CESNET Technical Report 03/2013

Homeproj – Project development support system

Jan Mach

Received 5.9.2013

Abstract

This paper describes the architecture, design and implementation of the system for project development support and all additional modules we have used or developed to integrate into the CESNET infrastructure and to streamline its administration. The support currently includes using Redmine ticket tracking system and providing hosting of multiple GIT source code repositories and APT, RPM and TAR package repositories.

Keywords

Project development support, Redmine, Git, APT, RPM, repository hosting, Charon

Introduction

Motivation

A major part of CESNET`s work consists of research and development. There are many teams in many departments working in various fields. So far, each of these teams has had their own source code repository installed somewhere and had to manage it and take care of its security and backup. They have usually used some kind of ticket tracking system different from the system used by another team. Thus, working in more than one team meant one had to learn to use more than one system. This approach was both time consuming and a waste of resources, so we came up with an idea to create a single system which would provide the teams with the ticket tracking and repository hosting services.

System requirements and system design goals can be summarized as follows:

1. Flexible ticket tracking system with following the key features:

1. Support for multiple projects, project hierarchy

2. Possibility of federated SAML[1] authentication via EduID [2]

3. Good administration interface with enabling delegating the responsibility for project management to project managers

4. Per project tools: wiki, roadmap, Git repository read access

2. Multiple Git repository hosting with good user access control

3. Multiple package (APT, RPM, TAR) repository hosting

System overview

Figure 1 below depicts the system architecture. Currently, there are five services provided by the system:

1. Redmine [3] project management and ticket tracking system
2. Git repositories [4]
3. APT package repositories
4. RPM package repositories
5. TAR package repositories

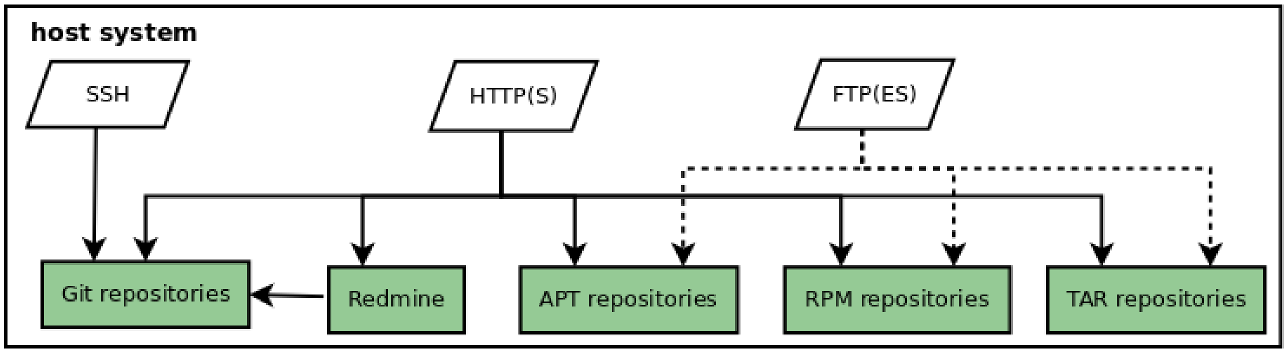

Figure 1: System overview

Those five services are accessible through three entry points. There is SSH, HTTP(S) and FTP(ES). SSH is used for authenticated read and write access to all Git repositories and it allows users to invoke restricted set of commands on the host system (see below for more information). HTTP(S) is used for read-only access to the public Git repositories and for read-only access to APT, RPM and TAR package repositories. Anonymous FTP is used for read-only access to APT, RPM and TAR package repositories. And finally, authenticated FTPES is used for write access to APT, RPM and TAR package repositories. Users upload their packages to the host system through this access point.

Design

Redmine

Redmine is a mature, stable, feature rich and easy to use project management and ticket tracking system chosen mainly for the following reasons:

1. Support for management of multiple projects, project hierarchy, public and private projects
2. Rich set of per-project tools: task management, time tracking, roadmap, forums, wiki
3. Read access to long list of VCS repositories including Git, possibility of referencing the source code in the repository from the wiki page or task description.
4. Clean user interface, powerful administration interface with possibility of delegating responsibility for project administration to project managers
5. Easy implementation of federated authentication

Installing Redmine system was pretty straightforward. The installation process for various platforms is well described in the official documentation [5]. The system deployment is depicted on Figure 2. We used Apache web server [6] for content serving. Since Redmine is written in Ruby, we used Phusion Passenger [7] as the actual application server and container. We use the MySQL database as a data persistent storage backend. Another possibility was PostgreSQL. Both are equivalent in performance and features for the Redmine's requirements and we opted for the one we are more used to.

The only tricky part during the installation was implementing the federated SAML user authentication. After some research we came across the *HTTP Authentication Plugin* [8], capable of authenticating the Redmine user using the environment variables previously set up by Apache web server authentication modules. Then it was an easy task to use Apaches`s *mod_shib2* [9] authentication module and thus shift the responsibility for user authentication to Apache. Using this approach, we could get around the need to customize the Redmine`s source code so that the future system upgrades are easy and painless. The only customization we made was done in *HTTP Authentication Plugin* and it was cosmetic - we changed the string on the login button. The authentication through the local mechanism is still possible, but this option is reserved for administrator accounts only.

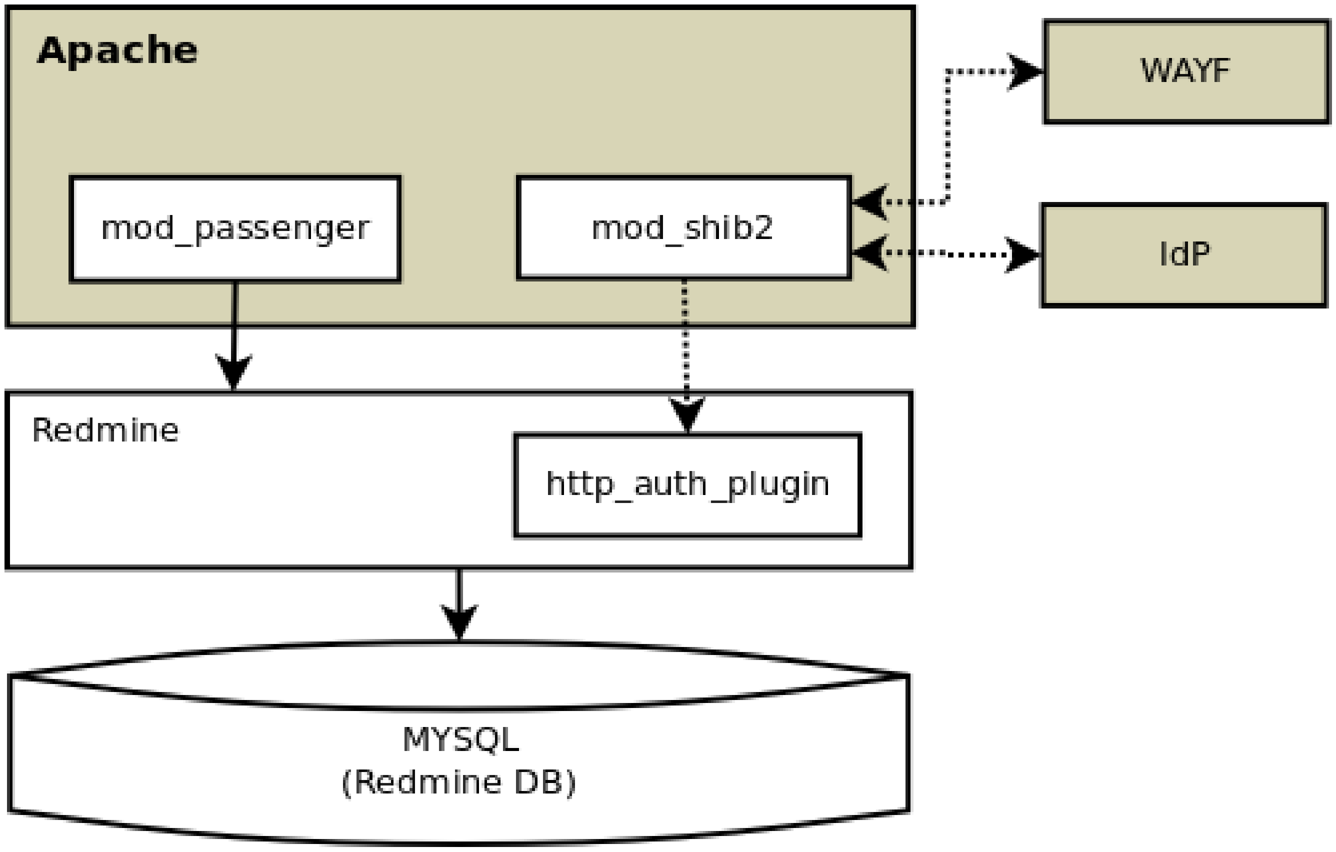

Figure 2: Redmine deployment scheme

Redmine was modified to use federated SAML authentication. However all user accounts still need to be created in the local database because Redmine needs various account metadata for general system functionality. To simplify the administration task of having to look for all the necessary information and creating the account manually, we created a very simple registration page. When accessing this page, the user is first redirected to the WAYF (Where Are You From) server and then authenticated with a selected IdP (Identity Provider). Once the authentication is complete, the database is checked and non-existent account is automatically created in the Redmine`s database. It is however created as locked and system administrators are informed about this event. After reviewing the request, the administrator can manually unlock and enable the account from the Redmine`s administration interface.

SSH publishing service

The SSH service is used for authenticated read and write access to the Git repositories and for invoking a specific set of repository management related commands. Figure 3 depicts the SSH related part of the project support system architecture. We use a standard OpenSSH server [18].

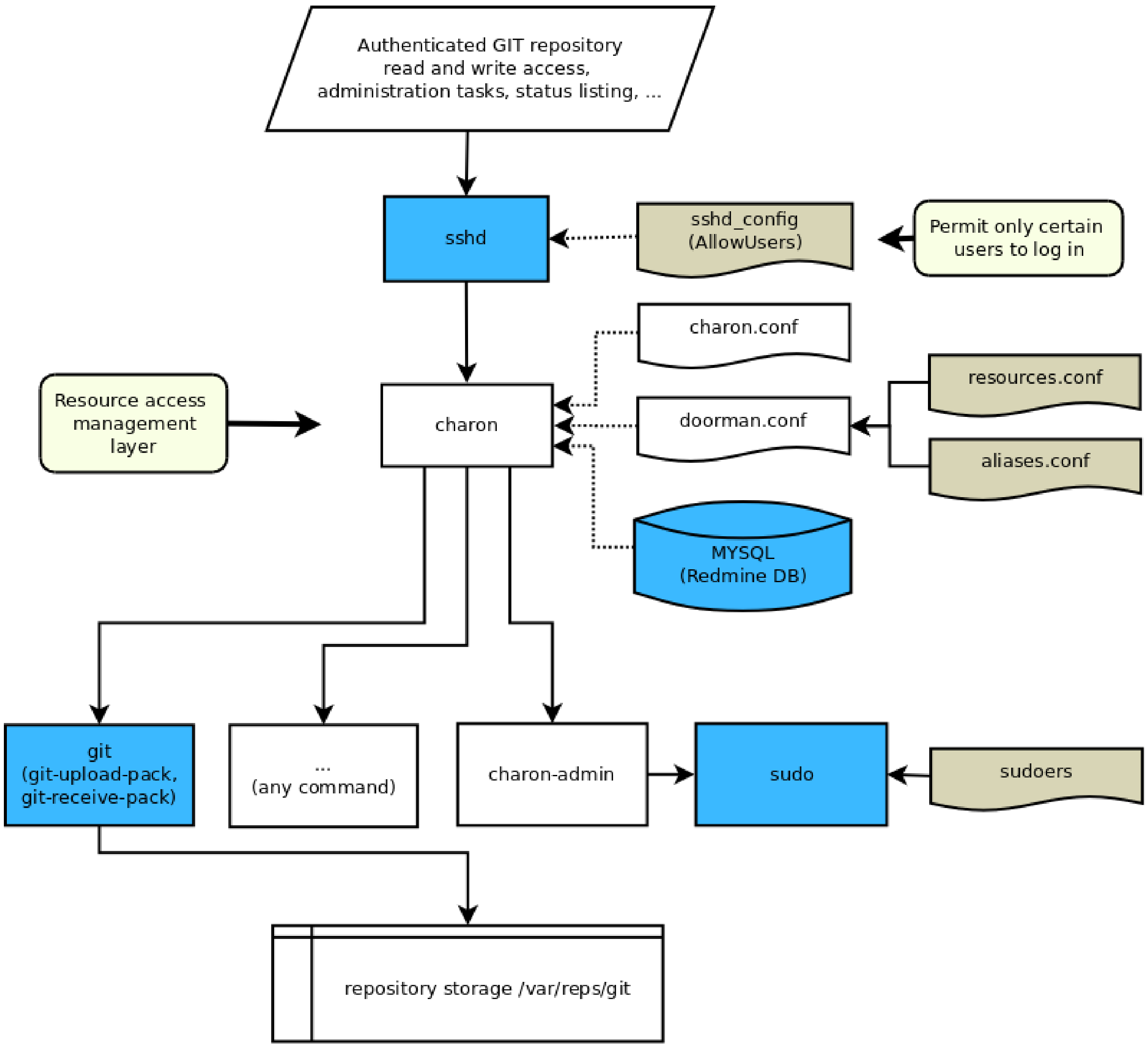

Figure 3: SSH publishing service scheme

SSH is the only (reasonable) way of securely accessing the remote Git repository for writing [10]. But SSH itself cannot restrict one user from writing into repository s/he should not have access to. The first obvious solution is using Linux file system permissions. It is an easy task to configure Linux file system permissions provided the groups of users that should be able to access two different repositories are disjoint. Once the group members begin to overlap, a configuration nightmare begins. The first solution is to use the extended file system attributes. This approach, however, does not solve the bigger issue with SSH access: users have access to the shell and can potentially harm the system.

More elegant solutions are the Gitosis[10,12] and Gitolite[10,13] projects. Those systems are configured to be executed instead of the login shell after a successful SSH authentication, deny interactive usage of SSH and restrict the set of commands user may remotely invoke on the host system. We tested the Gitosis utility for a while, but in the end we ended up developing our own solution which is not only Git specific like Gitosis or Gitolite and enables users to execute a defined set of arbitrary commands. This resource access layer is called ***charon*** and is described in detail in a separate technical report called *Charon – Resource Access Layer* [19].

But let us start from the top of the architecture scheme. Security was our big concern, so the first line of defence is using the *AllowUsers* directive in OpenSSH server`s configuration file. After a successful authentication, *charon* is executed instead of the login shell. It works exactly as Gitosis, it forbids users to run in the interactive mode and expects the command name to be passed as argument. *Charon* then looks into the ACL (Access Control List) and verifies whether the user is allowed to execute the given command. If yes, the command is executed on behalf of the user, otherwise the user`s request is denied. A part of the ACL configuration is performed by a text configuration file, but following data is also pulled directly from Redmine`s database:

1. List of projects with assigned Git repository
2. List of user accounts
3. List of groups and their members
4. Roles of users or groups within the projects

User roles from Redmine are mapped to the Charon access types and Redmine account names are mapped to the local system accounts. All ACL configurations are finally mixed together. For more information about *charon* please refer to the technical report *Charon – Resource access layer.*

To secure access to the git repositories, *charon* controls the execution of the following binaries:

1. git-upload-pack and git upload-pack (reading from the repository)
2. git-receive-pack and git receive-pack (writing to the repository)

Another advantage of using *charon* as proxy layer is that it does some translations and substitutions for the user. For example, following two lines are equivalent, currently fully functional and point to the same repository, but the latter makes use of the translation feature, is more convenient for the user and hides the actual repository location from him/her:

```
git clone [user]@[server]:/var/reps/git/test.git
git clone [user]@[server]:test
```

Besides restricting access to the Git related binaries, we also allow users to execute several other utility scripts:

```
1:     ssh [user]@[server] ssh-key
2:     ssh [user]@[server] status
3:     ssh [user]@[server] apt-update
4:     ssh [user]@[server] rpm-update
```

By using the command on line 1, the user may upload and set public SSH key to his/her account. The second command allows the user to view the list of all resources s/he has access to. The last two commands are used to regenerate the APT and RPM repository metadata after uploading new packages. For more details see appropriate chapters below.

HTTP(S) publishing service

The web service is used to access the Redmine interface, for unauthenticated read access to public Git repositories and for unauthenticated and authenticated access to APT, RPM and TAR package repositories. Figure 4 depicts the HTTP(S) related part of the system architecture. As stated before, we are using the Apache HTTP server.

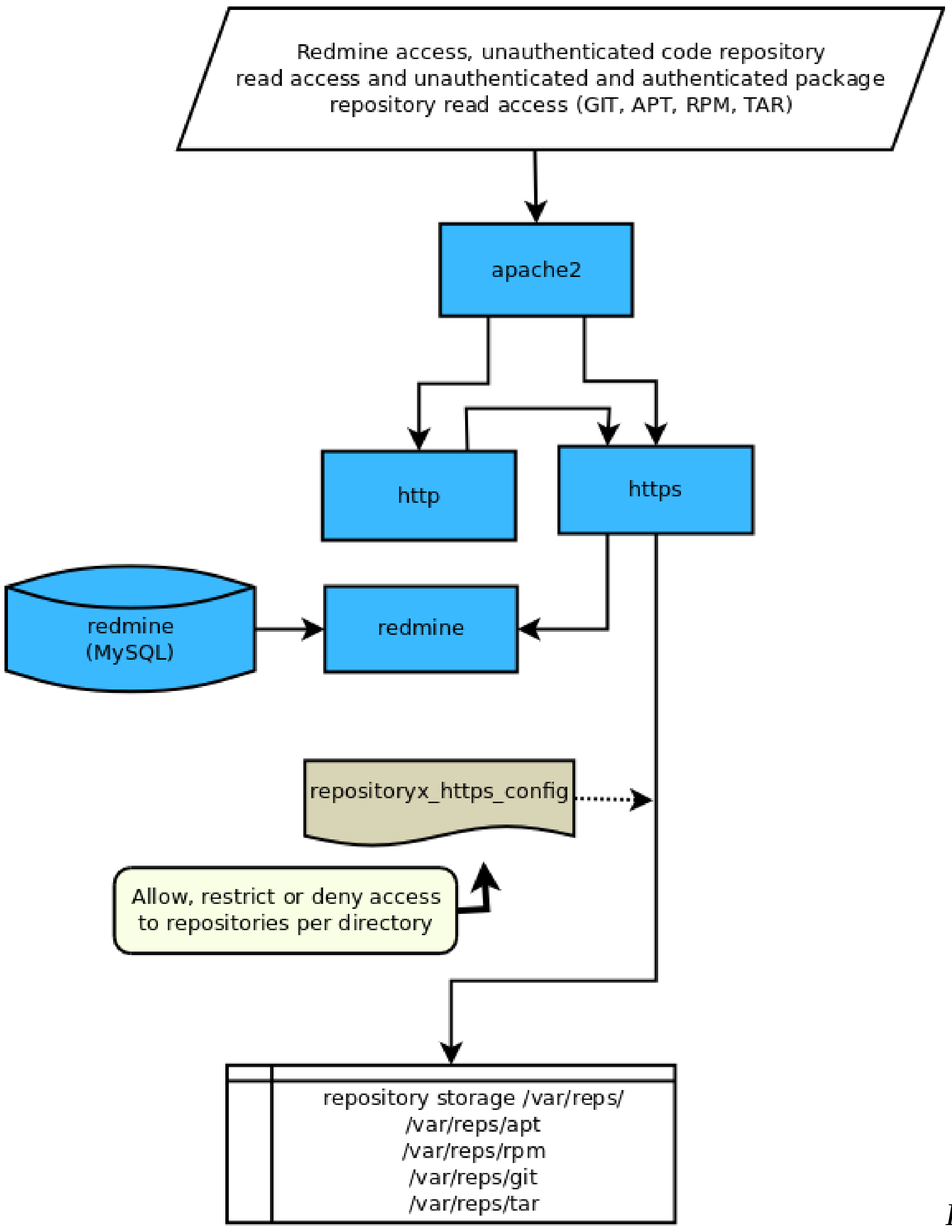
Redmine access, unauthenticated code repository read access and unauthenticated and authenticated package repository read access (GIT, APT, RPM, TAR)
apache2
http
https
redmine (MySQL)
redmine
repositoryx_https_config
Allow, restrict or deny access to repositories per directory
repository storage /var/reps/
/var/reps/apt
/var/reps/rpm
/var/reps/git
/var/reps/tar

Figure 4: HTTP(S) publishing service scheme

For security reasons, the Apache web server is configured to redirect plain HTTP to secure HTTPS. The access to private Git repositories is blocked in the Apache configuration file. To prevent the administrator from making the mistakes by manually editing the configurations, all necessary configurations are generated by the ***charon-admin*** administration utility (see the chapter Charon-admin utility below for more information).

FTP(ES) publishing service

The anonymous FTP service is used for read access to public Git repositories and to all APT, RPM and TAR package repositories. Authenticated FTPES (FTP over explicit SSL) is used for write access to all package repositories; users may upload new packages to the server through this entry point. Figure 5 depicts the FTP(ES) related part of the system architecture. We chose *vsftpd*[14] as FTP server, because we have been using it successfully in our other systems.

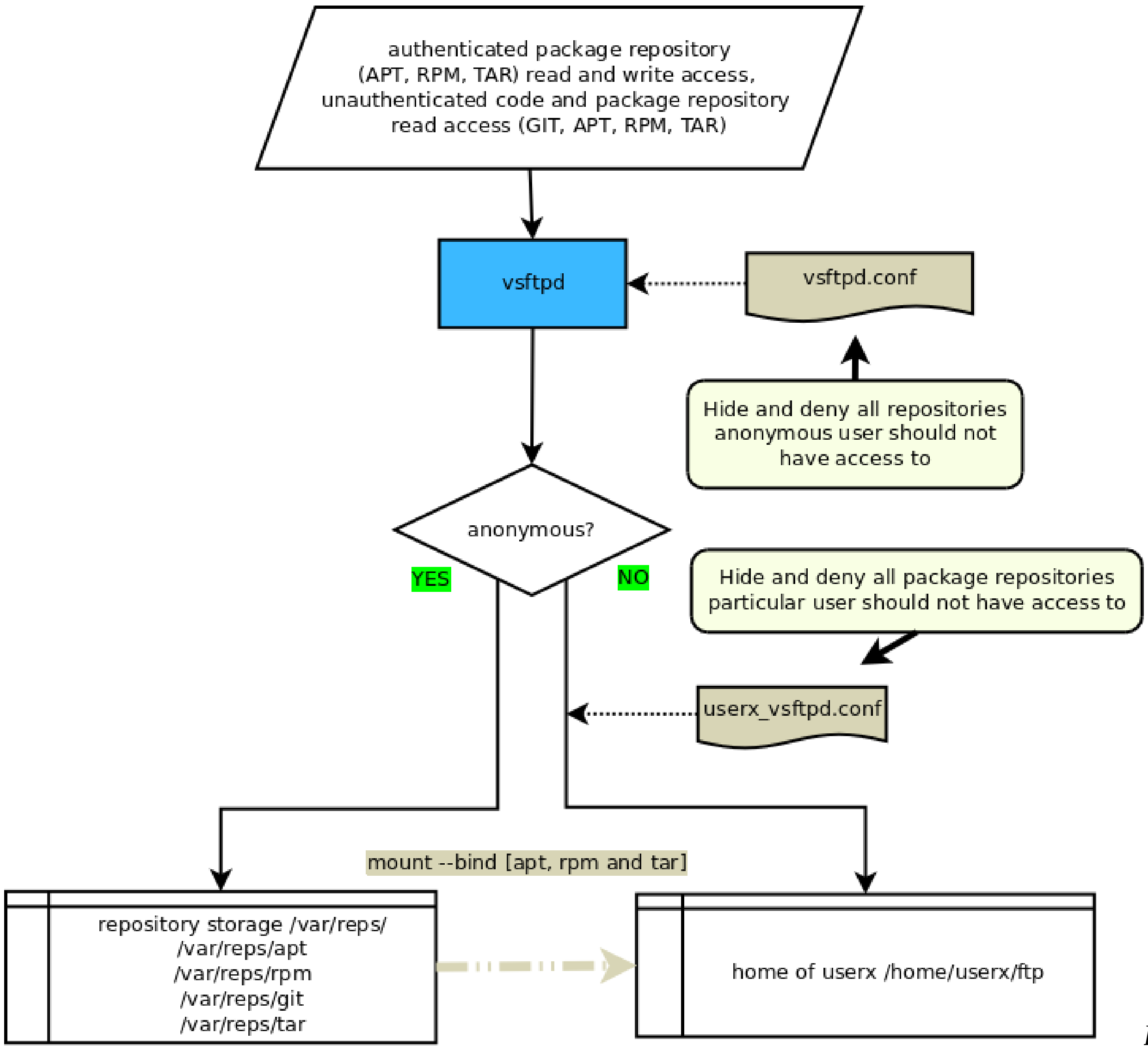
authenticated package repository (APT, RPM, TAR) read and write access, unauthenticated code and package repository read access (GIT, APT, RPM, TAR)
vsftpd
vsftpd.conf
Hide and deny all repositories anonymous user should not have access to
anonymous?
YES
NO
Hide and deny all package repositories particular user should not have access to
userx_vsftpd.conf
mount --bind [apt, rpm and tar]
repository storage /var/reps/
/var/reps/apt
/var/reps/rpm
/var/reps/git
/var/reps/tar
home of userx /home/userx/ftp

Figure 5: FTP(ES) publishing service scheme

Again, *vsftpd* configuration files for anonymous and local users are generated by the ***charon-admin*** administration utility (see the appropriate chapter below for more information). We are using the *deny_file* and *hide_file* configuration directives for restricting anonymous and local users from accessing repositories they should not have access to. This is the only way. When this article was written, *vsftpd* was not able to grant access selectively to a part of the file system tree only. Another tricky part was how to restrict directory access for local users. *Vsftpd* is capable of using per-user configuration files. Those files must reside in the */etc/vsftpd/* directory and the name of the configuration file must be the same as the name of the user account. All configurations in the user files take precedence before the configurations in the general config file. To restrict the user from wandering through the filesystem we used *local_root* directive and locked him/her in the home directory. However, this *chroot* operation forbids the user from accessing the repositories stored in the */var/reps* file system subtree and we had to find some solution. Our current solution is to use the bind feature of standard Linux *mount* utility:

```
mount --bind /some/source /some/destination
```

By invoking the above command, one can mount local folder */some/source* to another local folder */some/destination*. The ***charon-admin*** administration utility takes care of mounting and unmounting the repositories from user home directories (see the chapter Charon-admin utility below for more information).

Charon-admin utility

Charon-admin is a part of the Charon package and is described in detail in a separate technical report called *Charon – Resource Access Layer* [19]. We use this tool to consolidate all administration scripts and hide them behind a single interface. Below is the list of available *charon-admin* commands:

```
1:    charon-admin repository git create repository_id
2:    charon-admin repository apt create repository_id gpg_key
3:    charon-admin repository rpm create repository_id gpg_key
4:    charon-admin repository tar create repository_id
5:    charon-admin repository git|apt|rpm|tar repair

6:    charon-admin view allowed [resource | -] [permission_type]
7:    charon-admin view rights [account | -] [permission_type]

8:    charon-admin do apt-update|rpm-update

9:    charon-admin user create login alias Full Name
10:   charon-admin user delete login

11:   charon-admin ssh key add login

12:   charon-admin service http|ftp|ssh|all start|stop|restart
```

We created six *charon-admin* modules to facilitate the administration tasks. The first module called *repository* provides commands for repository management related tasks (lines 1-5 in the example above). So far there are commands for creating all repository types and for repairing corrupted file system permissions. This module is responsible for creating all necessary file system objects. The second module is called *view* (lines 6-7 in the example above) and using this tool administrator can check the ACL for specific user or resource. S/he can check which users have what type of access to a particular or all resources. Or s/he can check what resources are accessible to a particular user. The *do* module (line 8 in the example above) provides commands for regenerating APT and RPM repository metadata and resigning the packages with the GPG key after new packages have been uploaded successfully. This module is rarely used by the administrator, but users themselves execute this script via SSH and *charon* once they have uploaded the packages, as mentioned in chapter *SSH Publishing Service*. Next, the *user* module (lines 9-10) is used to create local user accounts. The *ssh* module (line 11) is just a shortcut tool for easy setting some user`s public SSH key. Finally, the *service* module (line 12) can be used to start, stop and restart publishing services. When the service is started or restarted, the appropriate configuration files are first regenerated using the Charon`s ACL configurations.

Git repositories

Git is a very fast and popular distributed version control system. We use it to aid developing many projects. On the project support system, we can host multiple Git repositories and define very fine access restrictions for specific repositories. There is a possibility of having either public or private repositories. Both of these types are accessible for for reading and writing to a defined set of users via SSH. Public repositories are also accessible for reading via HTTPS and anonymous FTP. Configuring all three (SSH, FTP, HTTPS) publishing services consistently is done by the *charon-admin* tool which generates the appropriate configurations based on the ACLs defined in Charon.

APT, RPM and TAR package repositories

APT repositories

APT package repositories are created and managed using the ***reprepro*** [15] tool. There is a great tutorial [16] in the Debian wiki documentation on using this tool which we followed during the implementation. But we automated and simplified the creation process by creating the *charon-admi*n module to generate the necessary file system tree objects and configuration files for us. The directory structure looks as follows:

```
/var/reps/apt/repository_id:
       -rw-r--r--    root   root          APT-GPG-KEY-repository_id
       drwxr-xr-x    root   root   conf
              -rw-r--r--  root root                     distributions
              -rw-r--r--  root root                     options
              -rw-r--r--  root root              override.squeeze
       drwxrwsr-x    charon charon db
       drwxrwsr-x    charon charon dists
       drwxrwsr-x    charon charon        incoming
              drwxrwsr-x  root charon            sid
              drwxrwsr-x  root charon            squeeze
       -rw-r--r--    root   root          repository_id.list
       -rw-r--r--    root   root          repository_id-unstable.list
       drwxrwsr-x    charon charon pool
```

Directory *conf* contains configurations for *reprepro*. This configuration describes the repository structure, its components, architectures, distribution codename and suite (stable, unstable), GPG key for signing packages etc. Users upload their packages to the appropriate subdirectory of the *incoming* directory. Once all packages have been uploaded, the user must invoke the *apt-update* command via SSH. This wakes up the processing daemon. All packages will then be read, signed and moved to appropriate places within the repository structure. The repository root folder also contains GPG key and prepared *sources.list* files for the APT package manager. From the user point of view, using the repository is the matter of simply invoking the following commands as root:

```
wget -O /etc/apt/sources.list.d/[reponame].list
https://server.domain/apt/[reponame]/[reponame].list


wget -O - https://server.domain/apt/[reponame]/APT-GPG-KEY-[reponame] | apt-key add -
```

```
aptitude update
```

RPM repositories

RPM package repositories are created and managed using the ***createrepo*** [17] tool. *Createrepo* is not as powerful a tool for RPM repositories as is *reprepro* for APT repositories. The administrator must create the whole directory structure himself and users are responsible for uploading packages to the correct places within the directory tree. *Createrepo* only creates the repository metadata files. It is not capable of signing the packages and moving them to the appropriate places. Below is an example of possible RPM repository structure. It can be customized to suit the actual needs:

```
/var/reps/rpm/repository_id:
       -rw-r--r--    root root           RPM-GPG-KEY-repository_id
       drwxr-xr-x    root root           devel
              drwxrwsr-x    charon charon i386
              drwxrwsr-x    charon charon noarch
              drwxrwsr-x    charon charon SRPMS
              drwxrwsr-x    charon charon x86_64
       drwxr-xr-x    root root           stable
              drwxrwsr-x    charon charon i386
              drwxrwsr-x    charon charon noarch
              drwxrwsr-x    charon charon SRPMS
              drwxrwsr-x    charon charon x86_64
```

After uploading packages to correct location, the user must invoke the *rpm-update* command via SSH which will resign all packages and regenerate the repository metadata.

TAR repositories

TAR package repositories are just a fancy name for a shared folder in which users may upload anything they need.

Conclusions

We developed a central project support system to remove some of the burden from our research teams. For our staff, we ensure the services of mature, feature rich and stable ticket tracking system Redmine. It allows read access to the Git repositories. Another features of our project support system are secure hosting of public and private Git repositories and hosting of APT, RPM and TAR package repositories for distribution of finished software packages. To prevent any duplicity, user access configuration is done by merging authorization data from Redmine and text files. Redmine accounts are mapped to local accounts and configurations for publishing services SSH, HTTP and FTP are automatically generated from that merged authorization data.

Currently our Redmine installation provides support for tens of projects, most of which also have Git repository created and assigned to them. There are a few projects without the Git repository and there are a few Git repositories without the Redmine support. When writing this report, there were only a handful of APT, RPM and TAR repositories active, but their numbers have been growing slowly. The whole system is used by some tens of users, a majority of our staff.

Acknowledgements

This work is supported by the research intent MSM6383917201 of the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1]: SAML, http://saml.xml.org/

[2]: EduID, https://www.eduid.cz/wiki/eduid/index

[3]: Redmine, http://www.redmine.org/, 2006 – 2012

[4]: Git, http://git-scm.com/

[5]: Redmine Documentation, http://www.redmine.org/projects/redmine/wiki/Guide

[6]: Apache HTTP Server, http://httpd.apache.org/

[7]: Phusion Passenger, https://www.phusionpassenger.com/

[8]: HTTP Authentication Plugin, https://github.com/AdamLantos/redmine_http_auth, Adam Lantos

[9]: mod_shib2, http://shibboleth.net/

[10]: Scott Chacon - Pro Git, 2009, ISBN: 978-80-904248-1-4

[11]: Extended File Attributes, http://en.wikipedia.org/wiki/Extended_file_attributes

[12]: Gitosis, http://scie.nti.st/2007/11/14/hosting-git-repositories-the-easy-and-secure-way

[13]: Gitolite, http://sitaramc.github.com/gitolite/

[14]: vsftpd, https://security.appspot.com/vsftpd.html

[15]: reprepro, http://wiki.maemo.org/Reprepro

[16]: reprepro tutorial, http://wiki.debian.org/SettingUpSignedAptRepositoryWithReprepro

[17]: createrepo, http://createrepo.baseurl.org/

[18]: OpenSSH, http://www.openssh.org/

[19]: Charon – Resource Access Layer, Jan Mach, CESNET Technical Report 02/2013