PROPAGATING MODIFICATIONS TO MOBILE POLICIES¹

Ken Smith, Don Faatz, Amgad Fayad, Sushil Jajodia

The MITRE Corporation

7515 Colshire Drive, McLean, VA 22102-7508, USA

{kps, dfaatz, afayad, jajodia}@mitre.org

Abstract: Mobile policies provide a flexible framework for enforcing access controls in distributed applications. But what happens when a mobile policy needs to be modified or certain permissions from a policy have to be revoked? Since a mobile policy is attached to the data and travels with the data over the network, it can be tricky to propagate any changes to the policy. In addition, real-world constraints affect the formulation of the problem and imply a variety of propagation algorithms. In this paper, we propose different approaches for propagating modifications to mobile policies consistent with these constraints.

Keywords: access control, mobile policy, policy modification

1. INTRODUCTION

Many data-centric applications, such as e-commerce, scientific collaboration, and intelligence production, require computing environments that are highly distributed. Individual users can request access to data from

¹ This work was funded by the MITRE technology program under project numbers 51MSR871, 51MSR203, and 0702M630. The work of Ken Smith was also supported by NIH grant R01-MH64417-01 funded jointly by the National Institute for Mental Health and the National Science Foundation.

multiple locations, and individual data items may be widely distributed and represented in multiple forms. Mechanisms are needed that can achieve consistent access control policy enforcement across the elements of such a distributed environment.

Chapin, et al. [1,4], propose a *mobile policy* framework that allows policies to move through the distributed system, accompanying the data it is intended to protect. In this framework, policy administration is separated from policy enforcement: policy is specified and administered at only those elements of the distributed system authorized to do so (frequently the “owner” of the data), whereas any trustworthy component in the distributed computing environment can enforce the policy.

In this paper, we consider the problem of modifying a mobile policy after it has been deployed (the *mobile policy modification problem*). As data is propagated, the associated policy also propagates throughout the distributed network. A policy distribution graph is formed in which each node contains a copy of the policy (and its data). When that policy is modified, the new policy must be propagated to each node in the graph.

Consider the following example from the domain of neuroinformatics: A principal investigator (PI) at Site **A** performs a structural MRI scan of a subject’s brain as part of an experiment. The resulting raw MRI image must be post-processed to enhance its usefulness [2,10], and competing algorithms exist to do this. Therefore, the investigator sends his data to sites **B** and **C**, which have developed postprocessing algorithms to normalize the volume in a standard stereotaxic space, correct for movements during the scan, and label tissue types (e.g., gray matter, cerebrospinal fluid). The enhanced images are sent to a neuroscientist collaborating with the PI at Site **D**, who examines them for accuracy and uses the best set in their experiment.

Now assume policy **P** is associated with the original image and each of its derivatives. **P** addresses two issues:

- A consent form was signed stating that the subject’s privacy would be protected. This protection is achieved by a policy requiring removal of the name and other identifying information from the images header.
- The PI does not want to publicly release his images until an in-progress publication has been submitted. This delay is achieved by issuing a policy limiting access to named collaborators and pipeline providers for six months.

Policy **P** is then propagated with the data to each site in the distribution graph.

After the experiment has been conducted, however, three events occur:

1. The images are requested by a public database at Site **E** used to derive composite “atlas” images [7] reflecting characteristic features of

demographic slices of the general population. The images are donated, along with the attached policy preventing their use for 6 months.

2. The anticipated publication is subsequently delayed, and the PI changes the policy to limit access for 12 months. He would like to enforce this policy change with a high level of assurance to prevent data from being accessible to competitors.
3. A new governmental ruling expands the notion of privacy for publicly accessible brain scans; the public database must additionally require the removal or obscuring of all facial features from any images it accepts. This new policy carries no “assurance requirements”, but if it is not implemented on all copies the public database will delete its own copy.

In event 1, the policy graph is dynamically enlarged, and **P** is sent to node **E** along with the image. In event 2, a new policy **P**₂ originates at node **A** and must be propagated to each node in the expanded graph and installed with high assurance. In event 3, a proposed policy **P**₃ (capable of obscuring facial features) originates at node **E** and must be propagated to each node in the distribution graph; subsequent acknowledgments must be sent back to **E** describing the extent to which the policy is accepted.

This example illustrates not only the basic problem of the propagation of policy modifications, but it also illustrates the following real-world complexities potentially associated with it:

1. The policy graph itself may evolve concurrently with the propagation of policy updates.
2. Policy updates may be introduced at multiple points in the graph.
3. The policy may require a mechanism to elicit an acknowledgement of its acceptance or to enforce its implementation at remote nodes.

A variety of algorithms/solutions are implicated because different policy scenarios display other combinations of these complexities. While some algorithms correctly solve this problem for *any* combination of complexities, they entail a greater overhead (e.g., for knowledge maintenance), requiring the analysis of tradeoffs between algorithmic power and overhead.

We provide two major contributions in this paper: First, we present, discuss, and classify into a logical framework a set of solutions to the mobile policy modification problem. Second, we illustrate through examples how various security policy scenarios correspond to solutions in this framework.

A fair amount of literature is available for propagating updates to replicated files in a distributed environment [3]. Most of the algorithms developed for this purpose can be grouped into one of two classes. Those in first class share the philosophy that mutual consistency is of greater importance than availability and allow updates to be made only if mutual consistency of the replicated files can be guaranteed. This means that sometimes updates cannot be allowed anywhere in the system, seriously

restricting availability; therefore, these algorithms are not suitable for propagating policy modifications. The algorithms in the second class emphasize availability and allow updates to be made from anywhere in the system. Since this may result in independent updates, the focus of these methods is on developing strategies for detecting possible conflicts and ways of resolving them. In this paper, our focus is on different methods of propagating modifications to mobile policies.

This paper is organized as follows. Section 2 describes our problem model in more detail. Section 3 explores the various approaches for solutions to this problem. In Section 4 we briefly discuss several policy scenarios and their correspondance to the solutions identified in Section 3. Section 5 presents conclusions and addresses future work.

2. PROBLEM MODEL

We model an instance of the mobile policy update problem as a directed acyclic graph (DAG), the *policy distribution graph*. (Note that many such graphs may simultaneously cover a set of nodes, each pertaining to the distribution of a different datum. For simplicity, we focus on a single graph.) Each node in the graph represents a site with a copy of the distributed data and its policy. In this section, we first describe a single node in the graph, and then we present the model for the entire graph.

A *basic* node in the policy distribution graph consists of five parts:

- 1. Data store.** The data store contains a copy of the data and its current policy. Policy updates are installed in the data store.
- 2. Data store interface.** The interface executes the mobile policy module whenever the data is accessed. We do not discuss the alternatives for interface implementations in this paper. The reader is referred to [1,4].
- 3. Auxiliary knowledge store.** Nodes must also maintain certain auxiliary knowledge for solutions described in Section 3 to execute correctly. An example is a list of addresses of other nodes in the graph (e.g., the owner, immediate the node's immediate parents/children, or the entire graph). The size, complexity, and maintenance overhead for this knowledge forms a key tradeoff in evaluating the suitability of different solutions.
- 4. Message queue.** Any node can receive either of two types of messages: *updated policy* or *changed graph*. Messages are labeled with their type and their destination address. Messages are stored in receipt order in the message queue. Each node's queue has an external address; anyone in possession of it can send messages to that node.
- 5. Message queue process.** This process executes periodically (after a standard interval). The *basic* version of this process does the following:

```

Queue_process()
{
  retrieve next message;
  if (message.destination_address  $\neq$  my_address) forward;
  if (message.destination_address == my_address)
    {
      if (message.type == updated policy) install in data store;
      if (message.type == changed graph) install in aux. knowledge;
    }
  return;
}

```

$G_D(N, E)$ is the policy distribution graph for a datum D . N is the set of all nodes currently containing an instance of D and its policy in their data store; one node N_O is designated D 's *owner*. E is the set of edges between nodes in N . Every edge forms a directed arc between two nodes, representing the migration path taken by D and its policy. From N_O , all other nodes in N are reachable via some set of directed arcs. Cycles are not permitted, resulting in a DAG.

Across all nodes in G_D , a policy P is uniquely identified by L , its *lineage*. L is a string of node names that have modified P , beginning with the owner and arranged in order of update (left to right). For example, if node A owns D , the original lineage of D 's policy P is simply " A ". If B later updates P , B distributes the updated policy with a lineage of " $A.B$ ", appending itself to the end of the lineage string². Function $L = \text{lineage}(D, N)$ returns the lineage string L of the policy on datum D currently in force at node N .

Definition 2.1 [Policy consistency] A policy P is *consistent* in graph $G_D(N, E)$ iff $\{\forall (N_i, N_j \in N) \text{ lineage}(D, N_i) = \text{lineage}(D, N_j)\}$.

An alternate way to state the mobile policy modification problem is to observe that a policy P in graph $G_D(N, E)$ becomes *inconsistent* when P is updated at some node in N . The solutions in Section 3 restore policy consistency to G_D after such an update by propagating the updated policy to all other nodes in G_D .

3. APPROACHES TO PROPAGATING MOBILE POLICY UPDATES

As shown in the figure below, we decompose all possible solutions into four non-overlapping quadrants along two key dimensions: (1) the origin of

² Formalisms other than lineage could also be used, such as version numbers and version vectors [3]. We omit the details due to space limitations.

updates, and (2) the requirement of acknowledgments. Updates can originate from the owner only or from any node in the graph. Acknowledgment of the policy installation can be unnecessary or required. These quadrants categorize solutions by relative complexity. In addition, security policy scenarios can show an affinity for particular quadrants, as illustrated by the examples given in Section 4.

		<u>Acknowledgment Required</u>	
		No Ack	Ack
<u>Update</u> <u>Origination</u> <u>Point</u>	Owner Only	Quadrant I	Quadrant III
	Any Node	Quadrant II	Quadrant IV

3.1 Quadrant I Solutions

Quadrant I solutions address the simplest case: only the owner of the data can cause updates to the policy, and no acknowledgment is required. We describe three distinct approaches.

3.1.1 Direct Contact. In Direct Contact, the owner sends an *updated policy* message to each node in the graph whenever it updates a policy. It is initialized by placing “*updated policy (Datum, Policy-New)*” in the owner’s message queue. The owner’s message queue process then installs this updated policy locally, and, for each node N_i in the propagation graph of the datum, it sends an “*updated policy (Datum, Policy-New)*” message to N_i . Note that only the *owner’s* message queue process exhibits this behavior; all other nodes continue to execute the basic process given in Section 2.

Direct Contact is robust: Failure to propagate to node N is independent of failure to propagate to node M . Despite this outcome, Direct Contact requires substantial and potentially costly maintenance of auxiliary knowledge. First, the owner must maintain knowledge of the entire graph to contact each node. (Although not directly considered in this paper, it is worth noting that the owner node must maintain knowledge about the entire graph of **every** datum it has distributed. For a server, this body of knowledge would be quite large.)

In addition, to maintain accurate knowledge of the graph’s topology by the owner, every node in the graph must *itself* maintain a direct or indirect reference to the owner so it can communicate local graph changes. For example, when node N_j gives datum D to new child N_k , the fact that N_k has joined the graph must be made known to the owner. If N_j has the address of the owner, it can directly send it a *changed graph* message. Otherwise, N_j must send this message to the owner via a chain of recursive forwarding.

3.1.2 Child Contact. Policy updates in Child Contact are only communicated to a node's immediate children. It is also initialized by placing "*updated policy (Datum, Policy-New)*" in the owner's message queue; however, *every* node's message queue process is modified to send an "*updated policy (Datum, Policy-New)*" message to N_i for each direct child N_i .

The auxiliary knowledge requirements of Child Contact are much lower than those of Direct Contact. Each node must maintain pointers to its immediate children (readily available from when they were added), but global knowledge of the propagation graph is not maintained by any node. Because no global knowledge is required, there is no need to propagate *changed graph* messages or to maintain pointers back to the owner.

Three new issues arise with this algorithm, however. First, it can be more vulnerable to failure than Direct Contact; the entire subtree below a node could potentially miss a policy update if propagation to that node fails.

A second issue is the potential for redundant policy updates. Propagation through a DAG can incur redundant updates *exponential* in the length of the longest path through the DAG. For example, whenever two paths extend from a node N_1 to its descendant N_2 , all policy updates from N_1 will be seen twice at N_2 . This doubling will continue if there are two paths from N_2 to N_3 , and so on. While correctness is not endangered by these redundant updates, in the worst case Child Contact could degenerate into a self-inflicted denial of service (DOS) attack.

Three strategies for addressing redundant updates are as follows: (a) do nothing, (b) force a topological traversal, and (c) test the lineage of each updated policy message. Doing nothing is reasonable if the policy distribution graph is a tree or has few "join" nodes like N_2 with multiple parents. Topological traversal forces propagation to descend synchronously through the DAG by levels: Each update waits at join nodes until all its redundant siblings join it. This hesitation, however, can lead to impractically long waits if any update fails and requires significant knowledge maintenance (especially when the graph changes). In lineage testing, the message queue process compares the lineage of each new updated policy message to the lineage of the policy currently in force. If identical, the new message is discarded. This action solves the problem for general DAGs without forcing waits or requiring auxiliary knowledge; it also has a low execution overhead.

Lineage testing can also be extended to handle a third issue: *out-of-order updates* can be prevented by requiring the lineage of any updated policy message to contain the existing policy's lineage as a proper prefix.

3.1.3 Phone Home. Phone Home is a "pull" solution: responsibility for propagation is placed with the recipient nodes. The basic message queue

process is used at all nodes. In addition, the owner node has a “publishing space” (such as a website) for policy updates. At each non-owner node, a “pull-process” wakes up at a policy-determined interval and checks the publishing space for a new policy version. If one is found, it places it in its own message queue and sleeps until the next interval elapses.

Phone Home has the lowest auxiliary knowledge requirement of the solutions described in this paper: every non-owner node needs the address of the owner’s publishing space (e.g., a URL). This knowledge can simply be bundled in the policy itself and propagated with the policy. Thus, whenever the pull process wakes up, it asks the current policy module for the publishing space address of new policy versions. Zero local knowledge of the graph or any *changed graph* messages is required.

Two key issues for Phone Home are as follows: (a) policy expiration, and (b) responses to failure. Expiration is a *metapolicy* that forces policies to cut off access to their data after a set time period. Pull-based policy expiration can augment assurance: the access policy can be written to require the pull process to periodically produce proof of a check for a new policy. Unless presented by the deadline, access to the policy’s data temporarily “expires”.

Failure is important in the context of policy expiration: owner inaccessibility could cause data to inadvertently expire. Key failure points are the integrity of the address and the availability of the owner’s publishing space. However, modern web architectures routinely overcome such problems and achieve reasonably high availability through techniques such as redundant servers, mirror sites, and autoforwarding. If policy expiration is *not* used as a metapolicy, failure has a less drastic effect: policy updates may be delayed, but data usage continues under the old policy.

3.2 Quadrant II Solutions

In this quadrant, *any* node is permitted to update the policy, as illustrated by nodes A and E in the example in Section 1. This adds significant complexity to solutions: auxiliary knowledge requirements grow and lineages become more complex. We present four solutions; three are generalizations of the solutions presented in Section 3.1 and one is new. Due to space limitations, we omit certain details, which can be found in [11].

3.2.1 Mutual Direct Contact. In Mutual Direct Contact, *every* node directly contacts *every other* node with policy updates, generalizing Direct Contact. Like Direct Contact, this algorithm is simple and robust.

An issue faced by this approach, or *any* in which multiple nodes can update policy, is incomparable lineages. If a node receives an update that neither dominates nor is dominated by the current policy (i.e., “A.A.B” vs.

“A.A.C”), several strategies can be used: an arbitrary tiebreaker (e.g., earliest timestamp), a semantic criterion (e.g., the strictest policy), contact by a human system administrator, or some combination of these.

Mutual Direct Contact also requires *substantial* auxiliary knowledge: each node must have a list of all other nodes. As $|N|$, the number of nodes, grows, this auxiliary knowledge grows by $|N|^2$. The maintenance of this knowledge incurs a significant overhead. Graph consistency problems can occur as well: two nodes can potentially add children that are mutually unaware of the other. Remedies, such as a global locking protocol for graph changes, require even greater overhead. We conclude that Mutual Direct Contact is only suitable for moderately small and stable graphs.

3.2.2 Local Contact. Local Contact is a generalization of Child Contact in which each node tells its neighbors (parents and children) about updated policy; this knowledge radiates outward from the node making the change. Each node maintains knowledge of only its immediate neighbors, obviating the large auxiliary knowledge maintenance overhead of Mutual Direct Contact.

Although Local Contact is susceptible to redundant, out of order, and even cyclic policy updates, the metapolicy of only accepting updates whose lineage contains the current policy as a proper prefix (coupled with a resolution strategy for incomparable “ties”) will resolve such problems. Such redundancy can actually make Local Contact robust, in that updated policies can travel several paths, compensating for the failure of any one route.

The only major issue for Local Contact is a vulnerability to graph partitioning (e.g., due to failure of one or more edges). Policy islands may form which continue to evolve independently. The subject of operation during network partitioning and restoration has been addressed thoroughly in distributed database literature [3].

3.2.3 Phone Everyone. Phone Everyone is a generalization of Phone Home; every node periodically polls every other to receive policy updates since the last polling cycle. A large maintenance overhead for auxiliary knowledge and graph consistency occurs, as in Mutual Direct Contact. Note also that policy expiration has much less assurance benefit than in Phone Home: a node must successfully contact every other node to provide “proof”. Even willing nodes would have trouble complying due to inconsistencies in the graph. Phone Everyone has little to recommend it.

3.2.4 Owner-Coordinator. In Owner-Coordinator, the owner coordinates all policy changes. The owner is sent all policy updates and periodically uses Direct Contact to broadcast the most recent policy to all nodes in the graph. Owner-Coordinator has roughly the same auxiliary

knowledge requirements of Direct Contact: the owner maintains knowledge of the entire graph and all graph changes must propagate back to the owner.

Issues for Owner-Coordinator are: (1) vulnerability to owner failure; (2) non-owners may miss intermediate policy versions, which could be addressed by permitting nodes to “phone home” instead of waiting on the broadcast; and (3) less vulnerability to policy islands and redundant updates.

Owner-Coordinator (and variants) maps well onto a web environment in which a server may be reliably reached by any node. It is a powerful approach when significant owner control must be retained, despite the ability of any node to update the security policy.

3.3 Acknowledgments for Quadrants III and IV

Quadrants III and IV address the need for some policies to receive acknowledgments that a policy update has actually been received and installed. As illustrated by the example in Section 1, a policy may use receipt of acknowledgments as a means of assurance that the policy has been put in force across the entire distribution graph or lack of acknowledgments as a signal to take some compensating action.

Implementing acknowledgments is straightforward. As described in Phone Home, we can assume any policy in Quadrants III and IV contains the address of the data owner. At the completion of installation, the policy code executes and sends an acknowledgment directly to the data owner. No extra knowledge is required, and the acknowledgment message does not significantly alter any of the algorithms previously described.

4. CORRESPONDENCE TO POLICY SCENARIOS

In this section, we describe several policy scenarios and show their correspondence to the strategy framework laid out in Section 3.

4.1 Neuroinformatic Example

We return to our motivating example in Section 1, which illustrates how subtle policy variations can require solutions from different quadrants. In its initial statement, a low-overhead Quadrant I solution (e.g., Child Contact or Phone Home) could be used. The extra assurance required by the policy update (P_2) might lead to the use of Phone Home with expiration, which is still a Quadrant I solution. However, the introduction of P_3 by the government suddenly changes the problem into one only solved by a more expensive Quadrant IV solution.

4.2 ORCON

ORCON is the intelligence community term for originator control. When ORCON access control is specified, only users or groups specified by the originator of the data are allowed access. If any additional users or organizations require access to the data, prior consent of the originator is required. ORCON requires the use of Quadrants I and II (since all changes must come from the originator). Only Quadrant II applies in the likely event that the originator would approve of the guarantee that every node receives the updated policy. In this case, if the originator does not receive an acknowledgment of receipt from a node, the originator resends the updated policy repeatedly until an acknowledgment is received.

4.3 NOFORN

NOFORN is a marking used by the U.S. DoD/Intelligence community to indicate that access to data requires U.S. citizenship. Let us consider the following example: Suppose site C_i modifies the policy of document D from NOFORN to NOFORN/REL CAN. Suppose further that another site in the graph, say site C_k , wants to share document D with a Canadian site; there are two possible cases to consider: (1) site C_k already has the new policy allowing release to Canadian citizens, which would be the result of a previous propagation of the new policy to site C_k by any strategy in any quadrant; and (2) site C_k did not yet receive the new policy. Here, site C_k does not know if a policy allowing REL CAN is available. In this case, site C_k can proactively “phone everyone” (Quadrant II) to determine whether such a policy exists instead of waiting for it to arrive. Since the “phone everyone” approach requires substantial overhead, it should only be used for high-priority events.

4.4 Certificate Revocation Lists

We provide an example from public key infrastructures (PKI). When a certificate is issued to a subject by a PKI certification authority (CA), it is expected to be in use during its entire period of validity. However, for various reasons (e.g., change of name or employment), a certificate may become invalid prior to the expiration of its validity period, in which case the CA needs to revoke that certificate and ensure that it is propagated to the users of the public key of that subject. In X.509, each CA issues a list, called a certificate revocation list (CRL), of certificates that are no longer valid [6]. CRLs are issued on a regular basis (e.g., hourly, daily, or weekly) and posted on the directories that offer a verification service to the public key users.

Here, the standard practice of directories is to use Phone Home with no acknowledgments (Quadrant I) to retrieve the CRLs from the CAs.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a framework for the mobile policy modification problem and suggested a number of solutions that correctly propagate modifications in a distributed environment. In addition, we provided a categorization of the solution space and illustrated through examples how various policy scenarios correspond to this categorization.

In this work, we have assumed that the new policy simply replaces the existing policy. Issues such as revising existing policies, merging new and existing policies, resolving conflicts, and criteria for comparing different solutions will be addressed as part of our future work.

6. REFERENCES

1. S. Chapin, S. Jajodia, and D. Faatz, "Distributed Policies for Data Management Making Policies Mobile," *Proc. 14th IFIP 11.3 Working Conference on Database Security*, Schoorl, Netherlands, August 2000.
2. D. L. Collins et al., *Automatic 3D model-based neuroanatomical segmentation*, Human Brain Mapping, Vol. 3, 1995 pages 190-208.
3. S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a partitioned network: a survey," *ACM Computing Surveys*, Vol. 17, No. 3, September 1985, pages 341-370.
4. V. Doshi, A. Fayad, S. Jajodia, and R. Maclean, "Using Attribute Certificates and Mobile Policies in Electronic Commerce Applications," *Proc. 16th Annual Computer Security Applications Conf.*, 2000, pages 298-307.
5. S. Farrell and R. Housley, *An Internet Attribute Certificate Profile for Authorization*, PKIX Working Group, Internet-Draft, March 2000.
6. R. Housley, W. Ford, T. Polk, and D. Solo, *Internet Public Key Infrastructure - X.509 Certificate and CRL profile*, RFC 2459, January 1999.
7. J. Mazziotta et al., "A Four-dimensional probabilistic atlas of the human brain," *Journal of the American Medical Informatics Association*, Vol. 8, No. 5, Sept/Oct 2001, pages 401-430.
8. C. J. McCollum, J. R. Messing, and L. Notargiacomo, "Beyond the Pale of MAC and DAC - Defining new forms of access control," *Proc. IEEE Symp. on Security and Privacy*, 1990, pages 190-200.
9. P. Samarati, P. Ammann, and S. Jajodia, "Maintaining Replicated Authorizations in Distributed Database Systems," *Data & Knowledge Engineering*, Vol. 18, No. 1, February 1996, pages 55-84.
10. J. Sled, A. Zijdenbos, A. Evans, "A comparison of retrospective intensity non-uniformity correction methods for MRI," *Springer Lecture Notes in Computer Science*, Vol. 1230, pages 459-464.
11. K. Smith, D. Faatz, A. Fayad, and S. Jajodia, *Propagating modifications to mobile policies*, Technical Report, The MITRE Corp., 2001.