

# Vulnerability Assessment as Part of a Security Strategy

Barton P. Miller

James A. Kupsch

Computer Sciences Department  
University of Wisconsin

Elisa Heymann  
Eduardo César

Computer Architecture & Operating  
Systems Department  
Universitat Autònoma de Barcelona

OGF28, München  
March 16, 2010

# The Obligatory Scary Slide

- The bad guys are trying to do *really bad* things to us.
- They are smart, dedicated and persistent.
- No single approach to security can be sufficient.
- The attackers have a natural advantage over the defenders.

We have to approach the defense of our systems as *security in depth*.



# Our Piece of the Solution Space

A brief story...

- We started by trying to do something simple:

Increase our confidence in the security of some critical Grid middleware.

- We ended up developing a new manual methodology:

First Principles Vulnerability Assessment

- We found some serious vulnerabilities ... and more vulnerabilities ... and more.



# Studied Systems



**Condor**, University of Wisconsin  
Batch queuing workload management system  
**15 vulnerabilities** 600 KLOC of C and C++



**SRB**, SDSC  
Storage Resource Broker - data grid  
**5 vulnerabilities** 280 KLOC of C



**MyProxy**, NCSA  
Credential Management System  
**5 vulnerabilities** 25 KLOC of C



**glExec**, Nikhef  
Identity mapping service  
**5 vulnerabilities** 48 KLOC of C



**Gratia Condor Probe**, FNAL and Open Science  
Feeds Condor Usage into Gratia Accounting System  
**3 vulnerabilities** 1.7 KLOC of Perl and Bash



**Condor Quill**, University of Wisconsin  
DBMS Storage of Condor Operational and Historical Data  
**6 vulnerabilities** 7.9 KLOC of C and C++

# In Progress Systems



**Wireshark**, [wireshark.org](http://wireshark.org)  
Network Protocol Analyzer  
in progress

**2400** KLOC of C



**Condor Privilege Separation**, Univ. of Wisconsin  
Restricted Identity Switching Module  
in progress

**21** KLOC of C and C++



**VOMS Admin, INFN**

Web management interface to VOMS data (role

in progress

**35** KLOC of Java and PHP



**CrossBroker**, Universitat Autònoma de Barcelona  
Resource Mgr for Parallel & Interactive Applications  
in progress

**97** KLOC of C++

# An Example Vulnerability Report



**CONDOR-2005-0003**

**SDSC**

## Summary:

Arbitrary commands can be executed with the permissions of the condor\_shadow or condor\_gridmanager's effective uid (normally the "condor" user). This can result in a compromise of the condor configuration files, log files, and other files owned by the "condor" user. This may also aid in attacks on other accounts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_shadow condor_gridmanager	6.6 - 6.6.10 6.7 - 6.7.17	all	not known to be publicly available	6.6.11 - 6.7.18 -
Status	Access Required	Host Type Required	Effort Required	Impact/Consequences
Verified	local ordinary user with a Condor authorization	submission host	low	high
Fixed Date	Credit			
2006-Mar-27	Jim Kupsch			

**Access Required:** local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that is running a condor\_schedd, to which the user can use condor\_submit to submit a job.

**Effort Required:** low

To exploit this vulnerability requires only the submission of a Condor job with an invalid entry.

**Impact/Consequences:** high

Usually the condor\_shadow and condor\_gridmanager are configured to run as the "condor" user, and this vulnerability allows an attacker to execute arbitrary code as the "condor" user.

Depending on the configuration, additional more serious attacks may be possible. If the configuration files for the condor\_master are writable by condor and the condor\_master is run with root privileges, then root access can be gained. If the condor binaries are owned by the "condor" user, these executables could be replaced and when restarted, arbitrary code could be executed as the "condor" user. This would also allow root access as most condor daemons are started with an effective uid of root.





# Our Piece of the Solution Space

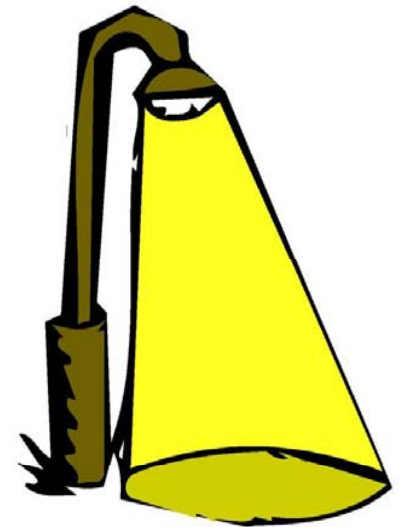
## First Principles Vulnerability Assessment:

- An analyst-centric (manual) assessment process.
- You can't look carefully at every line of code so:

We don't start with known threats ...

... instead, identify high value assets in the code and work outward to derive threats.

- Start with architectural analysis, then identify key resources and privilege levels, component interactions and trust delegation, then focused component analysis.



# Our Piece of the Solution Space

A brief story... (con'd)

- Finding vulnerabilities was only the beginning of the process.

The reporting to the software team, developing remediation, and announcing to users and the public can all be quite complex.

- Development teams complained a lot: why can't we use one of the well-regarded automated tools?





# Manual vs. Automated Vulnerability Assessment

The literature on static analysis tools is self-limiting:

- Missing comparison against a ground truth
- Tool writers write about what they have found

Every valid new problem that a tool find is progress, but it's easy to lose perspective on what these tools are *not* able to do

# Case Study: Methodology

- Previously assessed Condor using our FPVA
- Identified the best-regarded automated tools:
  - Coverity Prevent 4.1.0
  - Fortify SCA 5.1.0016
- Applied these tools to the same version of Condor as was used in the FPVA study
- Goal: to evaluate the ability of these tools to find serious vulnerabilities ...
  - ... keeping a low false negative rate, and ...
  - ... having a low false positive rate (non-exploitable and limited value exploits).

# Manual Assessment: FPVA Condor Results

## 15 significant vulnerabilities discovered

<http://www.cs.wisc.edu/condor/security/vulnerabilities>

- 7 implementation bugs
  - **easy to discover** - localized in code
  - use of troublesome functions:  
`exec, popen, system, strcpy, tmpnam`
- 8 design flaws
  - **hard to discover** in code - higher order problems
  - defects include:
    - injections, directory traversals, file permissions, authorization & authentication, and  
a vulnerability in third party library

# Automated Tools Results

	Coverity	Fortify	
Defects Found:	2,986	15,466	total
		3	critical
		2,301	hot
		8,101	warm
		5,061	info
Manual Defects Found:	1	6	total
	1	6	impl. bug
	0	0	design flaw

# Automated Tools Results

- Simple implementation bugs found
  - Coverity found 1
    - errs on the side of false negatives
    - only flags certain functions when input can be proven to come from untrusted sources
  - Fortify found 6
    - errs on the side of false positives
    - will always flag certain functions
- No design flaw defects found
- We could find no actual additional vulnerabilities in their reports.



# Manual Tool Comparison Study

- Showed limitations of current tools
- Emphasized the need for manual vulnerability assessment as a required part of a comprehensive security assessment
- Created a **reference set** of vulnerabilities to perform apples-to-apples comparisons
- Tests on additional tools by the NSA **show the same results.**
- Tests on additional software by UAB (on Crossbroker) **show the same results.**



# A Research Opportunity?

Study the vulnerabilities that the tools are not finding: can we formally characterize them?

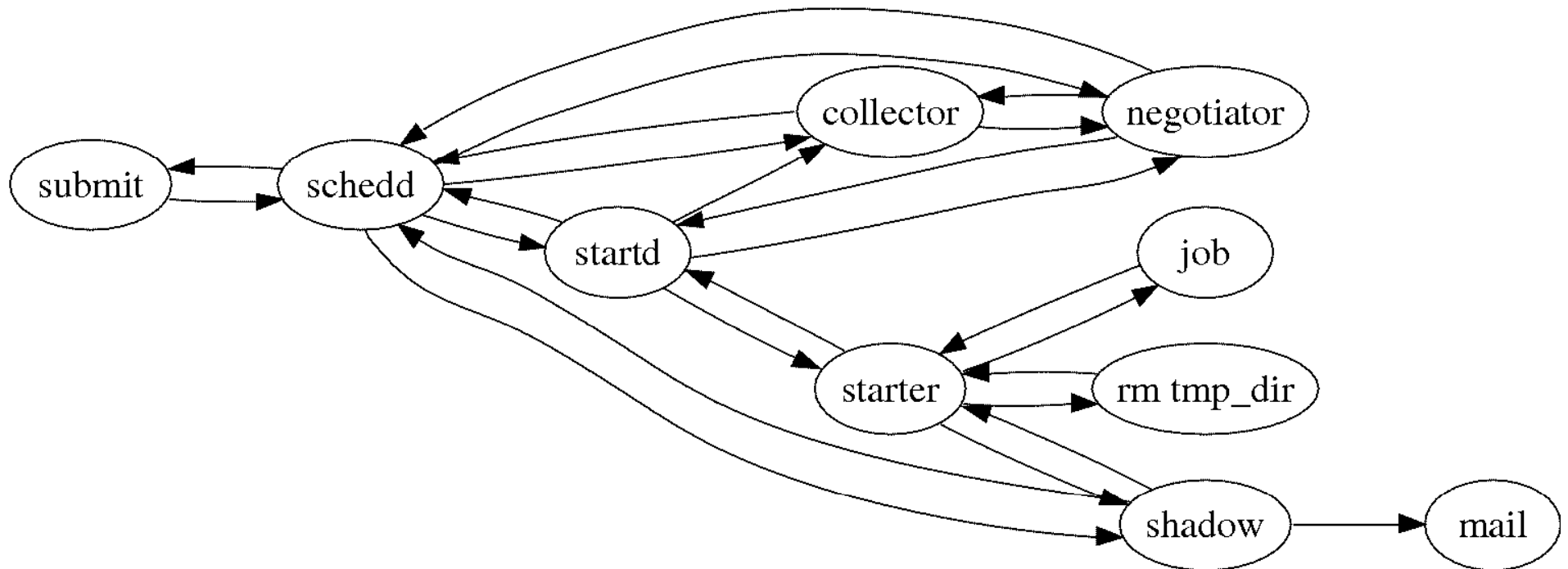
If we can characterize them, we can design detection algorithms. And from these algorithms, we can build tools.

Automate parts of the FPVA analysis task to make this go quicker.

Use techniques like "self propelled instrumentation" to extract architecture, privilege levels, and resources.

We (UW and UAB) are currently working on these problem areas.

# Automatically Extracted Architectural Information



# Some Experiences as Analysts

It is difficult to get a group started on conducting such assessments.

- There are many reasons to say "no".
- And, if you do say "yes" ...  
... when the first vulnerability reports come in - what will you do?

*Just say “no”.*

*(Nancy Reagan)*

# There are *Lots* of Reasons to Say No

“We use best practices in secure software design, so such an effort is redundant.”

*There's many a slip 'twixt cup and lip...*

*(old English proverb based on Erasmus)*



Even the best programmer makes mistakes.

- The interaction between perfect components often can be imperfect: *falling between the cracks*.
- Even in the best of cases, only works with formal specification and verification.

# There are *Lots* of Reasons to Say No

“We are really good at debugging and testing our own code, so this is redundant.”



*Testing your own code is about as effective as tickling yourself.* (Bart)

This is a bad idea for several reasons.

- Our ability to test is hampered by our biases and expectations.
- There is an inherent conflict of interest in evaluating your own work.



# There are *Lots* of Reasons to Say No

“It’s too expensive.”

*The best defense is a  
good offense.*  
(old sports adage)

*The only real defense is  
active defense.*  
(Mao)

- Yes, it is expensive.
- And, yes, if you are successful, you will only see an expense.
- However the cost to recover after a serious exploit is *prohibitive*.

# There are *Lots* of Reasons to Say No

“I’ll just run some automatic tools.”

*The era of procrastination, of half-measures, of soothing and baffling expedients, of delays is coming to its close. In its place we are entering a period of consequences.*

*(Winston Churchill, August 1941)*



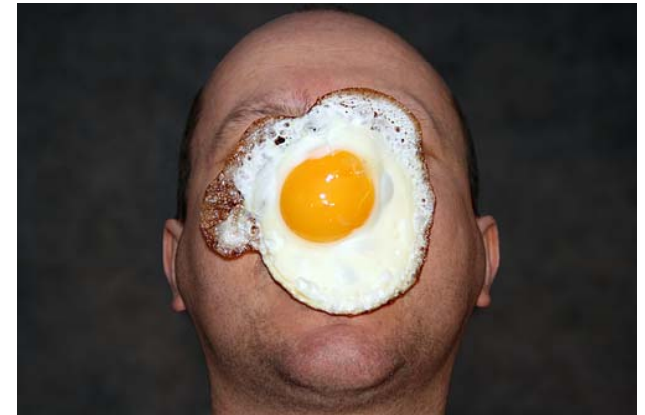
- Tools like Fortify and Coverity are worthwhile to use...
- ...however, don't let them give you a false sense of security. Our recent study demonstrates their significant weaknesses.

# There are *Lots* of Reasons to Say No

“If we report bugs in our software, we will look incompetent.”

*A life spent making mistakes is not only more honorable, but more useful than a life spent doing nothing.*

*George Bernard Shaw (1856 - 1950)*



- All software has bugs.
- If a project isn't reporting the bugs, either they are **not checking** or **not telling**.
- Our experience shows that users (and funding agencies) are more confident when you are checking and reporting.

*And the assessment team arrives...*



# During the Assessment

What makes our job harder:

- Incomplete or out-of-date documentation.
- Complex installation procedures, especially ones that are not portable and require weird configuration file magic.
- Lack of access to full source code.
- Lack of access to development team.

# During the Assessment

What you can expect from us:

- Our assessments follow the FPVA methodology.
- We work *independently*: crucial for an unbiased assessment.
- We will ask you lots of questions.
- We won't report any vulnerabilities until we're done...  
...however we *will* release our intermediate products - diagrams from the architectural, resource, and privilege analyses.
- It will take longer than you think...  
... we don't report a vulnerability until we can construct an exploit.



*And then the vulnerabilities arrive...*



# How do You Respond?

*When In Danger, When In Doubt, Run In Circles,  
Scream And Shout*



# How do You Respond? (really)

- Denial: “That’s just not possible in our code!”
- Anger: “Why didn’t you tell me it could be so bad?!”
- Bargaining: “We don’t have to tell anyone, do we?”
- Depression: “We’re screwed. No one will use our software and our funding agencies will cut us off.”
- Acceptance: “Let’s figure out how to fix this.”

# How do You Respond?

- Identify a team member to handle vulnerability reports.
- Develop a remediation strategy:
  - Study the vulnerability report.
  - Use your knowledge of the system to try to identify other places in the code where this might exist.
  - Study the suggested remediation and formulate your response.
  - Get feedback from the assessment team on your fix - very important for the first few vulnerabilities.
- Develop a security patch release mechanism.
  - This mechanism must be separate from your release feature/upgrade releases.
  - You may have to target patches for more than one version.

# How do You Respond?

Develop a notification strategy:

- What will you tell and when?
- Users are nervous during the first reports, but then become your biggest fans.
- Often a staged process:
  1. Announce the vulnerability, without details at the time you release the patch.
  2. Release full details after the user community has had a chance to update, perhaps 6-12 months later.
- Open source makes this more complicated!  
The first release of the patch reveals the details of the vulnerability.

# How do You Respond?

A change of culture within the development team:

- When security becomes a first-class task, and when reports start arriving, *awareness* is significantly increased.
- This effects the way developers look at code *and* the way that they write code.
- A major landmark: when your developers start reporting vulnerabilities that they've found on their own.



# An Ongoing Effort

Our team continues to:

- Assess new software
- Train analysts in the assessment and security coding techniques.
- Consult with organizations in this area.
- Research to improve the state of tools for both automated code analysis and faster manual analysis.

We encourage you to join in!

# Thank you.

# Questions?

<http://www.cs.wisc.edu/mist>

[bart@cs.wisc.edu](mailto:bart@cs.wisc.edu)

# What about Automated TOOLS?

- Everyone asks for them
- They may help but ...
  - ... they are definitely **not enough!**

# Our Piece of the Solution Space

A brief story... (con'd)

- This last point required us to do a careful study of these tools, in a way that had not been done previously.

Our suspicions as to the  
limitations of these tools were  
confirmed!



- And this has led to an interesting new research agenda.

# First Principles Vulnerability Assessment

## Understanding the System

### Step 1: Architectural Analysis

- Functionality and structure of the system, major components (modules, threads, processes), communication channels
- Interactions among components and with users

# First Principles Vulnerability Assessment

## Understanding the System

### Step 2: Resource Identification

- Key resources accessed by each component
- Operations allowed on those resources

### Step 3: Trust & Privilege Analysis

- How components are protected and who can access them
- Privilege level at which each component runs
- Trust delegation

# First Principles Vulnerability Assessment

## Search for Vulnerabilities

### Step 4: Component Evaluation

- Examine critical components in depth
- Guide search using:
  - Diagrams from steps 1-3
  - Knowledge of vulnerabilities
- Helped by Automated scanning tools (!)



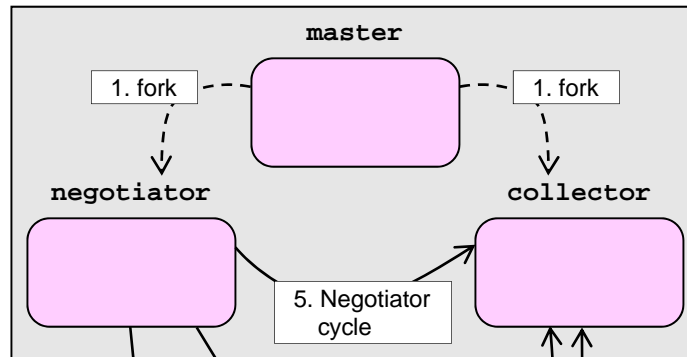
# First Principles Vulnerability Assessment Taking Actions

## Step 5: Dissemination of Results



- Report vulnerabilities
- Interaction with developers
- Disclosure of vulnerabilities

# Example Results: Condor

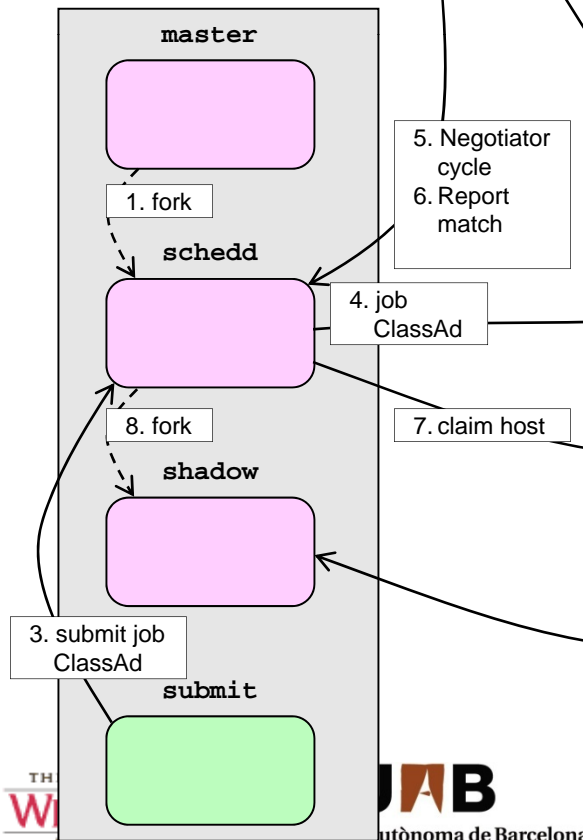
Condor execute host



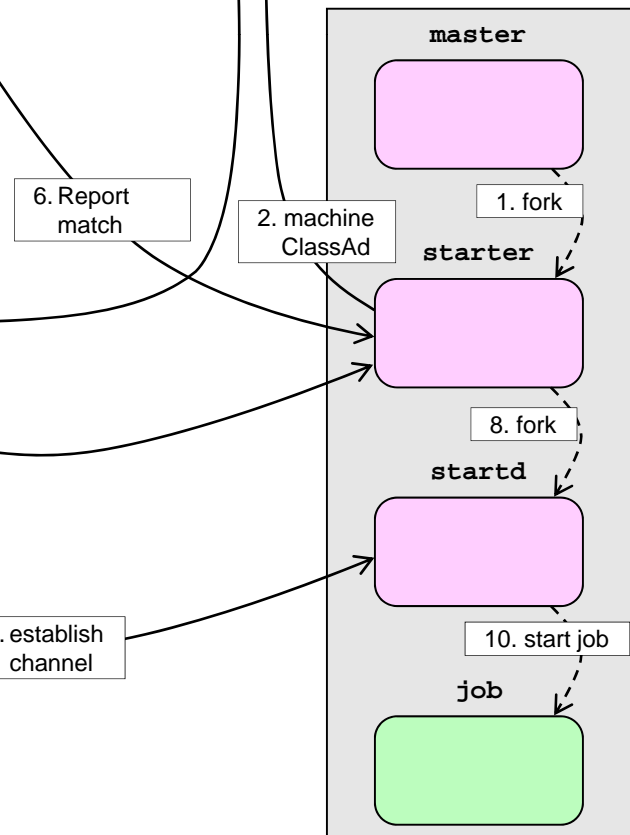
OS privileges

-  condor & root
-  user

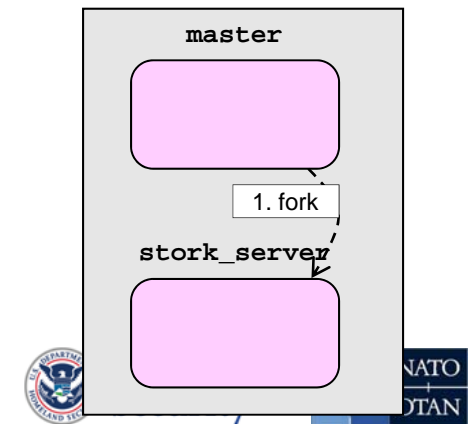
Condor submit host



Condor execute host

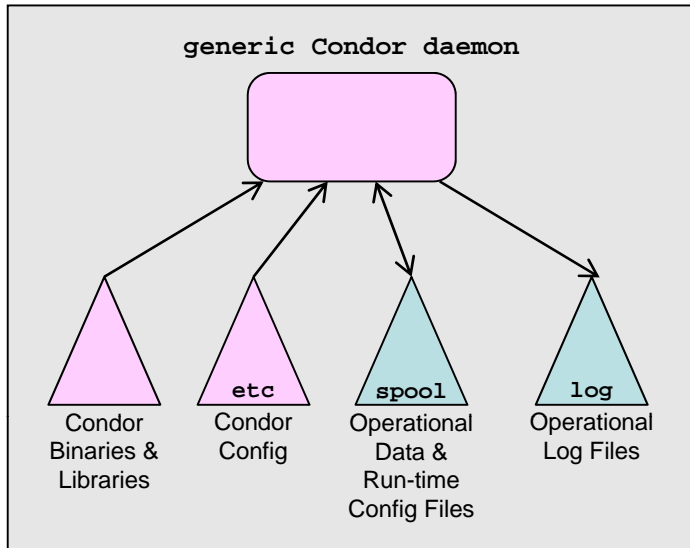


Stork server host

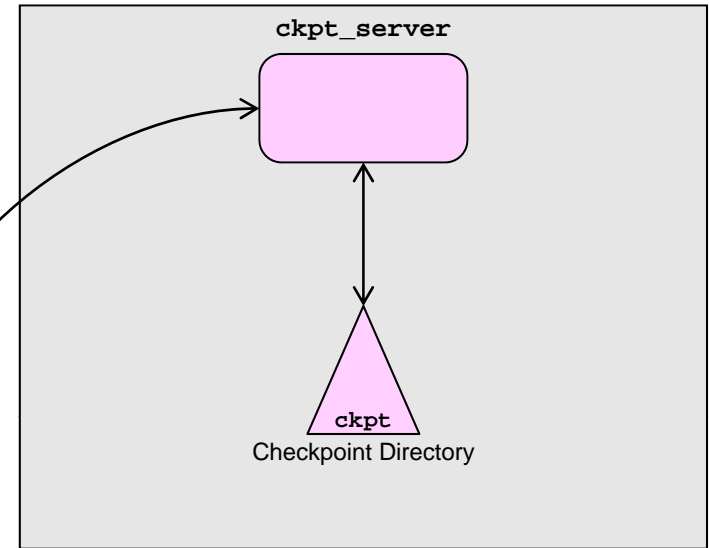


# Example Results: Condor

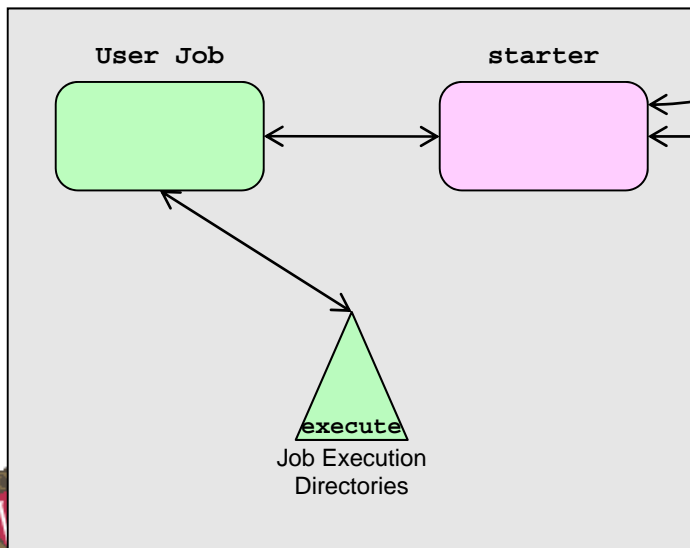
(a) Common Resources on All Condor Hosts



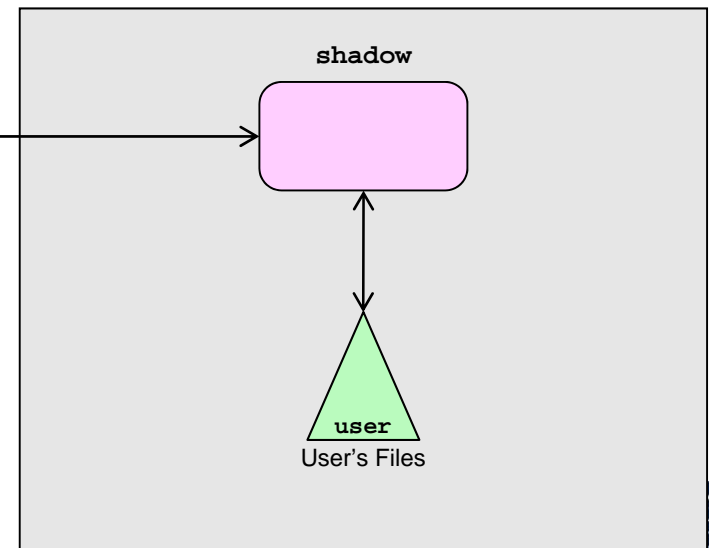
(b) Unique Condor Checkpoint Server Resources



(c) Unique Condor Execute Resources



(d) Unique Condor Submit Resources



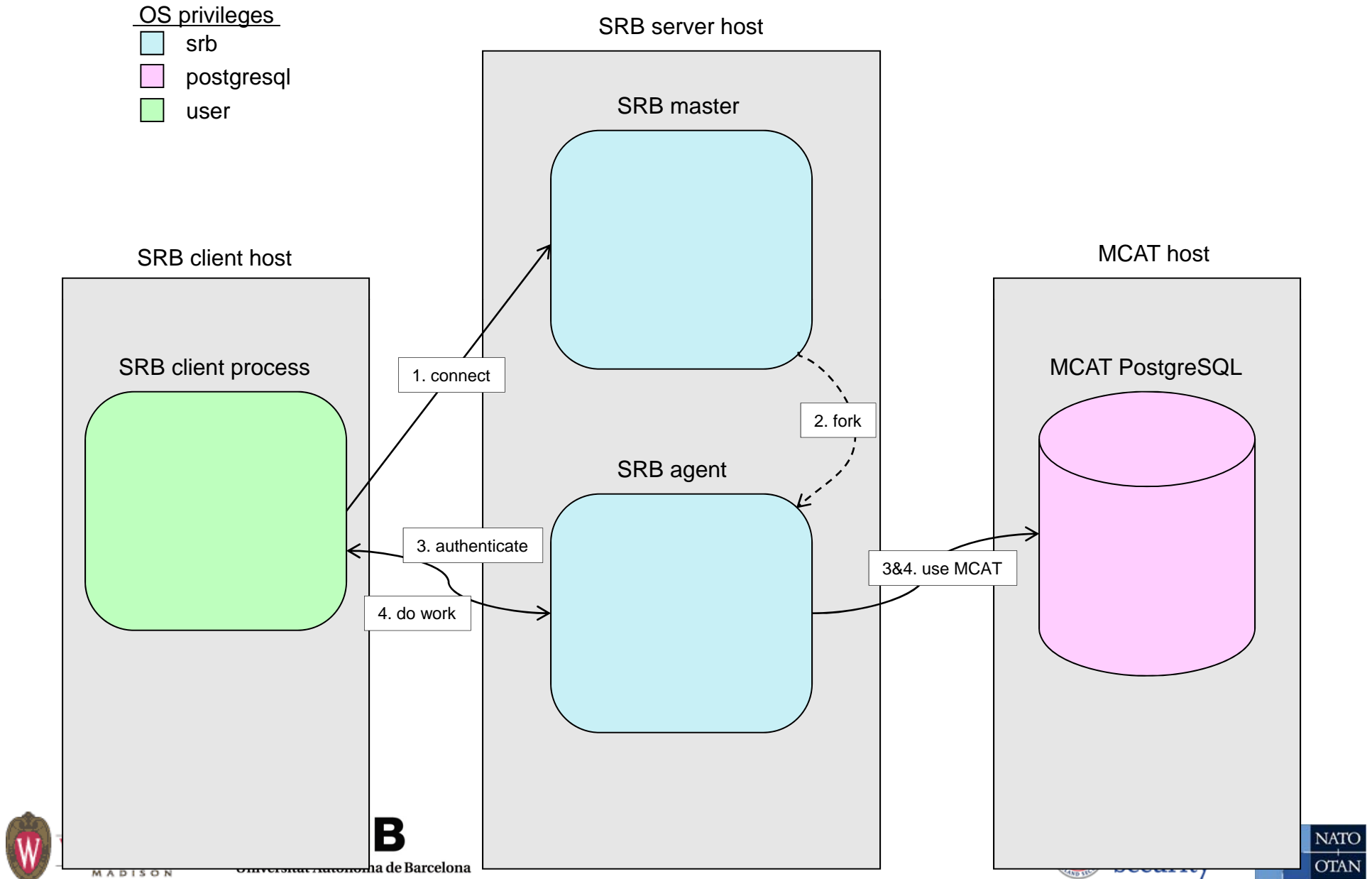
## OS privileges

- condor
- root
- user

Send and Receive Checkpoints  
(with Standard Universe Jobs)




System Call Forwarding and Remove I/O  
(with Standard Universe Jobs)

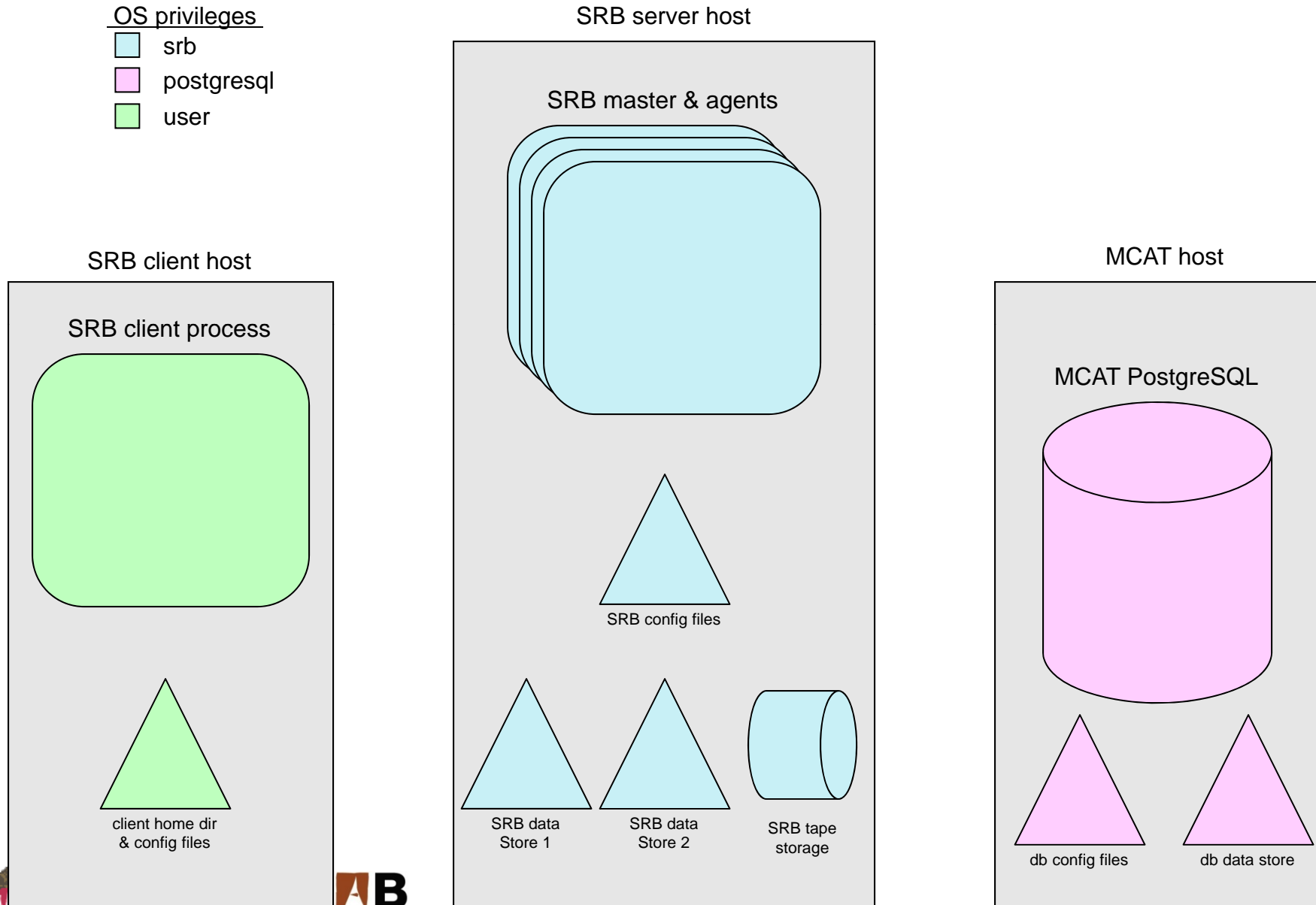
# Example Results: SRB

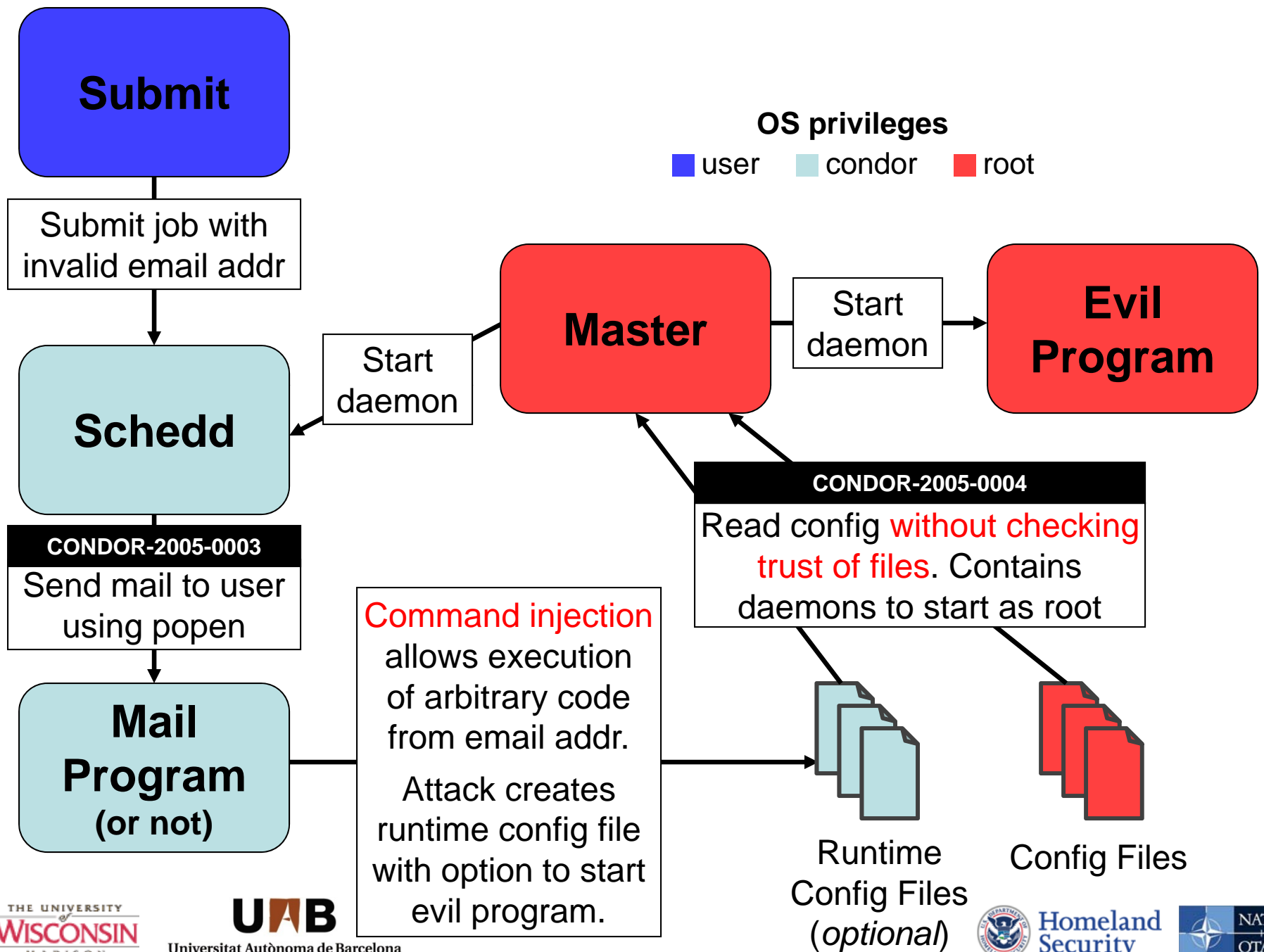


# Example Results: SRB

## OS privileges

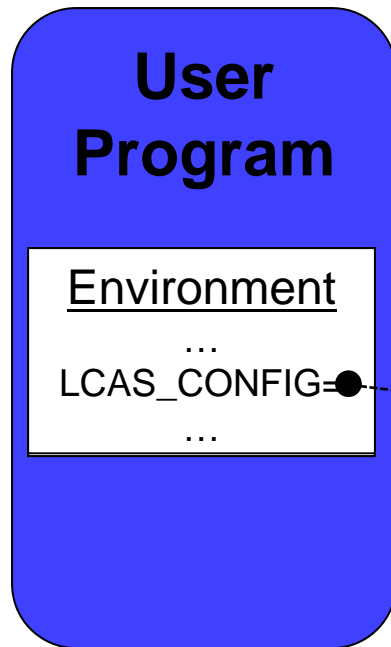
-  srb
-  postgresql
-  user



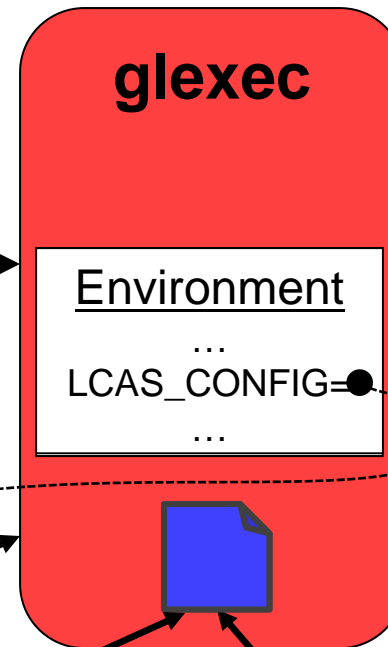


GLExec-2005-0003

User **controls shared library** loaded into gLExec using environment variable



Exec glexec  
(setuid root)



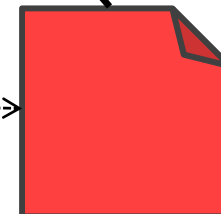
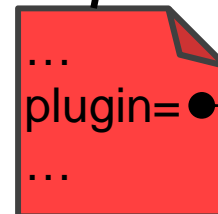
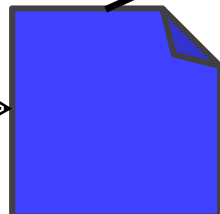
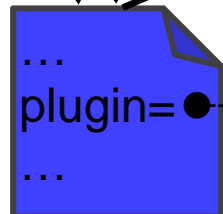
OS privileges  
■ user ■ root

Read Config

Load Plugin

Read Config

Load Plugin



Attacker LCAS  
Config File



Attacker  
Shared Library

Default LCAS  
Config File

LCAS Plug-in  
Shared Library





GRATIA-CONDOR-2010-0003

Data in history files is used  
to create a Python program  
allowing a **code injection**

OS privileges

■ user ■ condor ■ root

**Submit**

Submit job with  
invalid attribute

**Schedd**

Write job  
history files

Condor Job  
History Files

Read attributes  
of each history  
file to create  
records in Gratia

**Gratia  
Condor  
Probe**

Create Python program  
from unchecked attributes  
allowing a code injection

**Python**

# We do Find Vulnerabilities

System	Origin	Language(s)	Size (loc)	Vuln. Found
Condor	Wisconsin	C++	600K	15
SRB	SDSC	C, SQL	275K	6
MyProxy	NCSA	C	25K	5
gLExec	Nikhef	C	43K	5