FlowConSEAL: Automatic Flow Consistency Analysis of SEAndroid and SELinux Policies

B. S. Radhika[1(✉)], N. V. Narendra Kumar[2], and R. K. Shyamasundar[1]

[1] Indian Institute of Technology Bombay, Mumbai, India
`radhikabs184@gmail.com`, `shyamasundar@gmail.com`

[2] Institute for Development and Research in Banking Technology, Hyderabad, India
`naren.nelabhotla@gmail.com`

Abstract. SELinux/SEAndroid policies used in practice contain tens of thousands of access rules making it hard to analyse them. In this paper, we present an algorithm for reasoning about the consistency of a given policy by analysing the *information flows* implied by it. For this purpose, we model SELinux policy rules using the Readers-Writers Flow Model (RWFM). Using this model, our method identifies all possible indirect flows due to a given policy that could lead to inconsistency. One of the main features of the method is that it not only identifies inconsistencies in the policy but also traces the rules that lead to inconsistency. To distinguish between benign and vulnerable indirect flows, we further categorise the indirect rules that directly contradict `neverallow` rules in the policy and hence have a high potential for information leak. We further rank the rules and domains based on the number of policy violations they cause. We have also implemented a tool FlowConSEAL based on the above method and have applied it on various SELinux/SEAndroid policies for providing a succinct feedback to the user.

1 Introduction

In this digital era, protecting data from intentional and unintentional misuse has become a major concern. Security of Operating System (OS) plays a vital role in data protection and privacy. With Linux kernel forming the core of a wide range of computing devices ranging from mobile phones to supercomputers, its security is of paramount importance. Over the years, several efforts have been made to enhance the security of Linux, SELinux [1] being a prominent example.

Traditionally, Linux supports Discretionary Access Controls (DAC) where access decisions are taken based on the user identity and the permission bits of the object. It is well known that DAC alone is not powerful enough to effectively protect the system because of its inherent weaknesses.

We thank Asokan N, Elena R and Filippo B for their invaluable insights on SEAndroid policy analysis and for sharing SEAndroid policies in early stages of the work.

F. Kerschbaum and S. Paraboschi (Eds.): DBSec 2018, LNCS 10980, pp. 219–231, 2018.
https://doi.org/10.1007/978-3-319-95729-6_14

SELinux introduced Mandatory Access Controls (MAC) to overcome DAC's drawbacks and enhance security through fine-grained access control. It does so by labeling every entity in the system such as files, sockets, processes etc., and specifying a policy to control accesses based on the labels of subjects and objects involved in actions. In addition to providing better protection against unauthorized accesses, SELinux also helps in confining the attack in case of a breach. From Android 4.3 onward SELinux is also being used in Android (referred to as SEAndroid) to provide better application sandboxing and fine-grained access control[1].

In SELinux systems, a well-written policy is the key to protecting the system resources against security threats. However, as these policies get larger and complex, assuring the consistency of all the rules and information flows allowed by them becomes difficult. Currently, the tools [2] used for writing and analysing these policies are not sufficient for detecting information leaks in them. In this paper, we describe a method to analyse information flows implied by a given SELinux policy, and verify their consistency with respect to the accesses in the given policy. The main contributions of the paper are:

1 Automatically analysing consistency of SELinux policies via implied information flows (IF), enabling the policy writers in preventing IF leaks.
2 Identifying and producing evidence for indirect IFs which violate `neverallow` rules specified by the policy.
3 Identifying security critical rules and domains.
4 Implementation of the tool FlowConSEAL to demonstrate the effectiveness of our approach by applying it on various real-life policies.

In the rest of the paper, background is provided in Sect. 2, followed by the need for IF analysis of SELinux policies in Sect. 3. Our approach and experimental analysis are given in Sect. 4 and Sect. 5 respectively. Discussion on related work is presented in Sect. 6. Conclusions are presented in Sect. 7.

2 Background

2.1 SELinux

SELinux is a MAC system implemented using Linux Security Module (LSM) framework [3]. LSM modules work on top of Linux's built-in DAC and enhance its security. In an SELinux system, every subject (active entities like processes) and object (passive entities like files,sockets etc.) is assigned a label which consists of four fields corresponding to SELinux user, role, type and an optional level and it is denoted as `user:role:type[: level]`. The third field `type` represents the logical grouping to which the entity belongs (`type` of a subject is commonly referred as a domain). Although SELinux supports policies based on both `type`

[1] As SELinux and SEAndroid policies have the same syntax, our approach is applicable to both families.

field (Type Enforcement policy) and `level` field (MLS, MCS policies), Type Enforcement (TE) policies are the most commonly used in practice. The analysis presented in this paper is focused on SELinux TE policies.

TE policy supports several types of rules. In this paper we are concerned with two predominant rules - `allow` and `neverallow`. Every time a subject attempts to perform an action on an object, a request is sent to the SELinux module. Access decision is taken based on the subject and object's label. By default, every access is denied. `allow` rules are used to explicitly grant access permission. Unlike `allow`, the `neverallow` rules are used at policy compilation time to ensure that there are no corresponding `allow` rules. The general syntax of these rules is `rule source target:class permissions` where `rule` represents the rule name, `source` represents the `type` of the subject requesting the access, `target` represents the `type` of the object which is being accessed, `class` represents the category of the object (such as file, socket etc.) and `permissions` denote actions associated with the object class.

2.2 Readers-Writers Flow Model (RWFM)

In a MAC system, we can ensure information flow security by employing a suitable formal information flow model and ensuring that the MAC policy conforms to the model. In this paper, we use RWFM [4] model to capture the information flows in a given SELinux policy. RWFM is a powerful lattice-based information flow model based on Dennings model [5]. It can be used to provide both confidentiality and integrity. It supports dynamic labeling and declassification. Also, it can capture several well-known models like BLP [6], Biba [7] etc. Its labeling and access rules are described below.

Labeling: Let S and O be the set of subjects and objects in the system respectively. An RWFM label, also called as RW Class is defined as a triplet (s, R, W), where $s \in S$ denotes the owner of the information in the class, $R \in 2^S$ denotes the set of subjects which can read the objects of the class, and $W \in 2^S$ denotes the set of subjects which can write or which have influenced the class.

Access Rules: Let $owner(x), R(x)$ and $W(x)$ be the functions mapping $S \cup O$ to the *owner*, *readers* and *writers* components of the label respectively. Under the above labeling model, access rules of RWFM are specified as follows:

- A subject s is allowed to read an object o if $owner(s) \in R(o)$ and $R(o) \supseteq R(s)$ and $W(o) \subseteq W(s)$
- A subject s is allowed to write an object o if $owner(s) \in W(o)$ and $R(s) \supseteq R(o)$ and $W(s) \subseteq W(o)$

3 Consistency Problem of SELinux Policies

In this section, we explain the indirect and contradictory rules and the associated security concern. Consider the following set of rules **R**:

```
1 neverallow mozilla_t security_t:file write;
2 allow mozilla_t user_home_t:file write;
3 allow sysadm_sudo_t user_home_t:file read;
4 allow sysadm_sudo_t security_t:file write;
```

The second rule in **R** permits `mozilla_t` to write to `user_home_t:file`. The third rule allows `sysadm_sudo_t` to read from `user_home_t:file`, and the last rule allows `sysadm_sudo_t` to write to `security_t:file`. When actions permitted by the last three rules are performed in that sequence, `mozilla_t` can write some data into `user_home_t` file, and `sysadm_sudo_t` can then read this content and write it into a `security_t` file. As a result, `mozilla_t` can indirectly write to a `security_t` file which the policy writer intended to prevent using Rule 1.

SELinux enforcement of the policy fails to prevent such accesses because it only controls individual accesses, and does not take the information flows caused by these actions into account. Whenever a subject performs an action on an object, the action results in an information flow between them. The direction of such flow depends on the nature of the action. In case of a read, information flows from the object to the subject, whereas in case of a write, flow is from the subject to the object. When multiple actions are performed, the resulting information flow may lead to unintended accesses.

The main objective of this paper is to identify all potential indirect accesses caused by chaining legal accesses of a policy. However, not all of them necessarily lead to a security breach. We focus only on the set of indirect accesses which have corresponding `neverallow` rules in the policy similar to the indirect access resulted due to rules 2–4 in **R** which contradicts rule 1. Such rules allow the accesses explicitly denied by the policy writers and hence are obviously a security concern and need to be further analysed. We call such rules as contradictory rules and study their impact on security. Our analysis provides useful feedback to policy writers which can be used to better understand the impact of their rules and develop flow secure policies.

4 SELinux Policy Analysis: Our Approach

Our approach has five main steps that are described in detail below:

Step 1: Canonicalization of rules

In practice, a rule may contain sets of domains, types, object classes and permissions. In such cases, a single rule corresponds to multiple accesses, one for each (domain, type, class, permission) combination in the rule. To understand the effect of each individual access on the information flow, it is necessary to consider each such combination as a separate rule. So we canonicalize rules such that each resulting rule corresponds to a single access. It will help us extract precise information such as the rules responsible for an indirect flow, number of indirect rules caused by each domain and so on. Further, to clearly differentiate between objects of same type but different object classes, we use both type and class to uniquely identify such combinations. In the rest of the paper we use the term "object type" to refer to this combination unless specified otherwise. We define a function *canonicalize()* which takes a policy as input and returns the set of corresponding canonicalized rules.

Consider the following simplified policy P consisting of set of domains $D = \{d1, d2\}$, set of object types $T = \{t1, t2\}$, and permissions r and w which correspond to read and write operations respectively.

Policy Rules in P

```
1 allow d1 t1 {r, w}
2 allow d2 t2 {r, w}
3 allow d1 t2 {w}
```

Canonicalized Rules of P

```
1 allow d1 t1 r
2 allow d1 t1 w
3 allow d2 t2 r
4 allow d2 t2 w
5 allow d1 t2 w
```

Step 2: Extraction of labels of object types

In our analysis, we consider information flows between domains and object types in terms of RWFM rules. For this, we first need to assign RWFM labels to the domains and object types. Since we are working at the granularity of domain/object types, we ignore the owner field of the RWFM label. Thus the labels are of the form (R, W), where R stands for readers and W for writers/influencers.

For extracting readers and writers of any object type, we need to find the set of domains which have read and write permissions for that object type respectively. We do this by iterating over all the `allow` rules in the policy. For each `allow` rule of the form `allow d t r`, we add d to $R(t)$ and for `allow d t w`, we add d to $W(t)$. This procedure is described in Algorithm 1.

At present, we focus only on read and write permissions in the policy since they are the high bandwidth channels. Other permissions can be mapped into either read or write depending on whether they cause outward or inward information flow. Our implementation is generic and can use such mappings to consider any permission of interest.

On applying Step 2 on the set of rules obtained from Step 1, $t1$'s label will be $(\{d1\}, \{d1\})$ and $t2$'s label will be $(\{d2\}, \{d1, d2\})$.

Algorithm 1. LabelObjectTypes

```
Data: Canonicalized policy rules
Result: Labels ($L_{ot}$) of all the object types in the policy

foreach $t \in T$ do
    $R(t) = W(t) = \{\}$
end

foreach rule "allow d t perm" do
    if $perm = r$ then
        $R(t) = R(t) \cup d$
    else if $perm = w$ then
        $W(t) = W(t) \cup d$
    end
end
```

Algorithm 2. LabelDomains

```
Data: Canonicalized policy rules and $L_{ot}$
Result: Labels ($L_d$)of all the domains in the policy
foreach $d \in D$ do
    R(d) = W(d) = D
end
foreach $t \in T$ do
    foreach $d \in R(t)$ do
        R(d) = R(d) $\cap$ R(t)
    end
    foreach $d \in W(t)$ do
        W(d) = W(d) $\cap$ W(t)
    end
end
```

Step 3: Extraction of labels of domains
Once the labels of object types are obtained, we use them to derive labels for domains in the policy. Algorithm 2 describes this procedure. Here we start with the universal set of domains for reader and writer sets. For each object type t which contains d in its reader set, we update the $R(d)$ as the intersection of $R(t)$ and $R(d)$. Since read operation causes information flow from object to subject, as per RWFM rule, the label of the domain (subject) should dominate the label of the type (object). For this, $R(d) \subseteq R(t)$ should hold. Hence we update $R(d)$ as $R(d) \cap R(t)$. Similarly, when a domain d is in writer set of an object type t, we update $W(d)$ as $W(d) \cap W(t)$. On applying this algorithm on the sample policy, label of $d1$ will be $(\{d1\}, \{d1\})$ and label of $d2$ will be $(\{d2\}\{d1, d2\})$.

Step 4: Identification of indirect accesses
Once we have the labels for all the object types and domains in the policy, we apply the following RWFM access checks on each `allow` rule in the policy:

$$d \in R(t) \Rightarrow (R(t) \supseteq R(d)) \wedge (W(t) \subseteq W(d)) \tag{1}$$

$$d \in W(t) \Rightarrow (R(d) \supseteq R(t)) \wedge (W(d) \subseteq W(t)) \tag{2}$$

These checks help us verify whether the information flows caused due to the accesses respect the permissions specified in the policy. Algorithm 3 describes the procedure used for the checks.

Algorithm 3. AccessRuleChecks

```
Data: Canonicalized policy rules, labels of object types ($L_{ot}$), and labels of all domains ($L_d$)
Result: Set of rules corresponding to indirect flow
IndirectRulesSet = {}
foreach rule "allow d t perm" do
    if $perm = r$ AND $W(t) \not\subseteq W(d)$ then
        foreach $d1 \in (W(t) - W(d))$ do
            foreach $t1$ which has $d \in W(t1)$ do
                IndirectRulesSet = IndirectRulesSet ∪ {allow d1 t1 w }
            end
        end
    else if $perm = w$ AND $R(d) \not\supseteq R(t)$ then
        foreach $d1 \in (R(t) - R(d))$ do
            foreach $t1$which has $d \in R(t1)$ do
                IndirectRulesSet = IndirectRulesSet ∪ {allow d1 t1 r }
            end
        end
    end
end
```

With the label derivation methods described in Step 2 and 3, we can say that the conditions $R(t) \supseteq R(d)$ in (1) and $W(d) \subseteq W(t)$ in (2) will always be satisfied. Hence, we check only the remaining conditions. Failure to satisfy these conditions imply the presence of indirect flows. i.e if the condition $W(t) \subseteq W(d)$ fails in (1), then all the domains in $(W(t) - W(d))$ can indirectly write to all the types that d can write. Similarly, if $(R(d) \supseteq R(t)$ in the above condition fails, that means that all the domains in $(R(t) - R(d))$ can read everything that d can read. We construct `allow` rules corresponding to these indirect accesses and store them in IndirectRulesSet. Applying these checks to our sample policy will show that the check fails at rule 5. Here $R(d1) \not\supseteq R(t2)$. Hence $d2$ can read $t1$ even though there is no rule granting this permission.

The steps described above can detect only one level of indirection i.e indirect flows via a single pair of subject and object. However, there can be multiple levels of indirect flows. For example, if we add rules `allow d3 t3 {r, w}` and `allow d2 t3 {w}` to our example policy, it would lead to a two level indirection. To find multi-level indirect flows, we first need to add the first level indirect rules identified i.e rules in IndirectRuleSet to our policy rules and repeat Step 2 to Step 4. We do this until there are no more indirect flows caused by the rule set. Algorithm 4 describes this procedure.

Algorithm 4. SELinuxPolicyConsistencyCheck

```
Data: SELinux policy P
Result: Set of all possible indirect allows
consistent = False
RuleSet = Canonicalize(P)
while not consistent do
    $L_{ot}$ = LabelObjectTypes(RuleSet)
    $L_d$ = LabelDomains(RuleSet, $L_{ot}$)
    IndirectRuleSet = AccessRuleChecks(RuleSet, $L_{ot}$, $L_d$)
    if IndirectRuleSet is $\emptyset$ then
        consistent = True
    else
        RuleSet = RuleSet $\cup$ IndirectRuleSet
    end
end
```

For each rule in the RuleSet, along with the rule components, we store the iteration number in which the rule was generated (*Iteration*), set of rules causing the rule (*Cause*), and whether the rule is contradiction or not (*Contradiction*).

Step 5: Extraction of crucial information
In this step we extract the following information by using the data collected in the previous step:

Analysing Individual Rules: Here we try to understand the impact of each policy rule on flow security. We count the number of contradictions caused by each rule in the policy. Higher the number for a rule, larger is its potential to cause harm.

Analysing Domains: Here we study each domain in the policy and count the contradictions caused by them. The domains are then ranked based on this count. This information helps the system developers to understand the priorities that should be given while developing the processes in those domains.

Analysing Indirect Accesses: As seen in the earlier sections, a one-level indirection between a subject-object pair is caused by chaining of 3 accesses. For each such indirection, we store the rule corresponding to the second access as the causing rule. This helps in generating the complete sequence of rules causing a particular indirection. We can use this procedure recursively to determine a complete sequence of original policy rules causing any multi-level indirection.

Remarks: Given a general access matrix model [8] along with the assertion that a certain subject s can acquire a right ‘x’ on object o, it is of interest[2] to generate a command/rule sequence that could lead to the new state from the original state of the access matrix. From the given access rules, FlowConSEAL generates the new *rights* acquired by processes along with the sequence of rules required for realizing that *right*.

5 Experimental Analysis and Illustration

We have implemented FlowConSEAL using Python 2.7. Our implementation and experiments are performed on Ubuntu 16.10 running on a virtual machine configured with 64GB RAM. We demonstrate the effectiveness of FlowConSEAL on two policies, the Reference policy (`refpolicy- 2.20170805`)[3] which is the base policy used by all the Linux distributions for developing their SELinux policies and the SEAndroid policy provided as part of the Android Open Source Project (AOSP) tree in Android 7[4].

5.1 Analysis of Policies by FlowConSEAL

Here we provide a brief[5] analysis of the above two polices obtained through FlowConSEAL as depicted in Table 1.

Number of Types, Object Classes and Permissions: SELinux doesn’t have any predefined types whereas object classes and associated permissions are predefined. Policy writers define types based on the resources and services they want to confine and the overall security goals. Larger number of types help specifying fine-grained rules. But with increase in types, associated rules also increase drastically and the policy management becomes difficult. As general purpose Linux systems provide comparatively large number of services and resources, naturally, SELinux Reference policy contains larger number of types, object classes and permissions than Android’s AOSP policy.

Number of Canonicalized `allow` and `neverallow` Rules: Our tool parses the policy only once and stores the canonicalized `allow` and `neverallow` rules separately. All further processing is done on these rules. Hence performance of the tool depends on the number of `allow` and `neverallow` rules.

Number of Iterations: Number of iterations required to generate all possible indirect rules indicates the levels of indirect flows present in the policy. AOSP policy conforms to RWFM check in its second iteration i.e., it contains only single level of indirection. However, note that the Reference policy contains two levels of indirection.

[2] Note that it is a specific problem instance rather than the ‘safety problem’.

[3] https://github.com/TresysTechnology/refpolicy.

[4] https://android.googlesource.com/platform/manifest/.

[5] A full extended report on FlowConSEAL is available at http://isrdc.iitb.ac.in/reports/isrdc-tr-2018-rks-rbs-selinux-static.pdf.

Table 1. Experimental analysis

	Reference policy	AOSP policy
Policy size	3.3 MB	521.6 KB
Number of types	1276	612
Number of object classes	127	63
Number of permissions	447	286
Number of canonicalized allow rules	10374	24418
Number of canonicalized neverallow rules	22893	2369117
Number of iterations	2	1
Number of indirect rules generated	232189	244466
Number of contradictions	1545	11529
% of indirect allows that are contradictions	0.665	4.715
% of neverallows that are contradictions	6.75	0.486
Execution time	41 min	3 min

Number of Indirect and Contradictory Rules: The tool identifies all possible indirect information flows. As we can see in the table for both the policies, these rules are in hundreds of thousands in number. In order to avoid false positives and reduce these rules to a manageable subset, we consider only the contradictory rules. Larger the number of contradictions, weaker is the security of the policy. From the table, the Reference policy has lower number of contradictions even though it has large number of indirect flows.

Percentage of Indirect Allows that are Contradictions: This factor indicates the extent to which indirect rules can be exploited to cause policy violations. Theoretically, for perfect security, this number should be zero. The Reference policy with only 0.665% of its indirect allows causing contradiction, prove to be much more stronger against policy violations using indirect allows.

Percentage of `neverallows` that are Contradictions: This factor indicates the extent of potential policy violations. Larger the percentage, weaker is the security. From the table, we can notice that in case of the AOSP policy only 0.486% of `neverallows` can be violated using the indirect rules. Hence comparatively, this is a well written policy with respect to information flow.

Execution Time: From the table, we can see that the execution time especially that for the Reference policy is considerably high. However, considering the large size of the policy and the size of the meta data being generated (10 GB in case of the Reference policy), and the fact that this analysis is performed only once, we can say the tool is quite useful.

Number of Contradictory Rules Generated by each Rule: This is useful for understanding the impact of each rule on the flow security. Tables 2 and 3 show the top 3 `allow` rules along with the number contradictions that they cause (Ctd_r Count) in the Reference and AOSP policy respectively.

Domain Ranking: Here the number of contradictions caused by each domain is counted. Using this, we can get the domains which have high potential to be exploited to gain an unauthorized access. Therefore subjects in these domains need to be designed and implemented carefully. From our experiments, we noticed that high ranking domains are mostly system processes and daemons which are trusted by the system. However, considering the large number of policy violations that we have observed, it is important that these contradictions are carefully analysed, and the processes running in these domains are thoroughly verified to be safe against any attacks leading to those contradictions.

Table 2. Ctd_r count (Reference Policy)

Allow rule in the RuleSet	Ctd_r Count
`systemd_tmpfiles_t device_t:lnk_file write`	468
`udev_t device_t:lnk_file write`	468
`getty_t devlog_t:sock_file read`	146

Table 3. Ctd_r count (AOSP Policy)

Allow rule in the RuleSet	Ctd_r Count
`init cgroup:dir read`	5166
`init urandom_device:chr_file read`	5166
`system_server cgroup:dir read`	927

6 Related Work

Preventing unauthorized information flows (IF) is crucial for ensuring security. Uzun et al. [9] propose a method for preventing unauthorized IF in access matrix model based DAC systems. They identify one-step transitive flows and eliminate them by revoking necessary permissions; FlowConSEAL not only identifies multi-level indirections, but also provides a sequence of rules that lead to each indirect flow.

Over the years, several tools have been developed to understand and analyse SELinux policies [2]. SETools [10] is one of the commonly used collection of tools. It provides several tools for searching rules, comparison of policies and, IF analysis which is limited to listing all the flows between domains specified by the user. Unlike FlowConSEAL, it does not support verification of flows against a security model or checking if the indirect flows are contradicting any `neverallow` rules. PAL [11] is a logic programming tool that supports SELinux policies by first translating them into a logic program. The user needs to construct appropriate queries to analyse the policy. Thus, the onus is on the user to come up with properties as queries which is not an easy task. In comparison, FlowConSEAL yields the possible indirect flows, contradictions etc., without any user intervention.

Gokyo [12] analyses integrity of the "Example policy" using manually specified high integrity types as Trusted Computing Base (TCB). The tool considers only one level indirections between TCB types and non-TCB types, and checks for conflicts between the integrity goal and the policy rules. Similarly, SCIATool [13] also analyses integrity conflicts between TCB and non-TCB entities using Colored Petri-nets.

Several visualization-based SELinux policy analysis tools [14–16] have been developed to help policy writers to better understand the policies. Gove [14] presents a tool for understanding and comparing SELinux/SEAndroid policies by creating graph representations. SPTrack [16] helps visualize SELinux policies as well as its attack logs to track IF. SEGrapher [15] generates cluster-based focus-graphs of policies based on clustering.

Several tools have been developed specifically for SEAndroid policy analysis. [17] analyses SEAndroid policies from Android 5.0 devices from a number of OEMs and identify patterns of common problems. SELint [18] is an extensible tool built to help policy writers in writing secure SEAndroid policies. It's built-in plug-ins mainly focus on making a policy more compact and readable and identify potentially dangerous rules by assigning a risk score to each rule. The risk score of a rule is computed based on the risk level and trust level of the rule components whose values are policy-dependent and need to be manually configured by the policy writers. A semi-automated tool to identify potential SEAndroid policy misconfigurations is presented in [19]. EASEAndroid [20] analyses SEAndroid policies using large-scale semi-supervised learning.

To sum up, FlowConSEAL provides a succinct analysis of SELinux policies and enables the user to decide on benign and vulnerable indirect flows. One distinct characteristic of FlowConSEAL is that it works like a "pushbutton" tool unlike others that need user supplied abstraction of queries/properties.

7 Conclusions

In this paper, we have presented an efficient method and a tool FlowConSEAL to analyze information flows in SELinux/SEAndroid policies. Our method verifies the consistency of the policies in terms of indirect flows and helps in identifying potential vulnerabilities. Furthermore, we also rank the policy rules and domains based on their potential to misuse information. The tool enables the policy writers to understand the security loopholes in the policy and handle them appropriately to protect systems against flawed and malicious applications. The experimental results demonstrate the effectiveness of the method. One of the distinct advantage of using RWFM model is its capability to capture all the influencers succinctly.

References

1. Loscocco, P., Smalley, S.: Integrating flexible support for security policies into the linux operating system. In: USENIX Annual Technical Conference, pp. 29–42 (2001)
2. Eaman, A., Sistany, B., Felty, A.: Review of existing analysis tools for SELinux security policies: challenges and a proposed solution. In: Aïmeur, E., Ruhi, U., Weiss, M. (eds.) MCETECH 2017. LNBIP, vol. 289, pp. 116–135. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59041-7_7
3. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux security modules: general security support for the linux kernel. In: USENIX, pp. 17–31 (2002)
4. Kumar, N.V.N., Shyamasundar, R.K.: A complete generative label model for lattice-based access control models. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 35–53. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_3
5. Denning, D.E.: A lattice model of secure information flow. CACM **19**(5), 236–243 (1976)
6. Bell, D.E., LaPadula, L.J.: Secure computer systems: Mathematical foundations. Technical report MTR-2547-VOL-1, MITRE CORP BEDFORD MA (1973)
7. Biba, K.J.: Integrity considerations for secure computer systems. Technical report MTR-3153-REV-1, MITRE CORP BEDFORD MA (1977)
8. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. Commun. ACM **19**(8), 461–471 (1976)
9. Uzun, E., Parlato, G., Atluri, V., Ferrara, A.L., Vaidya, J., Sural, S., Lorenzi, D.: Preventing unauthorized data flows. In: Livraga, G., Zhu, S. (eds.) DBSec 2017. LNCS, vol. 10359, pp. 41–62. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61176-1_3
10. TresysTechnology: Setools: Policy analysis tools for SELinux. https://github.com/TresysTechnology/setools Accessed Nov 2017
11. Sarna-Starosta, B., Stoller, S.D.: Policy analysis for security-enhanced linux. In: WITS Proceedings, pp. 1–12 (2004)
12. Jaeger, T., Sailer, R., Zhang, X.: Analyzing integrity protection in the SElinux example policy. In: USENIX Security Symposium-Volume 12, p. 5 (2003)
13. Zhai, G., Guo, T., Huang, J.: SCIATool: a tool for analyzing SElinux policies based on access control spaces, information flows and CPNs. In: Yung, M., Zhu, L., Yang, Y. (eds.) INTRUST 2014. LNCS, vol. 9473, pp. 294–309. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27998-5_19
14. Gove, R.: V3SPA: a visual analysis, exploration, and diffing tool for selinux and seandroid security policies. In: IEEE VizSec, pp. 1–8 (2016)
15. Marouf, S., Shehab, M.: SEGrapher: Visualization-based SELinux Policy Analysis. In: Symposium on Configuration Analytics and Automation, SafeConfig (2011)
16. Clemente, P., Kaba, B., Rouzaud-Cornabas, J., Alexandre, M., Aujay, G.: SPTrack: visual analysis of information flows within SELinux policies and attack logs. In: Huang, R., Ghorbani, A.A., Pasi, G., Yamaguchi, T., Yen, N.Y., Jin, B. (eds.) AMT 2012. LNCS, vol. 7669, pp. 596–605. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35236-2_60
17. Reshetova, E., Bonazzi, F., Nyman, T., Borgaonkar, R., Asokan, N.: Characterizing SEAndroid policies in the wild. In: ICISSP, pp. 482–489 (2016)

18. Reshetova, E., Bonazzi, F., Asokan, N.: Selint: an SEandroid policy analysis tool. In: ICISSP, pp. 47–58 (2017)
19. Chen, H., Li, N., Enck, W., Aafer, Y., Zhang, X.: Analysis of SEAndroid policies: combining MAC and DAC in Android. In: ACM ACSAC, pp. 553–565 (2017)
20. Wang, R., Enck, W., Reeves, D.S., Zhang, X., Ning, P., Xu, D., Zhou, W., Azab, A.M.: EASEAndroid: automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In: USENIX Security Symposium, pp. 351–366 (2015)