

# Factoring High-Level Authorizations

Jon A. Solworth

Department of Computer Science, University of Illinois at Chicago

<http://www.kernelSec.org>

## 1. What problem are we trying to solve?

We are designing and implementing a sophisticated authorization model for an operating system. Here we consider just one threat.

### 1.1 Threat environment

- Programs from many sources,
  - some completely untrusted, may have been crafted by attacker
  - some from benign sources, but have dangerous flaws
- Programs propagate informally (e.g., as an email attachment)
- Interpreters and buffer overflow mean that data which appears to be merely read can be also executed

### 1.2 Total isolation is not a solution

- Inherently need to integrate data from multiple sources
- Therefore must regulate sharing and copying
  - constrain what an **executable**—as well as a user—can do
  - track information flow so user doesn't have to
- No magic “one size fits all”—need flexible mechanism

### 1.3 Example—Dynamic Sandboxing

Below is an example of a **dynamic sandbox**, implemented in kernelSec. Its purpose is to ensure that potentially dangerous email attachments are interpreted in a sandbox, and thus do not pose a danger to other parts of the system.

1. The Mail User Agent reads from a network connection (e.g., IMAP service) the mail and stores it locally with label “Mail”.
2. A viewer, xpdf, reads the attachment and displays it in a sandbox. The sandbox is triggered by reading Mail.
3. A program, cp, copies the attachment to a new file, the authorization model labels the new file as Mail (information flow).
4. The viewer, xpdf, if invoked on a local file is not sandboxed.
5. The viewer, if invoked on the copied file, is sandboxed.

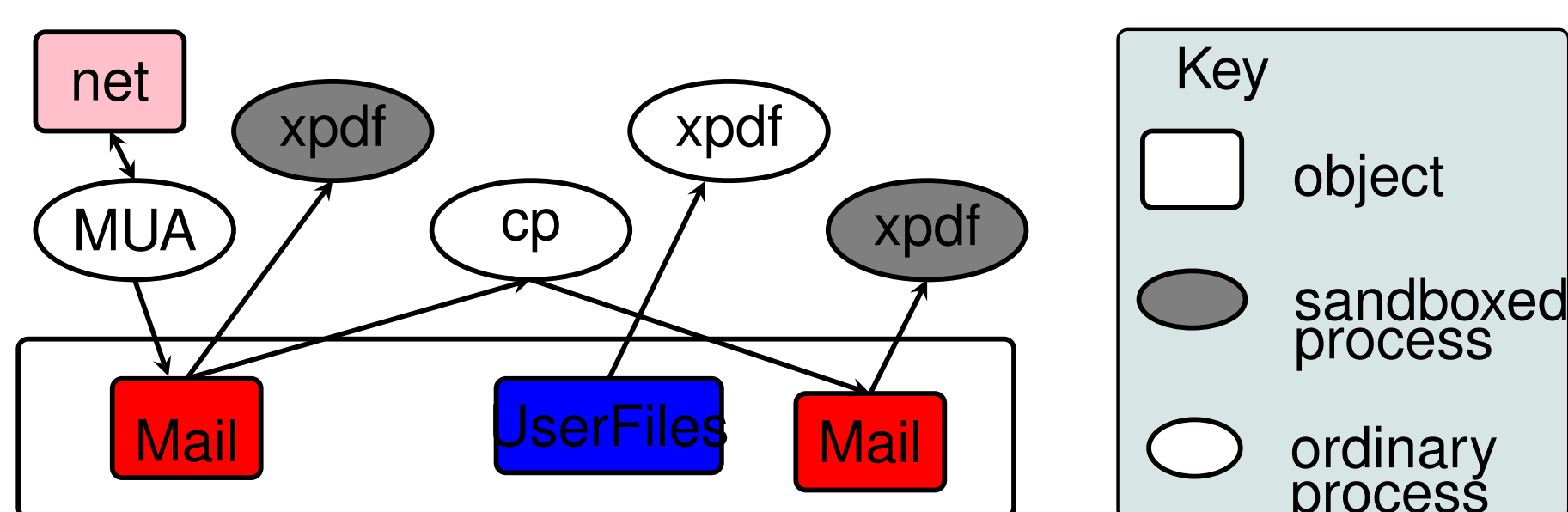


Figure 1. Sandbox mechanism implemented in kernelSec

## 2. Authorization Models and their tradeoffs

Many sophisticated authorization models (access controls) have been proposed and built, but are not in wide use because of their complexity.

- sophisticated authorization models are needed to combat current threats, but
- greater sophistication results in greater complexity.

**Fundamental question:** How can sophisticated protections be achieved with modest levels of complexity and thus be **usable**?

## 3. The kernelSec approach

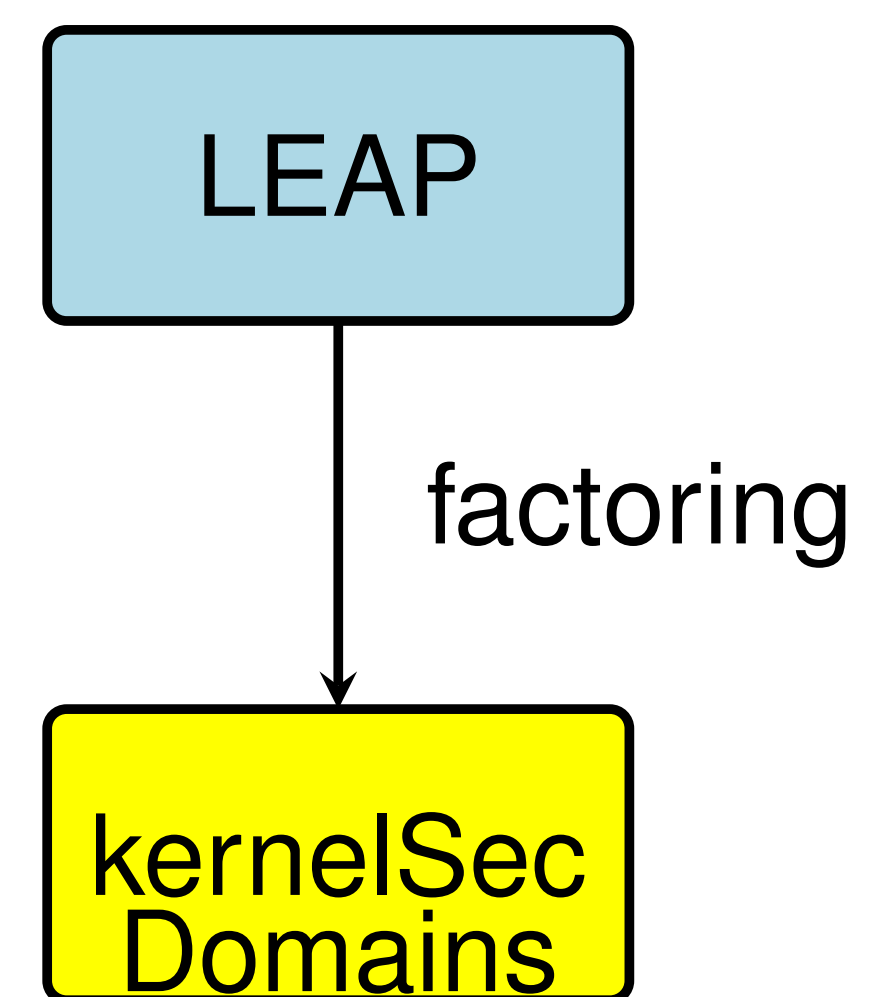
KernelSec is a sophisticated authorization model implemented in an operating system kernel. One of the methods we are investigating to reduce complexity is to specify authorization in a high level language we call **LEAP**.

**LEAP:** a **Language for Expressing Authorization Properties**. LEAP is succinct, composable, and mostly stateless.

**kernelSec Domains:** a low level enforcement engine implemented in the Linux kernel.

**factoring:** translates from LEAP to kernelSec Domains.

Current NSF Cybertrust grant supports development of LEAP and its translation (factoring) to kernelSec Domains.



## 4. LEAP

LEAP permissions are defined on labels. Given a label  $l$ , the unary permissions  $c(l)$ ,  $r(l)$ ,  $w(l)$ , and  $x(l)$  are defined. These specify create, read, write, and execute permission on objects with label  $l$ . In addition there are two binary permissions

**mayflow( $l, l'$ )** The permission to write  $l'$  after having read  $l$ . This is a necessary permission, also needed is  $w(l')$ .

**relabel( $l, l'$ )** The permission to change an object's label from  $l$  to  $l'$ .

Each of these permissions is defined by assigning it a **holder** which specifies which processes can use that permission. The form of the holder is a set of pairs  $(e, g)$ , where  $e$  is the label of an executable and  $g$  a group of users; however we shall use only  $e$  in the below example. For simplicity, we also ignore  $c(l)$  and  $x(l)$ .

$r(\text{net})$	=	MUA
$w(\text{net})$	=	MUA
$r(\text{Mail})$	=	viewer, MUA, scrubber
$w(\text{Mail})$	=	MUA
$r(\text{UserFiles})$	=	*
$w(\text{UserFiles})$	=	!MUA
$\text{mayFlow}(\text{net}, \text{Mail})$	=	MUA
$\text{relabel}(\text{Mail}, \text{UserFiles})$	=	scrubber

This specification describes all the behavior shown in Figure 1. In addition, we note other interesting behavior for this configuration:

- A scrubber program may read Mail and if it determines it to be safe, can relabel it as UserFiles.
- A pdf viewer can read and write UserFiles or
- A pdf viewer can read and write Mail
- But a pdf viewer cannot read Mail and write UserFiles since  $\text{mayFlow}(\text{Mail}, \text{UserFiles})$  does not include the viewer executable.

## 5. Factoring

LEAP specifications are automatically factored into kernelSec domains. Issues include

- optimizations to reduce number of generated domains
- Impact on the specification of domains
- semantic equivalence between LEAP and kernelSec domains.

## 6. Future directions

- Networking support (integrate firewalls with kernelSec)
- Strong distributed authentication
- Application porting to kernelSec, security analysis, usability, etc.

## 7. Conclusions

- Sophisticated authorizations are needed given current threats
- A multi-faceted approach is needed to fight complexity