Pegasus-CTSC Engagement Final Report

May 13, 2013

For Public Distribution

Randy Heiland, Scott Koranda, Von Welch

About CTSC

The mission of the Center for Trustworthy Scientific Cyberinfrastructure (CTSC, trustedci.org) is to improve the cybersecurity of NSF science and engineering projects, while allowing those projects to focus on their science endeavors. This mission is accomplished through one-on-one engagements with projects to address their specific challenges; education, outreach, and training to raise the state of security practice across the scientific enterprise; and leadership on bringing the best and most relevant cybersecurity research to bear on the NSF cyberinfrastructure research community.

Acknowledgments

CTSC's engagements are inherently collaborative; the authors wish to thank the Pegasus team, including Ewa Deelman, Karan Vahi, Mats Rynge, and Gideon Juve, for the collaborative effort that made this document possible. The authors also wish to thank Zach Miller and Dan Bradley from UW-Madison for their helpful insights regarding HTCondor, *ssh-agent* and *condor_ssh_to_job*, and Jim Basney for his insights into OpenSSH.

Craig Jackson of the CTSC project made significant contributions to this report.

This document is a product of the Center for Trustworthy Scientific Cyberinfrastructure (CTSC). CTSC is supported by the National Science Foundation under Grant Number OCI-1234408. For more information about the Center for Trustworthy Scientific Cyberinfrastructure please visit: http://trustedci.org/. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

Cite this work using the following information:

R.W. Heiland, S. Koranda, V.S. Welch, "Pegasus-CTSC Engagement Final Report," Center for Trustworthy Scientific Cyberinfrastructure, trustedci.org, May 2013. Available: http://hdl.handle.net/2022/15562

1 Executive Summary

The Center for Trustworthy Scientific Cyberinfrastructure (CTSC) engages with NSF-funded projects to address their cybersecurity challenges. This document presents the results of one such engagement with the Pegasus project, a workflow management system for computational science. Pegasus workflows typically operate across distributed resources and sometimes need to stage data files between compute resources to or from storage resources. When such staging requires secure shell (SSH), Pegasus' current practice is to send a private key with the workflow to perform a secure copy. The goal of this engagement was to examine this practice and recommend any possible improvements from the perspective of cybersecurity. We provide three recommendations to the Pegasus team to improve current practice: (1) If system administrators are willing, have them deploy a mechanism that supports security delegation, such as Kerberos or GSI; (2) provide assistance to users in using SSH's ability to impose restrictions in the *authorized_keys* file to limit the privileges of SSH keys used for workflows; and (3) utilize ssh-agent to minimize exposure of SSH credentials in the workflow by avoiding writing those credentials to the filesystem. We also describe alternatives we considered but do not recommend.

2 Background and Problem Statement

Pegasus [1] is a workflow management system (WMS) for scientific workflows. A workflow will, typically, operate across distributed compute and storage resources. Pegasus uses the HTCondor [2] high throughput computing software system for its distributed computing needs, scheduling jobs to run on available compute resources. Pegasus supports a variety of data transfer protocols and associated credential types for authentication: X.509 grid proxies, Amazon AWS S3 keys, iRods password, and SSH keys.

The goal of the Pegasus-CTSC Engagement was to examine a particular use case of the Pegasus WMS where data needs to be copied to or from the workflow to a storage system via secure shell (SSH) and determine what recommendations could be made to improve the current practice from the perspective of cybersecurity.

In our representative use case, shown in Figure 1, a user submits a workflow from a Submit Host (SH) to one or more worker nodes (WNs) which need to copy data to or from a remote Staging Site (SS). In this use case the WNs and SS do not share a filesystem, but instead access it over the SSH protocol. The SS may be copying data to or from other sites, but this is unimportant for our use case, and this copying is done via mechanisms other than Pegasus.

The current practice in Pegasus is to have the user supply his or her SSH credential (private key), which Pegasus includes with the workflow so that it can be used by a WN to perform a secure copy via SSH (scp). Users are encouraged to create a separate SSH credential to be used only by Pegasus to mitigate any misuse of that credential.

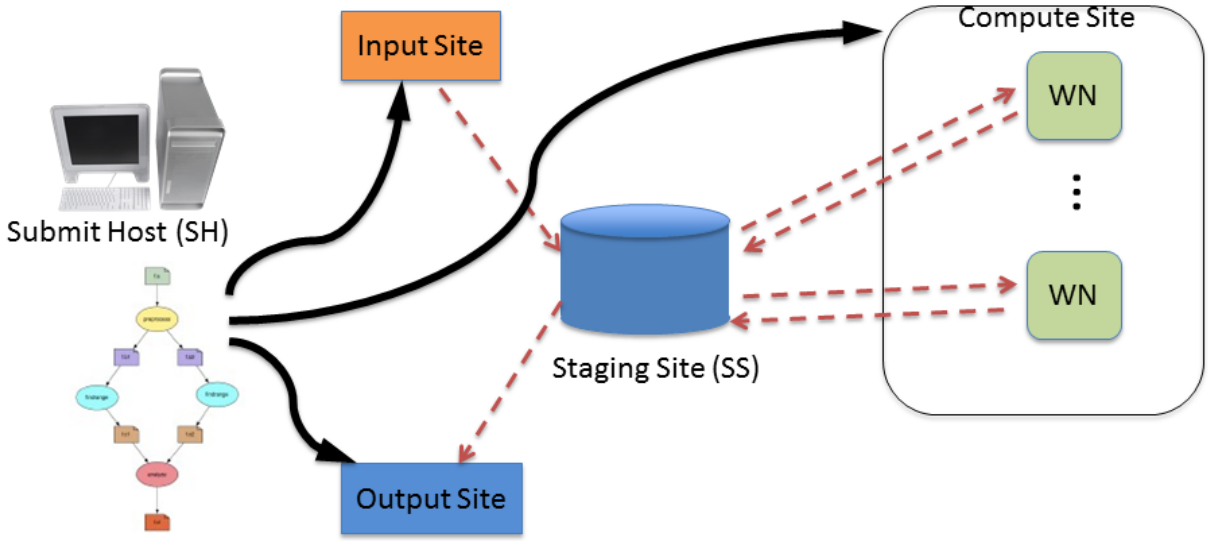

Figure 1. Use case: non-shared file system; scp to stage data. Solid arrows represent job submission by workflow; dashed arrows represent data movement.

3 Relevant Technologies

In researching alternatives to the current practice, we discovered or were already cognizant of a number of technologies relevant to Pegasus' usage of secure shell (SSH) [3] and its secure copy (scp).

We primarily considered SSH using public key authentication, in which a public-private cryptographic key pair is used to authenticate the client. On the server site, an `authorized_keys` file is used to limit which keys can access an account and restrict that access to certain commands[1].

Typically SSH private keys are encrypted on disk with a passphrase for security; this can create the need for repeated passphrase entry by and be onerous for users. To ease repeated use, *ssh-agent*[2], is a program that can decrypt a SSH private key once and then hold it in memory, allowing its repeated use while providing a more secure alternative to having no encryption.

An SSH-related command in the HTCondor system is called *condor_ssh_to_job*[3]. This command lets a user create an SSH session to a running job. We note it does not work for "standard universe" jobs. Section 7, Other Use Cases, discusses *condor_ssh_to_job* in more detail.

[1] refer to the "AUTHORIZED_KEYS FILE FORMAT" section of the sshd man page, http://www.openbsd.org/cgi-bin/man.cgi?query=sshd
[2] http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-agent&sektion=1
[3] http://research.cs.wisc.edu/htcondor/manual/current/condor_ssh_to_job.html

4 Security Criteria

Our first step was to determine our criteria from a cybersecurity perspective in judging alternatives to enabling authentication from the Worker Node to the Staging Site. We identified the following:

1. Minimize the exposure of any passwords or SSH private keys.
    a. Avoid transmitting such authentication material over the network entirely. Transmitting such authentication material is frowned upon, making any scheme using it difficult to socialize, because it exposes the material to potential interception both during transition and on other computers to which it is transmitted.
    b. If such authentication material is transmitted, reduce that exposure by, whenever possible, encrypting it and avoiding storing it on filesystems.
2. Minimize the privileges granted to any authentication passwords or keys provided to the workflow. In addition to reducing the exposure, and hence chance of misuse of authentication material, seek to reduce the impact of that misuse by reducing the privileges associated with that material.
3. Maximize ease-of-use for both the workflow initiator and the administrators of the Worker Node and Staging Site.
4. Minimize complexity of the resulting system, to maximize ease of implementation, deployment and explanation.

We note these criteria represent a tension: increasing security, as indicated by criteria 1 and 2, means more effort; however, criteria 3 and 4 are about minimizing effort for the different parties involved. Hence our evaluation of different alternatives had much to do with balancing the tradeoff of more security versus the added effort, and who would be burdened by that additional effort.

5 Recommendations

In this section we provide our recommendations to the Pegasus team with regards to their current practice. Alternative approaches that we considered, but did not choose to recommend, are included in the subsequent section.

As previously described, the Pegasus WMS currently directs its users to include an SSH credential with the workflow to handle the use case in question. This credential is then used by a Worker Node to authenticate to a Staging Site. In the Pegasus reference manual, it is recommended that a user create SSH credentials distinct from credentials used for other activities, in order to minimize the privileges of those credentials.

Recommendation #1: Have system administrators deploy an authentication mechanism that supports delegation. Ideally one would utilize one of the alternatives to SSH, namely Kerberos [4] and Grid Security Infrastructure (GSI) [5], which support delegation. These systems would allow a user to securely transmit a temporary token (ticket in the case of Kerberos, proxy certificate in the case of GSI) from the Submit Host to the Worker Node(s) which would allow access to the Staging Site. However, these alternatives must be installed and supported by the system administrator of the Worker Nodes and Staging Site and we understand there will be administrators who choose not to do so, hence we continued exploring other options.

Recommendation #2: Restrict any SSH credentials use for the workflow using SSH's *authorized_keys* file. Augmenting the Pegasus' team current recommendation of creating temporary, workflow-specific SSH credentials, we recommend the Pegasus team provide guidance (or software tools) to help users create restrictions for the *authorized_keys* file on the Staging Site to minimize the privileges of SSH credentials created for Pegasus workflows. As we described previously in Section 3, Relevant Technologies, SSH credentials must be registered into an *authorized_keys* file to be granted access; in this case, that file would be on the Staging Site. SSH allows the *authorized_keys* to contain additional restrictions, *e.g.* allowing a credential to be restricted to specific subdirectories. These restrictions are described in more detail in Appendix A.

Recommendation #3: Use ssh-agent to avoid storing SSH credential on the Worker Node filesystem. Currently SSH credentials are stored on the filesystem of the Worker Node. By instead storing the SSH credential in ssh-agent, this storage on the filesystem could be avoided, providing some additional security, *e.g.* if the disk space used by a job was failed to be cleaned up, the SSH credential would not persist there. We do note this would require some development by the Pegasus team and might require support from the underlying HTCondor system to completely avoid writing the credential to disk. Since the Pegasus team has indicated they do not wish to pursue this option, we have not explored all the details of this recommendation.

6 Other Alternatives Explored

In this section we discuss other alternative approaches we explored for the given use case, but do not put forward as recommended.

Alternative 1: Transmitting the User's long-term SSH keys with the workflow.

- Pros: Easy, no additional work required.
- Cons: User's long-term credentials are exposed and those credentials have full privileges of the user on Staging Site and any other service to which the user has access.
- Assessment: High ease of use, but low security. Broader exposure of the user's long-term credentials will be undesirable to both the administrators of the services and a large subset of the user community.

Alternative 2: Create a new set of SSH credentials for each instantiation of a workflow. The user, or more likely the Pegasus software, generates a new set of SSH credentials for each workflow *submission*, grants them privileges on the SS and, after the workflow completes, cleans up those privileges.

- Pros: Credentials are both limited in privileges and reduced in exposure. By automating the process of creating SSH keys, you remove any uncertainty regarding whether the user has created, as recommend, a separate set of credentials for running the workflow.
- Cons: Highly complex. Automatic modification of `authorized_keys` file may be fragile. Workflows that fail could start "cluttering" up the `authorized_keys` file on the SS. Benefit over workflow-specific credentials is unclear.
- Assessment: Higher security but low ease of use. Overly complicated and fragile. Would need significant testing. Unlikely that benefits outweigh costs.

Alternative 3: Avoid storing the SSH credential on the Worker Node filesystem by storing it in an environment variable. Similar to our Recommendation #3 in the previous section, this alternative seeks to avoid writing the SSH credential to the file system on the Worker Node, the difference is that the credential is instead stored in an environment variable and then retrieved by Pegasus when needed.

- Pros: Moderate ease of use; may provide some additional security.
- Cons: Would require additional development by the Pegasus team. The fatal flaw is that environment variables and their values are not private on all operating system and can be exposed, *e.g.*, via the terminal command "ps auxe" on OSX.
- Assessment: Effort put on Pegasus development team. The public nature of environment variables renders the benefits of this approach too unreliable.

Alternative 4: Provide credentials to Worker Node via ssh-agent forwarding. As previously described in Section 3, ssh-agent allows for repeated authentication using SSH credentials. SSH supports remote access to an ssh-agent over a SSH connection using a technique called *forwarding*. In theory, an SSH connection from the Submit Host to a Worker Node could be used via forwarding to allow the use of an ssh-agent running on the Submit Host to allow Worker Nodes to authenticate to the Staging Site. There are several challenges with this approach:

1. Forwarding requires an active, open SSH connection from the Submit Host to the Worker Node. This means the Submit Host must be available on the network (it cannot be a laptop that is suspended and put into a backpack) and there must be network connectivity between the Submit Host and the Worker Node (firewalls can cause issues here).
2. Establishing the SSH connection to do the forwarding is tricky. Having the Worker Node connect back to the Submit Host means the Worker Node needs credentials, which in effect is the problem the use case is trying to solve. Having the Submit Host connect to the Worker Node means the job on the Worker Node must start a SSH daemon and Pegasus running on the Submit Host must somehow know when that daemon is started, which could be a challenge if the workflow must wait in a job

queue for an indeterminate length of time. condor_ssh_to_job, described in Section 3, could be of help here, removing the need for the workflow to start a SSH daemon. We exchanged emails with the HTCondor developers, who indicated this might be possible, but we have not experimentally determined if condor_ssh_to_job supports forwarding of an ssh-agent.

3. If a workflow spans many Worker Nodes, there could be scaling issues with the Submit Host having to maintain a large number of SSH connections.

- Pros: Would avoid credentials having to be transferred to Worker Node altogether. Use of condor_ssh_to_job might simplify implementation.
- Cons: Would require additional development by the Pegasus team. condor_ssh_to_job works only for vanilla, vm, java, local, and parallel universe jobs (not standard universe jobs). Fragility is a concern considering the added demands for stable networking and handling a large number of Worker Nodes.
- Assessment: Effort put on Pegasus development team. A complicated approach that would need prototyping and testing to assess in practice.

Alternative 5: Use condor_ssh_to_job to deliver an SSH credential to a Worker Node once the workflow is running. Instead of including the SSH credential with the workflow description, wait for process(es) to be instantiated on the Worker Nodes and then deliver SSH credentials to them using condor_ssh_to_job.

- Pros: Reduces the exposure of the SSH credentials by not including them in the workflow description.
- Cons: Would require additional development by the Pegasus team. Has the challenge that the Submit Host needs to know when and where jobs are running.
- Assessment: Effort put on Pegasus development team. A complicated approach that would need prototyping and testing to assess in practice.

Alternative 6: Have system administrators deploy SSH trusted host configuration between the Staging Site and the Worker Nodes. SSH supports the ability for a system administrator to configure one host to trust another, that is, allow any user on another host to access their account of the same name on the local host without authentication materials. If the administrator of the Staging Service configured the SSH daemon on that system to trust the Worker Nodes, this could allow Pegasus jobs running on those systems to access the Staging Service without needing an SSH credential.

- Pros: Makes things easy for the user and Pegasus team. No SSH credentials would have to be exposed.
- Cons: Requires Staging Site to highly trust the Worker Nodes and requires alignment of account names between those systems.
- Assessment: Effort put on system administrators. Due to the cons, this is an approach that is unlikely to be readily adopted by anyone who has not already implemented it.

7 Other Use Cases

In addition to the main use case discussed in this document - *i.e.*, using a traditional Staging Site with SSH authentication - we briefly explored other similar use cases for this engagement.

Amazon Simple Storage Service (S3)

While not the primary focus of this engagement, we explored data staging using Amazon S3, its credentials mechanism and access controls. The interface to S3 is handled by the Python script, pegasus-s3[4], using the boto[5] Python module. Although we have not completely explored this space, it seems possible to use the `boto.s3.bucket.set_policy` function to programmatically limit access to buckets (containers for files). Appendix B contains more details. It seems that S3 credentials are analogous to SSH keys, in that S3 provides an "access key" (~ SSH public key) and a "secret access key" (~ SSH private key). Therefore, some of the recommendations that we made for SSH keys (*e.g.*, creating/maintaining temporary, workflow-specific keys) also apply to S3 keys.

Pegasus as a Service

The idea of "Pegasus as a service" with access via a hosted interface is a natural evolution for the Pegasus WMS and was discussed in a meeting with Pegasus and CTSC team members. Two interfaces are currently being considered - via a web browser and via REST[6]. The service model involves different security approaches from those we have discussed thus far. One widely used approach is OAuth[7]. Since Globus Online currently uses OAuth (for MyProxy)[8] and Globus Online will likely be a CTSC engagement, we should have more to say about this in the near future. (OAuth is also used by Science Gateways.[9])

[4] https://github.com/pegasus-isi/pegasus/blob/master/bin/pegasus-s3
[5] http://docs.pythonboto.org/en/latest/
[6] https://en.wikipedia.org/wiki/Representational_state_transfer
[7] http://oauth.net/
[8] https://www.globusonline.org/news/announcement/2011/11/15/myproxy-oauth/
[9] http://www.sciencegatewaysecurity.org/oauth-for-myproxy

8 References

[1] E. Deelman, G. Singh, M.H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, D.S. Katz. “Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems,” *Scientific Programming Journal, Vol 13(3),* 2005, pp. 219-237.

[2] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, Vol. 17, No. 2-4, pp. 323-356, February-April, 2005.

[3] D.J. Barrett, R.E. Silverman, and R.G. Byrnes, *SSH: The Secure Shell (The Definitive Guide)*, O'Reilly 2005 (2nd edition). ISBN 0-596-00895-3. Available: http://docstore.mik.ua/orelly/networking_2ndEd/ssh/index.htm

[4] B.C. Neuman and T. Ts'o. “Kerberos: An Authentication Service for Computer Networks,” *IEEE Communications*, 32(9):33-38. September 1994. Available: http://gost.isi.edu/publications/kerberos-neuman-tso.html

[5] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke. “A Security Architecture for Computational Grids,” *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pp. 83-92, 1998. Available: http://www.globus.org/ftppub/globus/papers/security.pdf

Appendix A: ssh authorized_keys restrictions

We simulate the Pegasus use case of copying files from a Worker Node (WN, “gw86” below) to a Staging Site (SS, “gw57” below) via ssh/scp with restrictions imposed via the SSH `authorized_keys` file.

authorized_keys/command

To begin, generate Pegasus-specific SSH keys on a Submit Host:

```
[heiland@SubmitHost]$ ssh-keygen -f ~/.ssh/id_rsa.pegasus
<no passphrase>
```

This generates the following private and public keys:

```
-rw-------. 1 heiland heiland 1675 Mar 19 16:19 id_rsa.pegasus
-rw-r--r--. 1 heiland heiland 407 Mar 19 16:19 id_rsa.pegasus.pub
```

On the Staging Site, in your .ssh subdirectory, create an “authorized_keys” file that contains the “command” keyword pointing to a wrapper shell script (or Python script), followed by the Pegasus-specific public key generated above:

```
[heiland@gw57 .ssh]$ more authorized_keys
# invoke a shell script for Pegasus-specific SSH public key
command="~/.ssh/ssh-wrapper.sh" <Pegasus-specific-public-key>
```

Create a wrapper shell script which will only allow a “scp” command (in association with the Pegasus-specific key):

```
[heiland@gw57 .ssh]$ more ssh-wrapper.sh
#!/bin/bash
# ssh-wrapper.sh

case $SSH_ORIGINAL_COMMAND in
      'scp'*)
      $SSH_ORIGINAL_COMMAND
      ;;
      *)
      echo "Invalid command"
      ;;
esac
```

NOTE: only a *single* command is allowed for each block of the “case” statement. The scp command that you want to allow is conveniently stored in the SSH_ORIGINAL_COMMAND environment variable.

Set appropriate permissions on the Staging Site’s .ssh subdir and files contained therein:

```
[heiland@gw57 .ssh]$ cd ~
[heiland@gw57 ~]$ chmod -R 700 .ssh  # from home dir

→
-rwx------. 1 heiland heiland 943 Mar 20 11:30 authorized_keys
-rwx------. 1 heiland heiland 206 Mar 20 10:28 ssh-wrapper.sh
```

Simulate a scp from WN to SS:

```
[heiland@gw86 .ssh]$ scp -q -i ~/.ssh/id_rsa.pegasus /home/heiland/foo.txt
heiland@gw57.iu.xsede.org:/home/heiland
```

On an actual WN, the private key would be sent in the workflow, copied to disk somewhere, and then that path specified, e.g.:

```
[WN]$ scp -q -i /tmp/fred/id_rsa.pegasus /home/heiland/foo.txt
heiland@gw57.iu.xsede.org:/home/heiland
```

Rf. “scp” function in the “pegasus-transfer” Python script for additional arguments to scp: https://github.com/pegasus-isi/pegasus/blob/master/bin/pegasus-transfer

Verify file is copied:

```
[heiland@gw57 .ssh]$ ls -l ..
-rw-rw-r--. 1 heiland heiland 4 Mar 20 09:53 foo.txt
```

Note that trying to issue a command other than “scp” will fail, e.g. trying to do a “ls”:

```
[heiland@gw86 .ssh]$ ssh -i ~/.ssh/id_rsa.pegasus heiland@gw57.iu.xsede.org
ls
Invalid command
```

Verify that you can still ssh to the remote machine (using personal SSH keys, not Pegasus-specific):

```
[heiland@gw86 .ssh]$ ssh heiland@gw57.iu.xsede.org
heiland@gw57.iu.xsede.org's password:
Last login: Tue Mar 19 16:13:01 2013 from gw86.iu.xsede.org
[heiland@gw57 ~]$
```

To avoid having to type your password, append your (personal) public key to the end of the SS’s ~/.ssh/authorized_keys:

→

```
[heiland@gw57 .ssh]$ more ~/.ssh/authorized_keys
command="~/.ssh/ssh-wrapper.sh" ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEArQ0pCM+RCqJvS2sItmiQp5TWaYaGxN9GrCEMi2
```

```
GN2MENQDmReR/jGL2V9+yTgVvQjOPfUEF8ot2RpiiesALEIPz1JM2Pp4wfGKHOdkMOkSujWA/slLr
MTcHJqvjGAs24JeUOt+3CHI7Hf3hCI
yDcNP0EV3OIfLP04uMGWbsMUj7RCUrDdAuoRB+gQ64Wwb/lnADVS82YujM2U23YMkHFjexBsXVT0I
D7oR0K7PdsZZa/+oCshglNJddsqqTy
BQ6jtejnhGoqpi1/abPk74lxNrVXCh3SfKouwMVxJ1I5erHsp4Znj+VKCmYZmnTfsxxUqd8YHBRLq
a4vXAtGWuSZ3w== heiland@gw86.iu.xsede.org
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAxSxLqeSkpIy7ttj7hg1WQ/yW4OxvhpCxkj+kX3EbDJ3LnSSF/
pPyrpQHF8wQMn9Kk66FWg
PbCA+m/eJpo/XsWoEceMyqq8105OlCd/w72a02PH6sxWwgHAHyBrcJ2eMxQEY7convNPVS1vbHspr
7BolGRDufzohzbdtV9MIpZcWQIOd3J
m5e0fm5Q1xbrEJOVGSA/LRFANqWClgZG/qqXG1wvvOTmY5FJfnGn7G8128sjfgs3KxEJVEVx8qFrk
bX3nuHDZzv571W0dqAHisysVdq1Enb
pfmr9P/lHNYvmwwdLubujzsPiur6oIxdWD6fHmfJXsPpE+ldVMUUyzfCJw==
heiland@gw86.iu.xsede.org
```

Restrict access to subdirectory

The following version of `ssh-wrapper.sh` will go further than the version above by restricting commands to “scp”, preventing an “upward” directory path (“..”), and restricting access to one specified subdirectory (plus, keeping a log file):

```
#!/bin/sh

cmd="$SSH_ORIGINAL_COMMAND"
echo "`date` ($SSH_CLIENT): $cmd" >> ssh-command-log # log all commands

# check that command doesn't involve upwards path components in
# any location to prevent, for instance, scp -t <path>/../
echo "$cmd" | fgrep '..' >/dev/null && echo "Forbidden command due to '..':
$cmd" && exit

case "$cmd" in
# scp\ *) exec $cmd ;;
 scp\ *-t\ /home/heiland/valid_subdir) exec $cmd ;;
 scp\ *-t\ /home/heiland/valid_subdir/*) exec $cmd ;;
 *) echo "Forbidden command: $cmd" ;;
esac
```

This will succeed:

```
[heiland@gw86 .ssh]$ scp -q -i ~/.ssh/id_rsa.pegasus /home/heiland/foo.txt
heiland@gw57.iu.xsede.org:/home/heiland/valid_subdir
```

and this will too:

```
[heiland@gw86 .ssh]$ scp -q -i ~/.ssh/id_rsa.pegasus /home/heiland/foo.txt
heiland@gw57.iu.xsede.org:/home/heiland/valid_subdir/bar.txt
```

However, this will not succeed as it lacks the “/valid_subdir”:

```
[heiland@gw86 .ssh]$ scp -q -i ~/.ssh/id_rsa.pegasus /home/heiland/foo.txt
heiland@gw57.iu.xsede.org:/home/heiland
```

Appendix B: Amazon AWS Security

We have excerpted sections from two documents on AWS security, an overview whitepaper[10] and another on credentials[11], that appear relevant to this engagement. In addition, we provide a simple example using the boto Python client to AWS that is used by Pegasus to access S3 resources.

Glossary

***Credentials**: Items that a user or process must have in order to confirm to AWS services during the authentication process that they are authorized to access the service. AWS credentials include the Access Key ID and Secret Access Key as well as X.509 certificates and multi-factor tokens.*

***Access Key ID**: A string that AWS distributes in order to uniquely identify each AWS user; it is an alphanumeric token associated with your Secret Access Key.*

***Secret Access Key**: A key that AWS assigns to you when you sign up for an AWS Account. To make API calls or to work with the command line interface, each AWS user needs the Secret Access Key and Access Key ID. The user signs each request with the Secret Access Key and includes the Access Key ID in the request. To ensure the security of your AWS account, the Secret Access Key is accessible only during key and user creation. You must save the key (for example, in a text file that you store securely) if you want to be able to access it again.*

***X.509**: In cryptography, X.509 is a standard for a Public Key Infrastructure (PKI) for single sign-on and Privilege Management Infrastructure (PMI). X.509 specifies standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm. Some AWS products use X.509 certificates instead of a Secret Access Key for access to certain interfaces. For example, Amazon EC2 uses a Secret Access Key for access to its Query interface, but it uses a signing certificate for access to its SOAP interface and command line tool interface.*

Amazon Simple Storage Service (S3) Security

Amazon S3 allows you to upload and retrieve data at any time, from anywhere on the web. Amazon S3 stores data as objects within buckets. An object can be any kind of file: a text file, a photo, a video, etc. When you add a file to Amazon S3, you have the option of including metadata with the file and setting permissions to control access to the file. For each bucket, you can control access to the bucket (who can create, delete, and list objects in the bucket), view access logs for the bucket and its objects, and choose the geographical region where Amazon S3 will store the bucket and its contents.

[10] http://awsmedia.s3.amazonaws.com/pdf/AWS_Security_Whitepaper.pdf
[11] http://docs.aws.amazon.com/AWSSecurityCredentials/1.0/AboutAWSCredentials.html#QuickStart

Data Access

Access to data stored in Amazon S3 is restricted by default; only bucket and object owners have access to the Amazon S3 resources they create (note that a bucket/object owner is the AWS Account owner, not the user who created the bucket/object). There are multiple ways to control access to buckets and objects:

- ***Identity and Access Management (IAM) Policies**. AWS IAM enables organizations with many employees to create and manage multiple users under a single AWS Account. IAM policies are attached to the users, enabling centralized control of permissions for users under your AWS Account. With IAM policies, you can only grant users within your own AWS account permission to access your Amazon S3 resources.*
- ***Access Control Lists (ACLs)**. Within Amazon S3, you can use ACLs to give read or write access on buckets or objects to groups of users. With ACLs, you can only grant other AWS accounts (not specific users) access to your Amazon S3 resources.*
- ***Bucket Policies**. Bucket policies in Amazon S3 can be used to add or deny permissions across some or all of the objects within a single bucket. Policies can be attached to users, groups, or Amazon S3 buckets, enabling centralized management of permissions. With bucket policies, you can grant users within your AWS Account or other AWS Accounts access to your S3 resources.*

Type of Access Control	*AWS Account-Level Control?*	*User-Level Control?*
IAM Policies	*No*	*Yes*
ACLs	*Yes*	*No*
Bucket Policies	*Yes*	*Yes*

You can further restrict access to specific resources based on certain conditions. For example, you can restrict access based on request time (Date Condition), whether the request was sent using SSL (Boolean Conditions), a requester's IP address (IP Address Condition), or based on the requester's client application (String Conditions).

boto: Python S3 client

Boto[12] is a Python client to AWS that is used by Pegasus (rf. `pegasus-s3`[13] and `pegasus-transfer`[14]) to interface with AWS S3.

To reproduce the following Python script (after installing boto), it is necessary to have set the following environment variables to contain the associated key strings: AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY

[12] http://docs.pythonboto.org/en/latest/, https://github.com/boto/boto
[13] https://github.com/pegasus-isi/pegasus/blob/master/bin/pegasus-s3
[14] https://github.com/pegasus-isi/pegasus/blob/master/bin/pegasus-transfer

```
>>> import boto
>>> conn = boto.connect_s3()
>>> bucket = conn.create_bucket('mybucket')
```
→ generates an error due to a non-unique bucket name:
```
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "/Library/Python/2.7/site-packages/boto-2.0-
py2.7.egg/boto/s3/connection.py", line 391, in create_bucket
     response.status, response.reason, body)
boto.exception.S3CreateError: S3CreateError: 409 Conflict
<?xml version="1.0" encoding="UTF-8"?>
<Error><Code>BucketAlreadyExists</Code><Message>The requested bucket
name is not available. The bucket namespace is shared by all users of
the system. Please select a different name and try
again.</Message><BucketName>mybucket</BucketName><RequestId>31F6685035
DA2912</RequestId><HostId>jEA7ksQ5k+woEQN7z65i6iISvS9BznPVyqF+TAqJ9fpF
f8kjAIJpObhMaEJUgAk1</HostId></Error>
>>>
>>> bucket = conn.create_bucket('rwh-bucket')
>>> from boto.s3.key import Key
>>> k=Key(bucket)
>>> k.key='foobar'
>>> k.set_contents_from_string("This is Randy's test of S3")
>>>
```

Exit the Python interpreter, re-start, and verify we can retrieve the contents of what we just put in the bucket:

```
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)]
on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto
>>> c = boto.connect_s3()
>>> b=c.create_bucket('rwh-bucket')
>>> from boto.s3.key import Key
>>> k=Key(b)
>>> k.key='foobar'
>>> k.get_contents_as_string()
"This is Randy's test of S3"
```

To copy the contents of a file and set access controls:
```
key.set_contents_from_filename('/home/fred/stuff.dat')
b.set_acl('public-read', 'foobar')C
```