface to check the obtained accesses satisfy the security policy model.

We have evaluated the proposed method on the HIMACF model, which integrates several advanced access control mechanisms, and its implementation inside Astra Linux distribution. We have developed and proved both Event-B specifications, which are required by the method. We have found that the specification of the system call interface, which is required by the method, turns out to be much larger and more complex than the specification of the security policy model. A part of the specification was manually translated to an executable form to obtain the proof of concept and test the replay algorithm of the proposed method. For this we have gathered system call traces with the SystemTap tool. The future work involves development of a translator from Event-B to an effective executable form and research the possibility of simultaneous OS execution and in-kernel verification of accesses.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int J Softw Tools Technol Transfer **12**(6), 447–466 (Nov 2010). https://doi.org/10.1007/s10009-010-0145-y
3. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-b. Fundamenta Informaticae **77**, 1–28 (04 2007)
4. Bell, D.E., La Padula, L.J.: Bell, D. E., LaPadula, L. J. Secure Computer System: Unified Exposition and MULTICS Interpretation. ESD-TR-75-306, Electronic Systems Division, AFSC, Hanscom AFB, 1976. (1976)
5. Bell, D.E., LaPadula, L.J.: Secure Computer Systems: Mathematical Foundations. ESD-TR-73-278 v. 1, Electronic Systems Division, AFSC, Hanscom AFB (1973)
6. Belousov, K., Viro, A.: Linux kernel LSM file permission hook restriction bypass. https://vulners.com/osvdb/OSVDB:25747 (2006)
7. Biba, K.: Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation (1977)
8. Devyanin, P.N.: The models of security of computer systems: access control and information flows. (in Russian). Goryachaya Liniya-Telecom, Moscow, Russia (2013)
9. Devyanin, P., Khoroshilov, A., Kuliamin, V., Petrenko, A., Shchepetkov, I.: Formal Verification of OS Security Model with Alloy and Event-B. In: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 309–313 (Jun 2014). https://doi.org/10.1007/978-3-662-43652-3_30
10. Devyanin, P.N., Khoroshilov, A.V., Kuliamin, V.V., Petrenko, A.K., Shchepetkov, I.V.: Using Refinement in Formal Development of OS Security Model. In: Mazzara, M., Voronkov, A. (eds.) Perspectives of System Informatics. pp. 107–115. Lecture Notes in Computer Science, Springer International Publishing (2016)
11. Edwards, A., Jaeger, T., Zhang, X.: Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 225–234. CCS '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/586110.586141, http://doi.acm.org/10.1145/586110.586141
12. Georget, L.: Add missing LSM hooks in mq timed send,receive and splice. http://thread.gmane.org/gmane.linux.kernel.lsm/28737 (2016)
13. Georget, L., Jaume, M., Tronel, F., Piolle, G., Tong, V.V.T.: Verifying the reliability of operating system-level information flow control systems in linux. In: 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE). pp. 10–16 (May 2017). https://doi.org/10.1109/FormaliSE.2017.1

14. Goyal, V.: Overlayfs SELinux support. https://lwn.net/Articles/693663/ (2016)
15. Guttman, J.D., Herzog, A.L., Ramsdell, J.D., Skorupka, C.W.: Verifying information flow goals in security-enhanced linux. Journal of Computer Security **13**(1), 115–134 (2005)
16. Huynh, N., Frappier, M., Mammar, A., Laleau, R., Desharnais, J.: Validating the RBAC ANSI 2012 Standard Using B. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 255–270. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (Jun 2014), https://link.springer.com/chapter/10.1007/978-3-662-43652-3_22
17. ISO/IEC 15408-1:2009. Information technology – Security techniques – Evaluation criteria for IT security – Part 1: Introduction and general model. ISO (2009)
18. ISO/IEC 15408-2:2008. Information technology – Security techniques – Evaluation criteria for IT security – Part 2: Security functional components. ISO (2008)
19. Jacob, B., Larson, P., Leitao, B., Da Silva, S.: SystemTap: instrumenting the linux kernel for analyzing performance and functional problems. IBM Redbook **116** (2008)
20. Jurgens, D.: SELinux support for Infiniband RDMA. https://lwn.net/Articles/684431/ (2016)
21. Kozachok, A.: TLA+ based access control model specification. Proceedings of the Institute for System Programming of the RAS **30**, 147–162 (Jan 2018). https://doi.org/10.15514/ISPRAS-2018-30(5)-9
22. Larson, P.: Testing Linux with the Linux Test Project. In: Ottawa Linux Symposium. p. 265 (2002)
23. Morris, J., Smalley, S., Kroah-Hartman, G.: Linux security modules: General security support for the linux kernel. In: USENIX Security Symposium. pp. 17–31. ACM Berkeley, CA (2002)
24. RusBITech: Astra Linux® Special Edition. https://astralinux.ru/products/astra-linux-special-edition/
25. Tsirunyan, K., Martirosyan, V., Tsyvarev, A.: The Spruce System: quality verification of Linux file systems drivers. In: Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering. ISP RAS (2012)
26. Vykov, D.: Syzkaller. https://github.com/google/syzkaller (2015)
27. Write, C.: LSM update, another missing hook. https://lwn.net/Articles/155496/ (2005)
28. Zanin, G., Mancini, L.V.: Towards a formal model for security policies specification and validation in the SELinux system. In: Proceedings of the ninth ACM symposium on Access control models and technologies. pp. 136–145. ACM (2004)