Enforcing the Principle of Least Authority on Desktop Applications Using Implied Authority

Brett Cannon
University of British Columbia
201-2366 Main Mall
Vancouver BC V6K 4A1
drifty@cs.ubc.ca

Eric Wohlstadter
University of British Columbia
wohlstad@cs.ubc.ca

1. INTRODUCTION

Desktop client applications are a popular form of software which interact heavily with both local and remote resources, such as local disk and network resources. This is both a benefit in terms of the rich features desktop clients (DCs) can provide, but also a potential security risk. Resources must be handled by the application appropriately to ensure confidentiality and integrity of the user's sensitive information.

A potential way to mitigate resource misuse would be to follow the *Principle of Least Authority* (POLA):"Every program ... should operate using the least set of privileges necessary to complete the job" [4]. This could be done by sandboxing a DC using an application–specific access control policy; then the application could only use those resources necessary for its operation. In the case of a security issue, any negative effects would be limited.

In previous work, Yee [6] introduced the *Principle of Explicit Authorization* to describe situations where input to an application from a trusted user can be used to guide access control decisions [6]. However, previous work takes advantage of a user's input for only one specific case. Karp et al. [2, 5] have implemented a custom implementation of a file chooser GUI widget to monitor the files chosen by the user and give the application access to only those files (which was subsequently the focus of a usability study published at SOUPS 2006 [1]). However, this support does not extend to other GUI or any non-GUI elements of the application, all of which could potentially be harnessed to guide access control decisions.

We use the term *implied authority* [1] as the authority granted to some subject (the agent) based on the demands made of it from some other trusted subject (the principal). To meet the demands of the principal, the agent will necessarily require certain authority (i.e. privileges). Our research looks into ways of detecting the privileges necessary for a DC to fulfill the demands placed on it.

We have identified two specific sources of demands placed on the application: the user and the reading of data driven from user's actions. This setup is shown at a high-level in Figure 1. The first, as identified by Yee [6], makes use of a user's intentions as expressed through their actions with the GUI. Our approach considers two specific kinds of GUI interactions: text input from the keyboard and item selection through the mouse. A developer using our approach attaches security semantics to GUI widgets by specifying a mapping between Java `class` fields that refer to GUI widgets and some policy. The second case is identifying resources referred to in (semi-) structured data such as XML files. Using our approach a developer attaches security semantics to XML data by providing a mapping between XML elements and a policy using a small XPath-like language.

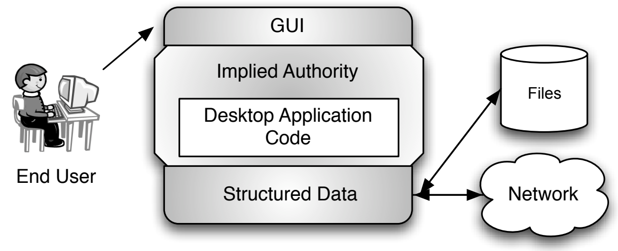

Figure 1: Basic structure for using implied authority. Interactions with the application carry implied authority, through both the GUI and structured data, granting and revoking authority to access various resources during execution.

These approaches allow for the end user to be a direct, but unconsciously-aware participant in the delegation of authority to the application. By hiding the security decisions behind GUI and data input events, the usability for an end user of enforcing POLA on an application is greatly simplified.

2. POLICY FOR POLA IN CLIENT APPLICATIONS

The high-level security model in our approach consists of:

- *DC*, a desktop client application
- *user*, the end user of the *DC*
- *developer*, a trusted developer of the *DC*
- $hosts = \{h_1, h_2, ..., h_\infty\}$, a set of network hosts
- $files = \{f_1, f_2, ..., f_\infty\}$, a set of local files

We wish to protect the *user* from unexpected disclosure or corruption of sensitive data which is stored in $files$ or in the run-time state of the DC. This is done by minimizing the access of the DC to $hosts \cup files$ (i.e. the resources).

[1]In contract law, implied authority is "The authority to perform acts that are customary, necessary, and understood by an agent as authorized in performing acts for which the principal has given express (explicit) authority" -Merriam-Webster's Dictionary of Law

We assume that the *user* trusts the primary *developer* of the *DC* to act in good faith. However, the *DC* is developed out of a large number of components, any of which could be faulty or infected by malware. So there is the potential for some problem to occur in the *DC*, outside of the trusted *developer*'s control.

Our enforcement mechanism builds from a traditional access control list (ACL). The security mechanism maintains an ACL for each application execution instance (i.e. process), which is the only subject for which the ACL applies. Each entry in the list contains three pieces of information, the first two of which should be familiar:

- *resource*, a canonical string representation of some element from $hosts \cup files$. These resources are the objects of the ACL.
- *permission*, a permission modifier such as `read`, `write`, `connect`.
- *lifetime*, an indication of when the entry is valid: `transient` or `persistent` (details described below).

With the application initially configured for complete user mediated resource access, certain application inputs are monitored and used to relax user mediation, by automatically granting some authority at run-time. We model this structure as follows:

- $format = \{s_1, s_2, ..., s_n\}$, a set of XML data formats used by the *DC* (e.g. HTML, RDF, Atom, etc.). We assume each $format$ has some well-defined schema (either formally or informally) so that data elements in an instance of that format can easily be queried.
- $gui = \{g_1, g_2, ..., g_n\}$, a set of `class` fields which may reference a widget that receives information from the user.

In our proposed approach, it is the responsibility of the *developer* to understand the details of the application as they relate to the $format$ and the *gui*. They can use that understanding to write a policy to control the authority of the application.

Writing a *DC* POLA policy consists of creating rules which identify elements from $format \cup gui$ which imply that demands are being made of *DC*. These rules act as generators for ACL entries. In our proposed approach, developers can generate entries using two kinds of policy rules:

- *xmlRule*, where an element of *xmlRule* is a query given in a small, custom, XML query language.
- *widgetRule*, where an element of *widgetRule* is a pair consisting of an application class name and some field (i.e. property/attribute) of that class which is meant to reference a widget object.

Essentially, a policy is just a map from *xmlRule* $\cup$ *widgetRule* to *permission* $\times$ $lifetime$. Each rule specifies that, if data elements matched by a query are parsed, or if specified widgets have data input by the user, then a new entry should generated and added to the ACL.

Each entry in the ACL includes a property indicating when the permission represented in the entry is valid. This *lifetime* of the entry indicates whether: the entry should exist only for the life of a specific thread context (`transient`) or the entry should be saved across application executions (`persistent`). The determination of an entry lifetime is made by the *developer*, based on their knowledge of the application's semantics.

In some cases, users will refer to resources in the GUI using a semantic representation of a resource (e.g. a title or description) rather than the lower-level URL or file path. To make the connection between the semantic representation of a resource and the resource's identity, the developer can create an alias. To set up an alias, a developer provides two *widgetRule*s or *xmlRule*s: one which matches the alias text and one which matches the resource identifier. This ensures that the two rules can be matched simultaneously so that an association between resource identifier and alias can be made automatically.

3. CASE-STUDY

We are applying our approach to restrict the authority of an existing open-source application, RSSOwl 1.2.4; a roughly 46,000 LoC Really Simple Syndication (RSS) reader written in Java and SWT [3]. RSSOwl makes use of nine third-party components, totaling 2201K worth of jar files. In the table below, we list the number of rules for implied authority found in RSSOwl through GUI interactions and data handling. We hope to determine whether or not the enforcement of POLA can be achieved using only the high-level code and data matching rules offered through our approach.

Rule type	Persistent	Transient	Alias
widgetRule	3	10	3
xmlRule	2	1	5

Table 1: Numbers of implied authority policy rules written for RSSOwl. Rows represent whether rules were triggered by GUI actions or application data input. Columns represent whether rules were used to generate ACL entries for a particular lifetime or used to generate an alias.

4. REFERENCES

[1] A. J. DeWitt and J. Kuljis. Aligning usability and security: a usability study of polaris. In *SOUPS '06: Proceedings of the second symposium on Usable privacy and security*, 2006.

[2] A. H. Karp. POLA Today Keeps the Virus at Bay. Technical Report HPL-2003-191, HP Laboratories Palo Alto, 2003.

[3] B. Pasero. RSSOwl. `http://www.rssowl.org/`.

[4] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Communications of the ACM*, volume 17, 7, 1974.

[5] M. Stiegler, A. H. Karp, K.-P. Yee, and M. S. Miller. Polaris: Virus Safe Computing for Windows XP. Technical Report HP:-2004-221, HP Laboratories Palo Alto, 2004.

[6] K. Yee. User Interaction Design for Secure Systems. In *International Conference on Information and Computer Security*, 2002.