# Towards an Algebra for Security Policies (Extended Abstract)

Jon Pincus[1] and Jeannette M. Wing[2]

[1] Microsoft Research,
One Microsoft Way, Redmond, WA 98052
jpincus@microsoft.com
[2] Computer Science Department, Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213
wing@cs.cmu.edu

## 1 Context

Clashing security policies leads to vulnerabilities. Violating security policies leads to vulnerabilities. A system today operates in the context of a multitude of security policies, often one per application, one per process, one per user. The more security policies that have to be simultaneously satisfied, the more likely the possibility of a clash or violation, and hence the more vulnerable our system is to attack. Moreover, over time a system's security policies will change. These changes occur at small-scale time steps, e.g., using setuid to temporarily grant a process additional access rights; and at large-scale time steps, e.g., when a user changes his browser's security settings. We address the challenge of determining when a system is in a consistent state in the presence of diverse, numerous, and dynamic interacting security policies.

Formal specifications of these security policies let us pinpoint two potential problems: when security policies for different components are inconsistent and when a component does not satisfy a given security policy. We present a simple algebra for combining and changing security policies, and show how our algebraic operations can be used to explain different real-life examples of security policy clashes and violations.

## 2 Model and Definitions

We model security policies as access rights matrices:

$$SP \subseteq P \times O \times R$$

where $P$ is a set of principals, $O$ is a set of typed objects, and $R$ is a set of rights. Principals include processes, users, applications, etc. Objects include files, directories, registry keys, communication channels, etc. Rights are type-specific: for each type of object, there are certain associated operations. For example, for a file, the operations might be open, close, read, write, and execute; for a Web service, the operations might be search, recommend, and purchase. Henceforth, when we say "security policy," we mean its underlying access rights matrix.

G. Ciardo and P. Darondeau (Eds.): ICATPN 2005, LNCS 3536, pp. 17–25, 2005.

For a given security policy, $sp$, principal, $p$, and object, $o$, we write $sp(p, o)$ to stand for the set of rights in $R$ that $p$ has on $o$. Informally, this means that for an operation in $sp(p, o)$, $p$ has the right to invoke that operation on $o$. Negative rights are represented implicitly, by the absence of an explicit right in the security policy.

**Definition 1.** *Two security policies,* $sp_1$ *and* $sp_2$*,* **clash** *iff* $sp_1 \neq sp_2$.

**Definition 2.** *Given two security policies,* $sp_1$ *and* $sp_2$*,* $sp_1$ **respects** $sp_2$ *iff* $\forall p \forall o \; . \; sp_1(p, o) \subseteq sp_2(p, o)$*; otherwise,* $sp_1$ **disrespects** $sp_2$.

Combining two security policies potentially introduces vulnerabilities. Whether there is a vulnerability depends on the way in which the two security policies combine. Let $\_ \oplus \_ : SP \times SP \rightarrow SP$ denote a combination operation on two security polices. There is a potential vulnerability if $sp_1 \oplus sp_2$ disrespects $sp_1$ or $sp_1 \oplus sp_2$ disrespects $sp_2$. Disrespecting security policies imply they clash.

## 3 An Algebra for Security Policies

Since security policies are ternary relations, i.e., sets of triples, we define combinations of security policies in terms of operations on sets. In practice, in combining two security policies we might require that we satisfy both (*And*); satisfy either (*Or*); or satisfy one but not another (*Minus*). We might simply override (*Trumps*) the second by the first.

For the following combinations, we assume that the security policies are defined over the same sets of $P$, $O$, and $R$.

**spec** SecurityPolicy
$\_And\_ : SP \times SP \rightarrow SP$
$\_Or\_ : Sp \times SP \rightarrow SP$
$\_Minus\_ : SP \times SP \rightarrow SP$
$\_Trumps\_ : SP \times SP \rightarrow SP$

$sp_1 \; And \; sp_2 = sp_1 \cap sp_2$
$sp_1 \; Or \; sp_2 = sp_1 \cup sp_2$
$sp_1 \; Minus \; sp_2 = sp_1 \setminus sp_2$
$sp_1 \; Trumps \; sp_2 = sp_1$
**end** SecurityPolicy

Whereas *And* and *Or* are commutative, *Trumps* and *Minus* are not, as should be clear from the equations above. The following facts follow from the definitions of *respects*, *And*, and *Trumps*:

$(sp_1 \; And \; sp_2)$ respects $sp_1$
$(sp_1 \; And \; sp_2)$ respects $sp_2$
$(sp_1 \; Trumps \; sp_2)$ respects $sp_1$

When we use *Or* to combine two security policies, $sp_1$ and $sp_2$, the combination might disrespect $sp_1$ or disrespect $sp_2$; similarly, when we use *Trumps*, the combination might disrespect $sp_2$.

We also introduce a way for one principal to gain (*Inherit*) its rights from another, so that $Inherit(p_1, p_2)$ means $p_1$ inherits rights from $p_2$; and conversely, for one principal to grant (*Delegate*) its rights to another. We include a revocation (*Revoke*) operator to remove all rights associated with a given principal.

**spec** ChangeSP **extends** SecurityPolicy
$Inherit : SP \times P \times P \rightarrow SP$
$Delegate : SP \times P \times P \rightarrow SP$
$Revoke : SP \times P \rightarrow SP$

$Inherit(sp, p_1, p_2) = sp \cup \{\langle p, o, r\rangle | \langle p_1, o, r\rangle \in sp \wedge p = p_2\}$
$Delegate(sp, p_1, p_2) = Inherit(sp, p_2, p_1)$
$Revoke(sp, p) = sp \rhd (Prin(sp) \setminus \{p\})$
**end** ChangeSP

where $\rhd$ is the domain restriction operator and $Prin : SP \rightarrow Set[P]$ returns the set of principals over which a given security policy is defined. *Inherit* (and similarly *Delegate*) has the following compositional property:

$Inherit(Inherit(sp, p_2, p_3), p_1, p_2) \supseteq Inherit(sp, p_1, p_3)$

which informally says "If $p_2$ inherits $p_3$'s rights and then $p_1$ inherits $p_2$'s rights, then $p_1$ inherits $p_3$'s rights."

## 4 Examples

We give a series of four examples. The first (Outlook and IE) illustrates an example of two different components, each of which has their own security policy; when put together, one security policy "trumps" the other, causing a potential privacy violation as well an surprising behavior to the user. The second (Run As) illustrates a use of inheriting rights that causes potential security vulnerabilities. The third (Google Desktop Search) illustrates another use of inheritance. Finally, the fourth example (Netscape and DNS) simply shows what can happen when an informally stated security policy is ambiguous, leading to an implementation with a security vulnerability. We present the first example in detail and give the gists of the problems for the other three. The first, third, and fourth examples are also examples of *composition flaws*, when two independently designed and implemented components are combined in a way that lead to unintended interactions, which could lead to security vulnerabilities.

### 4.1 Outlook and IE

Many applications (e.g., email clients and browsers) have security settings that users may modify. Each configuration of the settings represents a different security policy.

Let's consider an example where we would like to block the display of embedded graphics in our mail messages. Microsoft's Outlook email client includes a setting "Don't Download Pictures" which lets the user specify this behavior. Microsoft's Internet Explorer (IE) browser includes a similar setting "Show Pictures" which lets the user control the display of graphics embedded in any HTML document. Since Outlook uses IE's HTML rendering component, these two settings interact in the case of HTML email.

**Actual Behavior**

The following table describes the actual behavior when reading and forwarding messages. There are four possibilities for how an image is displayed:

- **display**: Retrieve and display the image.
- **red X**: Display a small icon of a red X, with a textual comment saying "Right-click here to download pictures. To help protect your privacy, Outlook prevented automatic download of this picture from the Internet."
- **small graphic**: Display a small "unknown graphic" icon, with the same textual comment.
- **sized graphic**: Display a correctly-sized box with a small "unknown graphic" icon (and no textual comment).

As will be discussed below, the behavior in italics is a vulnerability.

| IE Show Pictures | Outlook Don't Download Pictures | |
|---|---|---|
| | False | True |
| False | Read: sized graphic<br>Forward: sized graphic | Read: small graphic<br>Forward: sized graphic |
| True | Read: display<br>Forward: display | Read: red X<br>*Forward: display* |

**The Vulnerability and an Attack**

From a security perspective, there are at least two reasons to disable graphics in email. First, downloading the graphic transfers information back to the web site containing the graphic; at a minimum, this kind of information disclosure confirms that the mail was received and viewed, e.g., allowing spammers to confirm that an email address is real. Second, disabling graphics is a *Defense in Depth* strategy that mitigates the risk of unknown exploitable bugs, e.g., buffer overruns in image rendering code. Thus, the counter-intuitive display of graphics while forwarding email is a vulnerability.

An attacker can exploit this vulnerability in a social engineering attack if he can convince a user, Alice, to forward some email containing an image. For example, consider the scenario where the attacker wants to validate whether an email address `Alice@bigco.com` corresponds to a real user (perhaps in the context of a brute-force generation of all email addresses for the domain `bigco.com` to discover which are valid addresses to resell to spammers). If Alice has downloading of images disabled, the straightforward attack of simply getting Alice to read the mail will fail. However, if the attacker knows that `bigco` has an internal policy of forwarding all "phishing" email to a central alias for followup, then the attacker simply sends an obviously fake "phishing" email containing an embedded image to `Alice@bigco.com`. Alice can read the email without any danger; but when she goes to forward it, because of the vulnerability, the image is downloaded, and the attacker confirms that the address is valid.

**A Fix**

The vulnerability disappears if the behavior is as follows, where the only change, shown in boldface, is to display a red X on mail forwarding instead of retrieving and displaying the image.

| IE Show Pictures | Outlook Don't Download Pictures | |
|---|---|---|
| | False | True |
| False | Read: sized graphic<br>Forward: sized graphic | Read: small graphic<br>Forward: sized graphic |
| True | Read: display<br>Forward: display | Read: red X<br>**Forward: red X** |

**Using our Security Policy Formalism**
More formally, we can use our algebra for security policies to characterize the security policy clash. Let *viewread* be the right associated with the operation that lets us view an embedded graphic while reading a mail message; *viewforward*, while forwarding.

**Security Policy for Outlook** ($SP_{Outlook}$): If the "Don't Download Pictures" setting is false (i.e., downloading pictures is ok), then all users can view embedded graphics when both reading and forwarding mail messages. More precisely:

$$\text{Download Pictures = true}$$
$$\Leftrightarrow$$
$$\forall p : user\ \forall o : embedded\_graphic$$
$$\langle p, o, viewread \rangle \in SP_{Outlook} \wedge \langle p, o, viewforward \rangle \in SP_{Outlook}$$

(To avoid a double negative, we wrote above "Download Pictures = true" rather than the more accurate "Don't Download Pictures = false".)

**Security Policy for IE** ($SP_{IE}$): If the "Show Pictures" setting is true, then all users can view embedded graphics when viewing HTML. This setting applies both when reading and forwarding mail messages. More precisely:

$$\text{Show Pictures = true}$$
$$\Leftrightarrow$$
$$\forall p : user\ \forall o : embedded\_graphic$$
$$\langle p, o, viewread \rangle \in SP_{IE} \wedge \langle p, o, viewforward \rangle \in SP_{IE}$$

When Outlook is combined with IE, then the actual behavior differentiates between the reading and forwarding cases:

Read: $SP_{IE}\ And\ SP_{Outlook}$
Forward: $SP_{IE}\ Trumps\ SP_{Outlook}$

When reading email, then the actual behavior reflects the conjunction of the security policies associated with Outlook and IE; but when forwarding, then the actual behavior reflects IE's security policy, trumping Outlook's. Thus, when forwarding, the combined security policy disrespects the Outlook security policy:

$(SP_{IE}\ Trumps\ SP_{Outlook})$ disrespects $SP_{Outlook}$

The fix reflects conjunction in both cases, regardless of whether we are reading or forwarding email:

Read: $SP_{IE}\ And\ SP_{Outlook}$
Forward: $SP_{IE}\ And\ SP_{Outlook}$

By properties of *And* (Section 3), we know that the combined security policies respect both of the individual ones.

Note that the original combination of security policies led not only to a security vulnerability but also to a confusing usability problem. Users would normally expect a single policy to hold (whether it be Outlook's, IE's, or their conjunction) regardless of what operation they perform, e.g., reading or forwarding a mail message. When the two policies combine in one way for one operation and in another way for the other, the user's mental model is inconsistent, which invariably leads to usability issues. The proposed fix also removes this inconsistency.

### 4.2 Run as

Suppose in a file system, *FS*, we wish to let a process, $p$, perform operations on behalf of (i.e., "run as") a user, $u$; this "run as" capability allows $p$ to gain access temporarily to a set of files owned by $u$. We can model this behavior in terms of inheriting rights. For example, if $u$ can read a file, $f$, and if $p$ inherits $u$'s rights, then $p$ can read $f$. The Unix setuid mechanism and Windows impersonate privileges are implementations of this functionality.

Let $SP_{FS}$ stand for the security policy on a given file system. Now consider the following scenario. First, $u_1$ executes $p_1$. This has the effect of giving $u_1$'s rights to $p_1$:

[1] $SP_E = Inherit(SP_{FS}, p_1, u_1)$

Now, $p_1$ "runs as" $u_2$. This has the effect of first revoking $p_1$'s original rights and then giving $u_2$'s rights to $p_1$:

[2] $SP_{RA} = Inherit(Revoke(SP_E, p_1), p_1, u_2)$

Assuming $u_1$ controls $p_1$'s behavior, then for the lifetime of $p_1$, $u_1$ gets whatever rights $p_1$ acquires:

[3] $SP_C = Inherit(SP_{RA}, u_1, p_1)$

From the compositional property of *Inherits* (Section 3), [2] and [3] leads to [4] below, and we are left in a state where $u_1$ has inherited $u_2$'s rights:

[4] $SP_C \supseteq Inherit(SP_{FS}, u_1, u_2)$

Let's see how the use of "run as" can lead to a security vulnerability. First, we introduce a new kind of access right, $a$, to stand for "can run as." In our security policy, we have entries such as $\langle p, u, a\rangle$, which says that a process, $p$, "can run as" user, $u$.

Now consider a file system with entries such as $\langle u_1, p_1.exe, x\rangle$, which says that user $u_1$ has execute rights on the executable object, $p_1.exe$ (the executable associated with process $p_1$), as well as entries such as $\langle u_1, f, read\rangle$, which says that user $u_1$ has read access to file $f$.

A process's rights may change over time, depending on what user they are running as. Thus, a process can access a file, $f$, if the user the process is currently running as has access to $f$.

More concretely, a vulnerability can arise, if $SP_{FS}$ consists of the following entries:

$\langle p_1, *, a \rangle$
$\langle u_1, p_1.exe, x \rangle$
$\langle u_2, f, read \rangle$

The first entry is the troublesome one: it gives process $p_1$ the ability to run as any user. Since $u_1$ can execute $p_1.exe$, which creates a process $p_1$ with the given rights, and since $p_1$ can run as $u_2$ and $u_2$ has access to $f$, then $u_1$ gains access to $f$ even though that right is absent from the security policy.

### 4.3 Google Desktop Search

When a `google.com` request is made, Google Desktop Search (GDS) performs a search on the local file system with the same request. The local search results include 30-40 character snippets of local files that contain the query's terms. GDS integrates its local search results with the webpage returned by `google.com`. If the query has been made by a user in a standard browser window, the user sees a webpage that contains results of the search on the local host and results from `google.com`; but if the query is made by an applet, then GDS introduces a potential security vulnerability [2].

**Security Policy for Google Desktop Search** ($SP_{GDS}$): The Google Desktop Search process has read access to all files on the local host.

Or more formally,

$$\forall f \in localhost \; \langle GDS, f, read \rangle \in SP_{GDS}$$

Suppose an applet, *app*, sends a query that GDS executes. We have the following situation:

[1] $SP_{App} = Inherit(SP_{GDS}, app, GDS)$

Since the applet can connect to a remote host, in particular, the host, $rh$, from which the applet originally came, the remote host inherits the applet's rights:

[2] $SP_{RemHost} = Inherit(SP_{App}, rh, app)$

By compositionality of *Inherits* again ([1] and [2] leads to [3] below), we are in the situation where the remote host inherits the rights of GDS:

[3] $SP_{RemHost} \supseteq Inherit(SP_{GDS}, rh, GDS)$

which means the remote host has read access to files on the local host—a security vulnerability!

The fix that Google made to address this vulnerability is to disallow the applet from seeing the results of a local search using GDS. This has the effect of invalidating the state labeled [1] above.

### 4.4 Netscape and DNS

Our final example shows what can go wrong when the specification of a security policy is ambiguous, leading to an implementation with a security vulnerability. Here the vulnerable component is the browser, which operates in the DNS infrastructure environment [1]. Applets running in the browser sometimes need to contact the server from which it originated. We have the following security policy on applets (which also must hold for the Google Desktop Search example):

**Security Policy for Applet** ($SP_{applet}$): An applet should connect to the same server from which it originated.

More formally, in terms of an access rights matrix, we have the following, where *conn* stands for the right for an applet $a$ to connect to a host $h$:

$$\forall a : applet\ \forall h : host\ \langle a, h, conn\rangle \in SP_{applet} \\ \Leftrightarrow SameAs(a.OriginatingHost, h)$$

The ambiguity in the informal policy is what "same server" means. In the formal statement, it boils down to how *SameAs* is interpreted. There are two sources of ambiguity. First, does *SameAs* mean "same IP address" or "same name"? An interpretation of "same IP address" for *SameAs* seems too restrictive to support some common usage scenarios, and so Netscape chose to resolve this first ambiguity by using a check based on DNS names. Unfortunately, the name-based check raises two problems: (1) the name of a server might map to multiple IP addresses (machines) and (2) the mapping of names to IP addresses can change over time. The possibility of change over time is a second source of ambiguity: Doing a DNS lookup on a name at one point in time, e.g., when the applet is downloaded, does not guarantee the same result as doing it at a later point in time, e.g., when the applet wishes to connect to a remote host.

In more detail, here is the problem. In the effort to enforce the security policy, Netscape's original check used two DNS name lookups. Let *n2a* be the many-to-many relation that maps names to IP addresses, *From* be the name of the server from which the applet originated, and *To* be the name of the server to which the applet wishes to connect. If the lookup on both names yields a nonempty intersection of IP addresses, then the assumption is that *From* and *To* are "the same server," and we allow the connection. More succinctly, Netscape implemented this check:

**if** *n2a(From)* $\cap$ *n2a(To)* $\neq \emptyset$
**then** $\exists x \in n2a(From)\ \exists y \in n2a(To)$ such that *connect(x, y)*

where $connect$: $Host \times Host \rightarrow Bool$ means that for hosts $h_1$ and $h_2$ $connect(h_1, h_2)$= *true* iff there is a connection from $h_1$ to $h_2$.

There are two problems with Netscape's check. The first problem is directly related to the second ambiguity and leads to a vulnerability: in doing the lookup on *From* at the time when the applet wants to connect to *To*, the set of IP addresses to which *From* maps may be different from the time when it was first downloaded. The second problem is simply a logical flaw: choosing some $x$ in *n2a(From)* and some $y$ in *n2a(To)* to establish the connection does not even guarantee that the $x$ and $y$ are in the (nonempty) intersection of *n2a(From)* and *n2a(To)*.

By manipulating DNS lookup (for example, by running his or her own name resolver), an attacker can effectively allow an applet to connect to any host on the network without violating the policy.

Netscape fixed the vulnerability by storing the actual IP address $i$ of the originating server (eliminating the first lookup) and changing the intersection check to a membership check, $i \in n2a(To)$. This fix means the implementation matches the intended security policy.

## 5 Summary and Future Work

We sketched the foundation of a simple algebra for reasoning about security policies, viewed as access rights matrices. We also sketched the use of our algebra on four real-life examples: the Outlook and IE example shows a security policy clash; the Run As and the Google Desktop Search examples show two different uses of inheriting rights as security policies change over time; and the Netscape and DNS example shows the consequence of a violation of a security policy.

We would like to develop a security policy language and logic for expressing policies closer to the way in which software designers think about security requirements and use our algebra to show when clashes can occur or when designs violate their requirements. We are also interested in building tool support for automating our reasoning and for letting us scale our approach to large examples.

## Acknowledgments

We thank Oren Dobzinski for helping us think about the Google Desktop Search example and for comments on earlier drafts of this manuscript. The second author thanks Microsoft Research for partial support of this work. She is also partially sponsored by the Army Research Office under contract no. DAAD190110485 and DAAD19-02-1-0389, the National Science Foundation under grant no. CCR-0121547 and CNS-0433540, and the Software Engineering Institute through a US government funding appropriation.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the US Government or any other entity.

## References

1. D. Dean, E.W. Felten, and D.S. Wallach, "Java Security: From HotJava to Netscape and Beyond," *Proceedings of the 1996 IEEE Symposium on Security and Privacy,* Oakland, CA, May 1996.
2. S. Nielson, S.J. Fogarty, and D.S. Wallach, "Attacks on Local Searching Tools," Technical Report TR04-445, Department of Computer Science, Rice University, December 2004.