2013 IEEE Security and Privacy Workshops

Invalidating Policies using Structural Information

Florian Kammüller
Middlesex University London
f.kammueller@mdx.ac.uk

Christian W. Probst
Technical University of Denmark
probst@imm.dtu.dk

***Abstract*—Insider threats are a major threat to many organisations. Even worse, insider attacks are usually hard to detect, especially if an attack is based on actions that the attacker has the right to perform. In this paper we present a step towards detecting the risk for this kind of attacks by invalidating policies using structural information of the organisational model. Based on this structural information and a description of the organisation's policies, our approach invalidates the policies and identifies exemplary sequences of actions that lead to a violation of the policy in question. Based on these examples, the organisation can identify real attack vectors that might result in an insider attack. This information can be used to refine access control system or policies.**

***Keywords*-Insider threats, policies, formal methods**

I. Introduction

Insider threats are a major threat to many organisations. Insiders have special privileges and knowledge, which not only enable them to perform many actions unobserved, but also make them interesting for attackers outside an organisation [1]–[4].

Even worse, insider attacks are usually hard to detect, especially if an attack is based on actions that the insider has the right to perform. In this case, finding the legal but maliciously intended action is as hard as finding the proverbial needle in a haystack.

A natural reaction to this kind of threat would be to try to ensure that after an attack one has sufficient proof to identify the attacker, and to remedy the actions of the attack. The problem however is to decide what to log. Too much information makes it impossible to identify important pieces (the above mentioned haystack), but too little information makes it unlikely that there is a needle to find.

In this paper we present a step towards detecting possible insider attacks by invalidating policies. Invalidating policies is not a new idea. What is new in our approach is that we take structural information into account for guiding the invalidation. This structural information will usually be in form of a model of the part of the organisation being analysed, *e.g.*, a system model representing the organisation's infrastructure, or a representation of a workflow. Based on a model of the organisation and a description of the organisation's policies, our approach identifies exemplary sequences of actions that lead to a violation of the policy in question. This approach is inspired by model checking and its counterexample guided abstraction refinement.

Using the exemplary sequences of actions found, the organisation can identify real attack vectors that might result in an insider attack. This information can be used to refine access control systems, policies, logging, or combinations hereof.

Both approaches presented here are based on some formalisation of policies. The formalisations and our results show that logical policy formalisation can be a good starting point to invalidate policies, and may also be taken as a starting point for a systematic design of, *e.g.*, access control or policy systems.

The rest of this article is structured as follows. After discussing policies and their roles for insider threats in the next section, we present two approaches to invalidating policies based on structural information: the first uses system models (Section III), the second uses workflows (Section IV). After a discussion of related work in Section V, we conclude in Section VI.

II. Policies

Policies describe admissible or inadmissible behaviour in organisations. As such they also are (if complied to) an obvious means for regulating insider threats; at the same time research indicates that there is an upper limit to the number of policies that employees will comply with [5].

Since insiders will usually have good knowledge of policies and may use this knowledge combined with their respective access rights granted by those policies to circumvent regulations, the policies are a good starting point to explore insider attack possibilities. When we thus focus on the policies, the models of the actual workflows become a parameter whose level of detail we use to explore to what extent a given policy can be violated. We consider policies as restrictions on dynamic state based system models and additionally as logical constraints describing workflows of organisations. Full state exploration as well as simple propositional logic evaluation serve to invalidate these policies and thus exhibit attacks.

The two sections Section III and Section IV currently each use their own policy language. We are in the process of unifying these two branches, and to investigate their compatibility with policy languages such as XACML.

DOI 10.1109/SPW.2013.36

III. System-model based Policy Invalidation

In this section we discuss how to use structural information from system models [6], [7] to invalidate policies. The system models we consider are based on ExASyM [6].

As argued above, structural information can be instrumental in strengthening the invalidation of policies. The system models we consider describe especially access control specifications of the organisation in question, and these specifications can be used in the invalidation process to identify insiders that might have performed a certain action. It should be noted that for many actions we can not clearly identify a certain actor as source; instead, the source of an actions will most often only be describable by a set of capabilities, *e.g.*, a certain key or access rights. We plan to explore this in future work.

A. Example

In Figure 1 we show an example for the infrastructure of an organisation based on [6]. The infrastructure consists of two layers, the building and the computer network, some nodes of which are collocated, for example the computers and the offices. Most rooms in the building are controlled by some form of access control, the specification of which, together with our graph-based abstraction of the model is shown in the right hand side of Figure 1. For a more detailed discussion of the underlying model see [6].

Assume a policy that prohibits data of a certain type, *e.g.*, sales data, to leave the organisation. In the example this would mean that the data is not allowed to reach either of the nodes `WWW` or `OUTSIDE`.

B. Invalidating Policies based on System Models

The invalidation of policies based on the system model proceeds by negating the policy, and then trying to establish a series of actions that would result in a system state that fulfills the negated policy. At the same time this system state violates the original policy, thereby giving a counterexample for the policy. As discussed above, the obtained counterexamples can be used for refining the system model and its access control specification.

To illustrate this procedure a bit more in detail, let us start by negating the policy: data should go to `WWW` or `OUTSIDE`. Let us concentrate on the second case `OUTSIDE`. To practically realize this state exploration, we can use existing methods like model checking. We use a model checker specialized for multi agent systems, MCMAS [8]. We specify the goal as the negated policy, and then let the model checker find its way to a state that violates the policy. In a preliminary study [9] we have shown how this can be done for the current example (see Appendix for the MCMAS specification). To cut a long story short, we assume a secret file of user U. The state exploration reveals that there are several sequences of ExASyM actions that lead to this secret file being moved to `OUTSIDE`. If the user prints the secret file, it may rest on the printer and the janitor may pick it up because he has access rights to the server/printer room. Since the printout can go to the waste bin there is no secrecy restriction any more on this printout of the secret file and the janitor can transport it to `OUTSIDE`.

If the goal state is `WWW`, similar reasoning finds out that a process at `PC1` must have sent the file, where only the user can have started the (malicious) process. Subsequently the

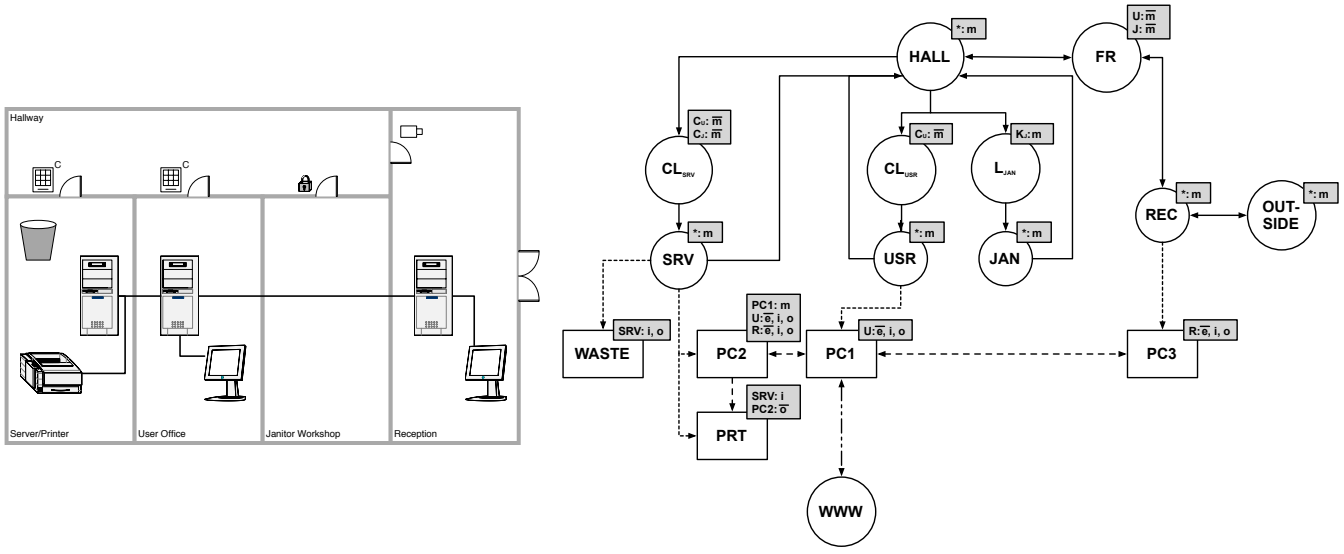

Figure 1. A simple example system and its representation as a graph in ExASyM [6], including access control annotations. The node `OUTSIDE` represents the physical world outside the organisation, and the node `WWW` represents the Internet as reachable by, *e.g.*, the mail server, here `PC1`.

analysis finds out that the process in question was brought in by the user from `OUTSIDE` or was received at `PC1`. This attack is especially interesting because it resembles a collection of attacks. The case of the malicious process being brought in from the outside represents the road-apple attack [10], the case of the process being received at `PC1` represents the case of the user receiving some malware by email or picking it up at a web page.

Usually model checkers are used to automatically prove that given models fulfill specified properties by a complete state exploration of the model. However, one very useful characteristic of model checking is that it produces counterexamples in case the examined property does not hold. These counterexamples are sequences of state changes representing the path leading from an initial state into a state violating the specified property. This quality we exploit here for invalidating policies. By negating the desired policy we use the complete state exploration of a model checker to produce a counterexample serving us as the key to refining the policy. A possible refinement just based on the first attack path to `OUTSIDE` could involve a refined lock mechanism at the door to the server room that would activate time delays on access for the janitor and user alerts as pickup reminders when secret files are printed.

IV. Workflow-based Invalidation

In this section we apply a similar idea as in the previous section to the problem of invalidating policies that address workflows. As before, we have a mechanism for describing policies, and we use the additional structural information from the workflow to guide the invalidation of the policies.

A. Example: DC Insider Attack

Some years ago, the District of Columbia (DC) was target of an insider attack launched by one of their employees with the help of colleagues to undergo the policy that was in place to avoid check refund frauds [11], [12]. This employee was known as a skillful and reliable person working with IT, and used her position and knowledge of the real estate tax refund policies to undergo the security perimeter and cash in bogus check refunds.

She was also involved in setting up the scene so that their frauds would remain undetected: since costs for a new, integrated Tax System had already used up the planned resources, the insider contributed to the decision to leave out her department that handled real estate tax refunds. As a result, she could cash in bogus real estate tax refunds for fictitious parties, often collaborating with colleagues or their partners. There were a few simple policy restrictions in place that the attackers exploited to remain undetected:

- Checks below a $\$40000$ threshold did not require a supervisor's approval, and
- there was no test in place to verify whether a check had already been cashed, so checks could be cashed-in more than once.

B. Invalidating Policies based on Workflows

We illustrate now that a simple logical formalization does suffice to make these fairly intuitive attack policies easily detectable. To formally describe the policy of the case study that has been tampered with, we introduce a datatype for checks entailing the addressed department, the recipient of the money, and the cash sum.

$$\text{check} \quad : \quad \text{department} \times \text{recipient} \times \text{sum}$$

The parts of the policies are as follows:

- *supervise* : *check* $\rightarrow$ *bool* is an abstract predicate denoting that a check refund must get the approval of a supervisor,
- *own-dept-exempt* $x \equiv x.\textit{department} = \textit{real-estate}$ is a concrete predicate defining which checks are exempt from checking because they are addressed to the real estate tax department,
- *threshold-exempt* $x \equiv x.\textit{sum} < 40000$ is a concrete predicate encoding that checks with a sum below 40000 need not be checked.

Finally, the current policy can be expressed simply as the following combination of these three predicates mandatory for all cashed checks of the DC tax department.

$$\begin{aligned} \text{policy } x \quad \equiv \quad & \text{own-dpt-exempt } x \vee \text{threshold-exempt } x \\ & \vee \text{ supervise } x \end{aligned}$$

To express the global policy we assume a concrete set of "cash-ins" and then quantify the policy over all checks in the set. The operator $\mathbb{P}(S)$ denotes the powerset of a set S.

$$\begin{aligned} \text{cash-ins} \quad &: \quad \mathbb{P}(\text{check}) \\ \text{Policy} \quad &\equiv \quad \forall x \in \text{cash-ins. policy } x \end{aligned}$$

C. DC Insider Attack Example Analysed

The logical expression of policies as discussed in Section II provides a tool to invalidate them. We have to analyse the ways how a fake check can pass the policy; to do so, we simply assume a fake check:

$$\textit{fake} \quad : \quad \text{check}$$

Then the following three simple properties express three ways of invalidating the policy.

$$\begin{aligned} &1. \quad \text{own-dept-exempt } \textit{fake} \quad \Rightarrow \quad \text{policy } \textit{fake} \\ &2. \quad \text{threshold-exempt } \textit{fake} \quad \Rightarrow \quad \text{policy } \textit{fake} \\ &3. \quad \text{supervise } \textit{fake} \quad \Rightarrow \quad \text{policy } \textit{fake} \end{aligned}$$

The first two invalidation properties correspond to the actual attacks that happened in the DC case. The third one has *not* taken place but represents another loophole for an

insider attack: by conspiring with an insider that may act as supervisor, this additional invalidation property exhibits an additional insider threat with respect to this policy.

The analysis of the invalidation of the policy is somewhat ad hoc. It has been shown that policies can be formalized in First Order Logic (FOL) [13]. However, using Higher Order Logic (HOL), we can schematize and reason on a higher level about invalidation of policies thus rendering the above ad hoc logical analysis as a generic tool.

Although Higher Order Logic is generally undecidable, cutting edge tools like Isabelle [14] or Coq provide sufficient automation to support applications like this example to a degree of automation.

D. Detecting Multiple Cash-ins

The policy and analysis presented above lack one possible attack, a flaw that has been exploited in the real case. Since check cash-ins exempt from supervisor control were not scrutinized properly, checks could be cashed in more than once. This flaw of the actual workflow persists as a weakness also in our logical specification. Since cash-ins of checks are represented as *sets* of checks, their cash-in is not audited as in real life in a sequence of actions (for example by adding time stamps to cash-ins). Therefore, a double cash-in cannot be noticed in the abstract specification.

In order to invalidate the policy, we refine it to a more concrete specification closer to the physical reality of workflows in a tax department. We use *sequences* of check cash-ins as a representation of the cash-flow of the tax department.

Double cash-ins can now be easily detected by auditing the sequence of cash-ins and while checking for double occurrences of the same check. The function count $x \in s$ is used to return the number of occurrences of x in sequence s.

$$\begin{array}{lcl} \text{cash-ins} & : & \text{sequence(check)} \\ \text{double-cash-ins} & \equiv & \exists\ c : \text{check. count } c \in \text{cash-ins} > 1 \end{array}$$

Concerning the automated invalidation analysis of a refined policy, we propose action-based verification tools. A powerful formalism like HOL is – unlike FOL – fully capable of modeling datatypes like sequences and even provide some automated support like decision integrated procedures in tools like Isabelle [14]. Appendix VII-3 contains a complete Isabelle-formalisation of the above described invalidation example. But, for insider attacks, we inherently concentrate on organisational security since insiders are by definition part of an organisation. Therefore, when scrutinizing policies for invalidation, we typically consider workflows of organisation. These are most naturally expressed as systems of actions not as datatypes.

This final example of multiple cash-ins, illustrates nicely that an action-based model of the real estate tax department is more suitable for an analysis of action sequences. The representation as sketched in the refined specification above, is naturally amenable to an analysis in a specification formalism like ExASyM. As a preliminary result of this study into modeling and analysing insider attacks using logical and mechanized analysis techniques, we propose an integration of purely propositional logics policy statements and action-sequence based methodologies.

V. Related Work

Popescu *et al.* [15] provide a policy engine for distributed objects implemented in their system Globe. This work is interesting from our perspective because it addresses the gap between abstract policy specifications and their actual implementation as a service in a distributed object system. In the context of model driven engineering similar goals to ours have been pursued by Mouelhi *et al.* [16] also regarding in particular the relationship between specification and testing presenting a generic security meta-model that can be used for early consistency checks in the security policy. Breaux *et al.* [17] consider semi-formal techniques for analysis and tracing of legal requirements. Gofman *et al.* [18] implement policy checking into a virtual machine mechanism.

Generally, the approach of using invalidation is reminiscent of the counterexample guided abstraction refinement approach in model checking [19] commonly called CEGAR feedback loop. This abstraction process uses an invalidation of an abstraction of a concrete specification to gradually approximate a faithful model checking representation. Conversely, we use the concrete level to invalidate abstract specifications representing a security policy.

The work by Dimkov *et al.* [7] is relevant as it also uses an invalidation procedure to derive attack scenarios.

VI. Conclusion

In this paper, we have presented an approach to supporting invalidating security policies with structural information from organisational models. These two possibilities have been illustrated on examples: the classical janitor example and a insider attack case study (DC real estate tax fraud). Techniques for the automated analysis have been shown and discussed.

One issue with our proposition for invalidating policies for insider attacks is the need for a common framework possibly compatible with a widely known policy language like XACML. This should not pose major problems but must be inlined with a series of representative case studies to ensure that the designed common policy language will be sufficiently expressive to cover common scenarios. The technique of invalidation has been demonstrated to be suitable for case studies from insider attacks. We believe that this particular domain profits from the invalidation technique more than general policies since the insider attacks are centered on breaking in from within, *i.e.*, not violating the usual rules.

VII. Acknowledgments

Part of the research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318003 (TREsPASS).

We would like to thank the anonymous referees for their constructive comments that helped to improve the paper.

References

[1] C. W. Probst, J. Hunker, D. Gollmann, and M. Bishop, Eds., *Insider Threats in Cybersecurity*. Springer, 2010.

[2] S. Stolfo, S. Bellovin, S. Hershkop, A. Keromytis, S. Sinclair, and S. W. Smith, Eds., *Insider Attack and Cyber Security: Beyond the Hacker*. Springer, 2008.

[3] E. Cole and S. Ring, *Insider Threat: Protecting the Enterprise from Sabotage, Spying, and Theft*. Elsevier, 2006.

[4] B. T. Contos, *Enemy at the Water Cooler*. Elsevier, 2007.

[5] A. Beautement, M. Sasse, and M. Wonham, "The compliance budget: Managing security behaviour in organisations," in *New Security Paradigms Workshop*, 2008.

[6] C. W. Probst and R. R. Hansen, "An extensible analysable system model," *Information Security Technical Report*, vol. 13, no. 4, pp. 235–246, Nov. 2008.

[7] T. Dimkov, W. Pieters, and P. Hartel, "Portunes: representing attack scenarios spanning through the physical, digital and social domain," in *Proceedings of the 2010 joint conference on Automated reasoning for security protocol analysis and issues in the theory of security*. Springer, 2010, pp. 112–129.

[8] A. Lomuscio, H. Qu, and F. Raimondi, "Mcmas: A model checker for the verification of multi-agent systems," in *CAV*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 682–688.

[9] F. Kammüller, C. Probst, and F. Raimondi, "Application of verification techniques to security – modelchecking insider attacks," in *SETECEC 2012*, 2012, keynote, To appear.

[10] S. Stasiukonis, "Social engineering the usb way," 2006, last accessed February 2013. [Online]. Available: http://www.darkreading.com/security/news/208803634/social-engineering-the-usb-way.html

[11] S. L. Pfleeger, J. B. Predd, and J. H. C. Bulford, "Insiders behaving badly," *IEEE Transactions on Information Forensics and Security*, vol. 5, no. 1, pp. 169–179, 2010.

[12] C. D. Leonnig and D. Nakamura, "D.c. tax scam total rises to $20 million, officials say," *Washington Post*, p. A01, 2007.

[13] J. Y. Halpern and V. Weissmann, "Using first-order logic to reason about policies," *ACM Transactions on Information and Systems Security*, vol. 11, no. 4, 2008.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer-Verlag, 2002, vol. 2283.

[15] B. C. Popescu, B. Crispo, A. S. Tanenbaum, and M. Zeeman, "Enforcing security policies for distributed objects applications," in *Security Protocols Workshop*, ser. Lecture Notes in Computer Science, B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe, Eds., vol. 3364. Springer, 2003, pp. 119–130.

[16] T. Mouelhi, F. Fleurey, B. Baudry, and Y. L. Traon, "A model-based framework for security policy specification, deployment and testing," in *MoDELS*, ser. Lecture Notes in Computer Science, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds., vol. 5301. Springer, 2008, pp. 537–552.

[17] T. D. Breaux and D. G. Gordon, "Regulatory requirements traceability and analysis using semi-formal specifications," in *19th Working Conference on Requirements Engineering: Foundations for Software Quality (REFSQ'13)*, 2013, to appear.

[18] M. I. Gofman, R. Luo, P. Yang, and K. Gopalan, "Sparc: a security and privacy aware virtual machinecheckpointing mechanism," in *WPES*, Y. Chen and J. Vaidya, Eds. ACM, 2011, pp. 115–124.

[19] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169.

Appendix

Appendix VII-1 show the MCMAS specification for the user, and VII-2 the specification for the janitor from [9] mentioned in Section III. Appendix VII-3, finally, contains the Isabelle-experiment we conducted to support our argument about workflow policy invalidation and refinement.

1) Specification of agent user:

```
Agent User
Vars:
  initialposition : { hall };
    -- The initial state is in the hall
    -- all moves are spelled out
  currentposition : { hall,pc,server };
    -- The current position
    -- the data is modeled flatly as a boolean flag
  print_secretfile: boolean;
end Vars
  Actions = { print, move, move1, move2 };
Protocol:
  currentposition = pc or currentposition = server : move;
  currentposition = hall : {move1,move2};
  currentposition = pc : {print};
end Protocol
Evolution:
  currentposition = pc if (currentposition = hall
                           and Action = move1);
  currentposition = server if (currentposition = hall
                           and Action = move2);
  currentposition = hall if ((currentposition=pc or
          currentposition=server) and Action=move);
end Evolution
end Agent
```

2) Janitor specification in MCMAS:

```
Agent Janitor
Vars:
  initialposition : { janitor };
  currentposition : { hall, server, janitor };
    -- the data is modelled flatly as a boolean flags
  has_secretfile: boolean;
end Vars
  Actions = { pickfromwaste, move, movej, movep };
Protocol:
  currentposition = janitor
        or currentposition = server : {move};
  currentposition = hall : {movej,movep};
end Protocol
Evolution:
  currentposition=server if (currentposition=hall
                             and Action=movep);
  currentposition=janitor if (currentposition=hall
                             and Action=movej);
  currentposition=hall if ((currentposition=server or
           currentposition=janitor) and Action=move);
  has_secretfile = true if (currentposition = server
                             and User.Action = print);
end Evolution
end Agent
```

3) DC Fraud in Isabelle:

```
theory DCfraud
imports Main
begin
datatype department = Dept string
datatype recipient  = Recp string
datatype sum = Sum nat
datatype check = Check department recipient sum
consts supervise :: check ⇒ bool
consts cash_ins :: check set
consts fake :: check

primrec get_nat_sum :: sum ⇒ nat ("#_")
where  "#(Sum n) = n"
primrec get_dept :: check ⇒ department  ("_.department")
where get_dept (Check d r s) = d
primrec get_recp :: "check ⇒ recipient"("_.recipient")
where get_recp (Check d r s) = r
primrec get_sum :: "check ⇒ sum" ("_.sum")
where get_sum (Check d r s) = s

definition own_dept_exempt :: check ⇒ bool where
"own_dept_exempt x ≡ x.department = Dept(''realestate'')"

definition treshold_exempt :: check ⇒ bool where
"treshold_exempt x ≡ ((#((x).sum)) < 40000)"

definition policy :: check ⇒ bool where
"policy x ≡ own_dept_exempt x ∨ treshold_exempt x
             ∨ supervise x"

lemma invalidate_one: own_dept_exempt fake ⟹ policy fake
by (simp add: policy_def)

lemma invalidate_two: treshold_exempt fake ⟹ policy fake
by (simp add: policy_def)

lemma invalidate_three: supervise fake  ⟹ policy fake
by (simp add: policy_def)

definition Policy :: check set ⇒ bool where
"Policy l ≡ ∀ x ∈ l. policy x"

lemma Policy_empty: "Policy {}"
by (simp add: Policy_def)

(* Refinement for Invalidation and base for CEGAR loop *)
consts cash_ins_seq:: check list

primrec count :: ['a, 'a list] ⇒ nat where
"count a [] = 0" |
"count a (b # l) = if (a = b) then Suc(count a l)
                   else count a l"

definition Policy_seq :: check list ⇒ bool where
"Policy_seq l ≡ ∀ x. x mem l ⟶ (count x l = 1 ∧ policy x)"

lemma no_zero_count_mem: "count a l > 0 ⟶ a mem l"
by (induct_tac l, simp+)

lemma no_double_cash_ins_in_seq_model:
"count c l > 1 ⟹ ¬ (Policy_seq l)"
apply (simp add: Policy_seq_def)
apply (rule_tac x = c in exI)
apply (rule conjI)
apply (simp add: no_zero_count_mem)
apply simp
done

definition policy_refinement:: check list ⇒ bool where
"policy_refinement l ≡ Policy(set l) ⇒ Policy_seq l"

lemma ex_double_cash_ins_invalidation_by_refinement:
"cash_ins_seq =
[Check (Dept (''realestate''))(Recp ''insider'')(Sum 100000),
 Check (Dept (''realestate''))(Recp ''insider'')(Sum 100000)]
⟹  ¬ (policy_refinement cash_ins_seq)"
apply (unfold policy_refinement_def)
apply simp;
apply (rule conjI)
apply (erule ssubst)
apply (simp add: Policy_def policy_def own_dept_exempt_def)
apply (rule no_double_cash_ins_in_seq_model)
apply (erule ssubst)
by simp

lemma Finite_set_list: "finite s ⟹ (∃ l. set l = s)"
apply (erule finite.induct)
apply simp
apply (erule exE)
apply (rule_tac x = "a # l" in exI)
by simp

lemma hd_lem[rule_format]: "l ≠ [] ⟹ hd l mem l"
apply (induct_tac l)
by auto

lemma double_cash_ins_invalidation_by_refinement_gen:
"∀ s. finite s ∧ s ≠ {} ⟶
(∃ l. set (l) = s ∧ Policy s ⟶ ¬(policy_refinement l))"
apply (rule allI, rule impI)
apply (erule conjE)
apply (drule Finite_set_list)
apply (erule exE)
apply (rule_tac x = "hd (l) # l" in exI)
apply (rule impI)
apply (simp add: policy_refinement_def)
apply (rule_tac c = "hd l" in no_double_cash_ins_in_seq_model)
apply (simp add: no_zero_count_mem_one)
apply (drule sym)
by (simp add: hd_lem)

end
```