The Comprehensive Blub Archive Network: Towards Design Principals for Open Source Programming Language Repositories

Seamus Brady, seamus@corvideon.ie

28/03/2013

Abstract

Many popular open source programming languages (Perl, Ruby or Python for example) have systems for distributing packaged source code that software developers can use when working in that particular programming language. This paper will consider the design principals that should be followed if designing such an open source code repository.

1 Introduction

Imagine that there is a software company that owns an average programming language which we shall call Blub [1] . This hypothetical company is going bust and in true altruistic fashion they open source the entire Blub language.

A small but enthusiastic community grows up around Blub and they decide they would like to emulate the older, more developed open source languages. It is decided that Blub needs it's own open source language repository like CPAN (the Comprehensive Perl Archive Network). This is a centralised system that manages packaged source code (and associated metadata) that is available for download and installation onto a developers machine.[2]. The Comprehensive Blub Archive Network (CBAN) will be an online Blub code repository that all can share and use.

However some worries are expressed after availability and security issues were reported with the Ruby gems website [3] (the Ruby language analogue of CPAN - in Ruby pre-packaged code is known as a "gem" [4]). It is decided that a set of design principals for a secure and reliable open source code repository such as CBAN will be drawn up to mitigate any future problems.

This paper addresses this hypothetical challenge.

In Section 2, the reliability and security challenges facing an open source code repository are outlined. An set of solutions is identified to address these challenges. In Section 3, the six design principals arising out of these solutions are discussed. Related work is discussed in Section 4.

1.1 Resources and Methods

This problem was approached by researching existing open source code repositories for Perl, Ruby and Python and examining their strengths and weaknesses. I also looked at operating systems package managers such as Debian's apt-get.

It should be noted that while there is some criticism of existing open source code repositories in this paper, no existing system embodies all the design principals for an open source code repository. There is much to be done in all systems, even in the most developed system CPAN, especially on mirror and package signatures.

2 Background

2.1 The Challenges: Reliability and Security

If CBAN were to adopt a Prime Directive it would be "Trustworthy Software" as identified by John Chambers [5]

> "...the software provider [has] a strong responsibility to produce a result that is trustworthy, and, if possible, one that can be shown to be trustworthy..."

A brief analysis of the rubygems.org [3] hacking incident reveals two different challenges:

- When the rubygems.org system administrators noticed that the server had been hacked, the system was taken offline. This meant that developers trying to install gems (packaged Ruby code) were unable to do so. The system had no reliability built in for incidents such as this. This was a single point of failure.
- As the rubygems.org system was compromised, all gems on the system were no longer trustworthy. A full security audit had to be done before the system could go back online. The gems installed on developers machines just before the rubygems.org server went offline could no longer be considered "safe".

The full impact of compromised gems (used in some popular web development stacks such as Ruby on Rails) was outlined by Patrick McKenzie, a Ruby developer:

> One of my friends who is an actual security researcher has deleted all of his accounts on Internet services which he knows to use Ruby on Rails. That's not an insane measure. [6]

So when an open source code repository has problems, these problems tend to break down across two axis:

- **Reliability**: when an open source code repository goes offline, developers may not be able to work due to lack of code availability.
- **Security**: Compromised packaged code may make it onto multiple computers before any intrusion is even detected. Developers must be able to trust and identify secure code packages.

If we wish to fulfill CBAN's Prime Directive of "Trustworthy Software", CBAN has to be **reliable** and **secure**.

2.2 Solutions to Challenges

There are three main solutions to this challenge - mirrors, signatures and standards.

2.2.1 Mirrors

Reliability in a distributed web system can have many different facets - availability, performance, cost [7]. Let us assume that all other system operations such as backups, load balancing etc. are taken care of. What would be the most important remaining factor in avoiding downtime for developers using CBAN?

The simplest answer is built in redundancy:

> If there is a core piece of functionality for an application, ensuring that multiple copies or versions are running simultaneously can secure against the failure of a single node.
>
> Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. [7]

This is done by providing CBAN with a set of mirrors. A duplication of the packaged source code and it's associated metadata will be available across several official public servers and replicated from a master server. For instance, CPAN has it's own global set of mirror servers [8]. The improvement of the mirroring system for Ruby gems is under discussion [9].

The full system for CBAN would also involve several other auxiliary servers for code searches, source code control and other administrative functions as displayed in Figure 1.

2.2.2 Signatures

When code becomes compromised on a public code repository, an attack on a user of the repository can come from several different directions [10]:

- The system may download and install arbitrary packages containing a destructive payload.
- The system may download and install older or out of date packages with know security holes that can be exploited.

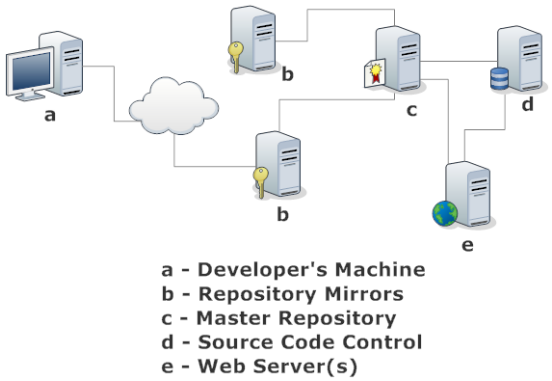

Figure 1: Repository and Associated Servers

- The system may stop the download and installation of newer packages that fix security holes.
- The system may download extra “unsafe” packages by masking them as dependencies to known “safe” packages.

These attacks can be avoided by using signing on both the code package and the metadata associated with the code. Figure 2 gives an overview of the structure of a code repository.

- At (1) is the root metadata - this is a list of compressed package files, the packages secure hashes and the root metadata signature. This signature stops tampering of the contents of each package, and if associated with a timestamp, allows a way to check if a mirror has an up-to-date and correct copy of the root metadata.
- Each code package has metadata as listed at (2) where we can see the package contents and package metadata signature. This can be matched against the embedded package metadata in the package file (3) itself to check that the package is safe.

Enforcing both these levels of signature checking will stop the attacks listed above as the data can be checked at the various stages of deployment:

- Root metadata can be used to secure mirror-to-mirror replication including an enforced timeout for updates.
- Package level signatures can be used to secure dependency checking and installation.

Again, if the rubygems.org system enforced signatures even at the packet level, no compromised code would have been installable from the system. This has been noted as an inadequacy and is currently being addressed [11].

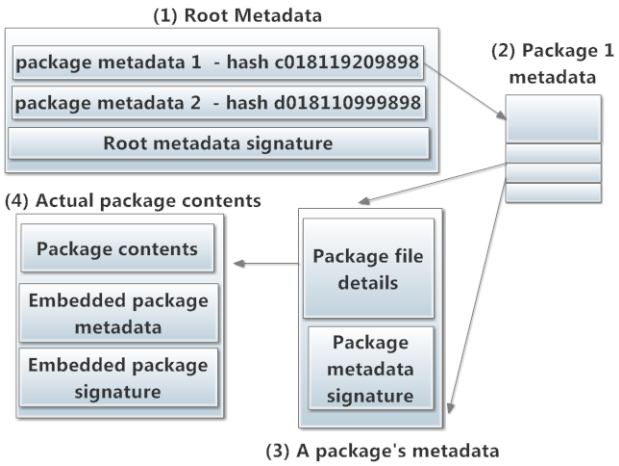

Figure 2: Repository Internal Structure (based on Cappos, Samuel, Baker and Hartman[10])

2.2.3 Standards

This is a more nebulous solution than then previous two, but important nonetheless. Standards are important as they allow predictable results. Predictable results means better tooling and automation. Better tooling and automation lead to more easily scaled systems. Standards do not need to be set in stone of course, they evolve with the language. Nevertheless, if simple standards are adopted early on, it will allow CBAN to grow and evolve gracefully.

Packaging and deployment: This need for adoption of standards can be seen in the problems and challenges facing a Python developer when packaging their code for deployment [12].

> The absence of good packaging standards has complicated the development of Python package managers. Installers have trouble finding the most recent version of packages due to the lack of a complete standard for package version numbers. Pip must create a nonstandard record of files installed for each package to support uninstallation. Setuptools had to introduce its own method of defining dependencies to work with the way packages are distributed. These nonstandard additions have helped address real problems, but have contributed to the fragmentation of the Python packaging ecosystem. [13]

If packaging and deployment standards are adopted, then the CBAN system can develop the way that the Perl CPAN system has developed. As the Test Anything Protocol has been adopted by Perl as a standard, every module

downloaded from CPAN can provide it's own test suite that is run on installation [15]. Building on such standards, Perl authors can now use such tools as Dist::Zilla to automate their CPAN deployments [16] and run tests across multiple operating systems automatically by using CPAN Testers [17].

Licensing and code ownership policy: Similar problems can happen when the licensing is not standardised across a project, holding back contributions while legal advice goes back and forth. See the long drawn out history of the Squeak programming language and the Apple license for instance [14].

Policies regarding code ownership should also be standardised. It may be useful to distinguish code ownership from code authorship in some cases. For instance, can ownership of code revert back to the community if the original author abandons it? Can the community hand the ownership over to another maintainer? These issues can be problematic in an open source community when people cannot continue their commitment to projects they started or someone else tries to take over from an existing author [18].

Also, as the signature section above outlined - knowing who wrote and signed the code package you want to install is important information.

Namespace conflicts The programming language should have some method for allowing similarly named modules to exist in the system in different packages without causing conflict.

Adopting a standard on this early in the process of building CBAN will allow greater flexibility for would-be code contributors who won't have to worry about managing module naming conflicts in a global namespace.

Other areas There are many other areas where standards can be adopted such as community policies on spam, governance and grant aid for developers. Any standards here would help CBAN develop and flourish. These are implied here, rather than listed exhaustively.

3 Results - The Six Design Principals

The six design principals below can be abstracted away from the discussion above. Each of these design principals has one or more statements that can be used as a standard for developing a system such as CBAN.

- Distributed Systems Design
  - The system should avoid a single point of failure by providing a set of public, secure mirrors.
  - Mirrors should be authenticated against the master repository using signed root metadata and an expiration timestamp.

- Package Reliability
  - Namespaces should be provided so that naming conflicts do not occur.
  - Correct metadata should be provided for dependency management.
  - A test suite should be provided by the author that can be run on deployment.
  - Reviews and reports on packages should be available on an associated web site.
- Package Security
  - Package level signing should be available so that compromised packages can be identified.
  - Changelogs and history for each package should be available for public audit.
- Source Management
  - All contributed code should be licensed on one or more standard open source licenses to avoid legal issues.
  - Library authorship should be distinct from library ownership so that abandoned projects can be managed.
  - Source code control should be put in place.
- Reliable Deployment
  - The system should provide standard packaging and deployment tools to encourage automation.
- System Manageability
  - Standard documented policies should be developed to encourage procedural openness, commitment and involvement.
  - The system should be managed by community via an open and accountable organisation - as the Zen of CPAN says...

    > Perhaps the most demanding thing is commitment: someone must keep things running. A slowly decaying and dusty archive is almost worse (and certainly more sad) than no archive at all [19].

4 Related Work

No work has been done directly on the design principals for open source language repositories. Several short histories of existing open source language repositories have been written however [12, 19].

Research work has been done on related systems such as operating system package management. Professor Justin Cappos of Polytechnic Institute of NYU has published in this area [20].

5 Summary

Six design principals for designing an open source code repository have been outlined.

These are:

- Distributed Systems Design
- Package Reliability
- Package Security
- Source Management
- Reliable Deployment
- System Manageability

References

[1] Beating the Averages. http://www.paulgraham.com/avg.htmlhttp

[2] http://www.cpan.org/misc/cpan-faq.html#What_is_CPAN

[3] RubyGems.org hacked, interrupting Heroku services and putting sites using Rails at risk. http://bit.ly/WA4o7n

[4] http://rubygems.org/pages/about

[5] Page 4, Chambers, John M. (2008). Software for Data Analysis: Programming with R. Springer. ISBN 0-387-75935-2.

[6] What the Rails security issue means for your startup. http://www.kalzumeus.com/2013/01/31/what-the-rails-security-issue-means-for-your-startup/

[7] Scalable Web Architecture and Distributed Systems. http://www.aosabook.org/en/distsys.html

[8] http://mirrors.cpan.org/

[9] https://github.com/rubygems/rubygems-mirror/wiki/Mirroring-2.0

[10] J. Cappos, J. Samuel, S. Baker and J. Hartman. A Look In the Mirror: Attacks on Package Managers, 2008. http://isis.poly.edu/~jcappos/papers/cappos_mirror_ccs_08.pdf

[11] A Practical Guide to Using Signed Ruby Gems. http://blog.meldium.com/home/2013/3/3/signed-rubygems-part

[12] Python Packaging. http://www.aosabook.org/en/packaging.html

[13] The Future of Python Packaging. https://us.pycon.org/2012/schedule/presentation/498/

[14] http://squeak.org/SqueakLicense/?version=4

[15] http://search.cpan.org/~petdance/Test-Harness-2.64/lib/Test/Harness/TAP.pod

[16] Dist::Zilla.http://dzil.org/

[17] http://static.cpantesters.org/page/about.html

[18] http://help.rubygems.org/discussions/questions/55-someone-has-acquired-ownership-of-a-gem-without-my-permission

[19] http://www.cpan.org/misc/ZCAN.html

[20] http://isis.poly.edu/~jcappos/publications.html