Rubygems Trust Model

"As soon as you trust yourself, you will know how to live" -
Johann Wolfgang von Goethe[1]

Introduction

This document outlines design goals and key considerations for a trust model for Rubygems. It will also include reference material and signposts to proposals up until it becomes unwieldy in size. The content of this document is not fixed.

Short (and public) link to this document: http://goo.gl/ybFIO

Those readers unfamiliar with Ruby may wish to read the Definitions section first, to aid in readability of the rest of the document.

Schedule

The current schedule is a proposal, and is based on a few assumptions.

- 2013/02/09 Goals finalized (primary content in this document)
- 2013/02/23 Working proposal selected and published
- 2013/03/09 Proposal finalized (implementations begin)

This schedule proposal is aggressive. We will likely want to break implementations and even proposals into phases. Critical goals to be achieved first, and later goals to be achieved using other or complementary means. The "finalized proposal" need not be complete, but should be agreed in principle and validated with relevant parties / reviewers as not having significant deficiencies.

[1] "As soon as you trust yourself, you will know how to live."
<http://www.quotationspage.com/quote/2919.html>

Contents

Goals

Critical

- Reduce the time to known good during incident response significantly
- Reduce the complexity of incident response significantly
- Provide clear and well documented guidance for all incidents in the Threat Model
- Provide a template for unknown threat response
- Provide documentation and recommendations for users about the Trust Model
- Verifiable content[2] even after transport through hostile networks / Servers

High Importance

- Provide user enforceable levels of trust in Servers
- Provide user enforceable levels of trust in Gems
- Clearly defined processes for Gem removal
- Clear definition of abuse
- Clear process and scope for abuse response
- Clear process for transfer of publishing rights for a Gem namespace
- Introduce as few new vectors in the Threat Model as possible
- Support transfer of projects, addition and removal of developers
- Handle orphaned projects or projects that have minimal activity

Important

- Systematic trust verification can be performed by offline tools
- Tools for any of the above requirements can be integrated and are easy
- Servers can be managed by non-expert community volunteers
- Gems (future and existing) are continually usable (e.g. when Servers are not)
- A mechanism for assigning and consuming trust in specific Gems
- Implementable without new, significant, external dependencies for clients

Suggested

- Use rather than create solutions
- Levy mature, well understood cryptographic solutions
- Gem author verification mechanisms
- Specific Server verification mechanisms (including mirrors)
- Do not break rsync/http cache semantics

---

[2] The content may not be safe, but it must be able to be validated as being authentic

- Prefer faster global recovery over vector reduction or complexity
- Compatible with non-ruby toolchains
- Do not depend on dynamic serialization protocols
- Resilient to bit-rot and service expiry

Non-Goals

- Create a mirroring system for Servers
- Create a Gem content (code) validation process/service/solution
- Centralize Ruby ecosystem identity management
- Authentication of Gem uploads to a Server (connection/protocol wise)
- Redefinition of minimum client requirements
- Solve Rubygems and native package manager integration
- Solve CAP
- Solve Zooko's Triangle

Additional side goals / high level requirements

- An implementation must take a reasonable amount of time
- An implementation must be possible within all current Ruby interpreters
- An implementation must not lead to additional legal responsibilities for supporting parties
- The implementation must be backward compatible
- The implementation must be acceptable to the current Rubygems team

Assumptions

- The critical goals are possible
- Incident response capabilities are more important than incident prevention
- User education helps to mitigate user risk
- A sufficiently easy to use system is possible, and will be used
- Gems may be in use forever
- Servers may not be available as long as Gems
- The system will be flawed

Customers

rubygems.org:

As a distribution service provider rubygems.org has a set of responsibilities to it's users:

- To authenticate publishers such that namespaces are protected
- To maintain security levels over publishable namespaces
- To provide a certain minimum service level

"official" mirrors:

Mirrors of the main rubygems.org repository have these responsibilities:

- Validate the authenticity of the upstream mirror (avoid mitm)
- To maintain security levels over mirrored data (avoid injection of new data not from the master site)
- To provide a certain minimum service level

gems.github.com:

Essentially the same as rubygems.org. The service has been archived. As such it will remain entirely static. Addition of new features to aid in this trust model is highly unlikely. Ongoing use, if the outcome of this work adds features for enhancement of trust, will be vehemently discouraged.

Internal servers (corporate distributions):

There are several "on the market", geminabox, launching rubygems.org, using `gem index` after simply dropping gems in a directory. For the sake of backward and forward compatibility and specification coherence, it may be a good idea to maintain gem index in a state that is in tune with the rubygems.org trust model. This may not include all of the server side portions necessary for some form of identity or cryptographic enforcement. Internal servers have many different sets of requirements, some may not even require authentication due to being hidden inside of a trusted walled garden.

Gem publishers:

Publishers in general are unaware of trust model concerns, but they do have some general expectations:

- Publishing gems should be easy
- Publishing gems should be fast (available on servers soon after publishing)
- Publishing new gems should not require undue process

- Publishing in a "responsible" (read: secure) way should also be easy
- rubygems.org should provide most of the security they need in order to satisfy their users

Rubygems has had a basic signing model for a long time through `gem cert`, and the gem signing features. To date this feature set is rarely used, in general only used by publishers that use a toolchain that implies usage. It is evident that this user class has a strong desire for the signing and/or security processes to be largely automatic and mostly hidden from view. There may be little objection to an automated solution with these users.

Gem consumers:

Consumers are in general even less aware of trust model concerns than publishers, and as such they have several critical assumptions:

- rubygems.org has an authorization solution, therefore it is secure
- Gem authors are generally trustworthy (MINSWAN)
- They would hear about any contradictions of the above

At large, gem consumers have the following derivative desires:

- rubygems.org authorization is generally trustworthy (identifies authors)
- Rubygems will warn them in the event of trust problems
- There is some significant level of protection against hostile Servers
- There is some level of protection against hostile Gems
- It is easy to consume trust in the above, without additional complex steps
- Trust is largely someone elses problem

Gem consumers that do audits:

Some gem consumers are more rigorous in their consumption, and more correct in their understanding of the currently provided trust model. They are presently aware of the following:

- New gem authors are able to publish without oversight
- Servers can be hostile
- Gems can be hostile

In general, these consumers audit the first version of a new gem they consume. A much smaller subset of this user group audit each individual version, or the changes that occurred between versions. These consumers have an increased desire for:

- Increased trust in end to end safety - a Gem can be safely transported through hostile Servers
- Increased trust in Gem files, particularly for repeat publishings (new verisons). They wish

to be able to trust specific authors to reduce the audit load (e.g. using author signatures).
- Author authorizations should be associated with namespace authorizations on rubygems.org, and those authorizations should be in some way verifiable.
- rubygems.org is sustained in a state that is generally trustworthy.
- Interacting with rubygems.org through Rubygems is safe, even when rubygems.org has become hostile.

Requirements Summary

- Publishing gems must remain easy (gem build & gem push)
- Publishing gems must remain fast (<10 minutes)
- Requesting authorization to publish in new gem names must be fast (time?)
- Cryptographic additions must rely on OpenSSL
- Servers should be able to verify Gems, based on author and authorization data
- Servers should keep records/backups of Gems and/or Gem checksums
- Servers should be able to revoke published authorizations and credentials
- Servers should be able to remove Gems (index entries and files)
- Gems should not be modified after they have been published
- Rubygems can verify the authenticity of a Gem based on some known credentials
- Rubygems can verify the authenticity of an author given author credentials
- Rubygems can maintain an up to date revocation of credentials
- Rubygems performs regular automated checks for revoked credentials and Gems
- rubygems.org and Rubygems communication has some MITM safety by default
- Verification and/or cryptographic features are not exclusive to rubygems.org
- The enhanced trust model is the default in new versions of Rubygems

Discussion Notes

The largest issue from the recent incident was the lack of a ledger of checksums and/or the lack of .gem file backups. Additionally the authorization database should have had offsite and offline (or at least inaccessible) backups available for restoration. It is luck that prevented these issues from begin critical. These issues are very quickly and easily solvable, and go a long way toward solving the first two goals.

The current gem signing system has two frequently discussed flaws. Publisher private keys are not protected by a one time pad, password or other form of local storage encryption. Publisher public certs have no currently well defined or easy process for distribution. This reduces the trust and use in the system significantly. The current system (and incidentally, most proposals) provide no assurance for the content of Gems, other than the content has been signed by a particular key. As a result, the presence of a trusted signature is only a layer of confidence, not any assurance of the safety of use of the Gem contents.

User education is critical. The security level of rubygems.org and the Rubygems ecosystem has in no way been miscommunicated in the past. Users, even expert users, have simply not been appropriately considering the current trust model. The assumption that namespace authorization and server perimeter security is enough, is now not enough for some users, just due to a change in awareness. While this position seems somewhat reactionary, it is also entirely valid, as it is an alignment with their considered stance against the current trust model.

Rubygems and rubygems.org are not the only systems to have trust models vulnerable to these kinds of vectors, and to have these kinds of requirements. While similar systems for other language package systems may have some differing properties, they also share most. Many language oriented (as opposed to distribution oriented) package systems also provide namespace authorizations to arbitrary requesting individuals, and ship packages that may or may not be appropriately signed by authors. Some of these systems use SSL appropriately, but many do not. The SSL handling in Rubygems was found to be better than some, although unfortunately currently not used in the default configuration. This should change as soon as possible. Contrary to distribution packaging processes, there is no central authority or proposition of responsibility for package publishers. Package publishers have little to no relationship with rubygems.org other than credentials that authorize them to publish Gems of particular names. This lack of assigned responsibility and lack of ability to audit alters both the threat model, and the incident response processes. It has the greatest effect on users, who assume that the model is similar to that of OS package distribution.

It is generally believed that some form of package signing can be of significant assistance in enhancing the current trust model. Many other package distribution systems use PGP, and this

has lead to many discussions about it's use. Currently Ruby does not ship with a PGP library, and it has been stated by several maintainers that a PGP solution is unlikely to satisfy our requirements as a result - at least for native inclusion. Many alternatives have been presented utilizing X.509 based PKI, that is available through the already available dependency on OpenSSL. Many proposals include the use of a CA type notion, a central countersigning authority that provides features of central authority and potentially public key distribution, as well as support in a revocation model. None of the existing proposals to date have a complete coverage of a revocation model that is resistant to significant abuse in the event of a CA and/or rubygems.org breach. Some have proposed that such resistance may be impossible to practically implement, and that a simple double-signature solution (signed by author and signed by distributor (rubygems.org)) is simple and good enough.

Additional topics have arisen regarding providing additional signing features that provide for further enhanced trust models. Specifically the concept of a cryptographic user vouch. The proposal is that such a system could provide the ability for non-author users to upload cryptographic signatures of Gems to be optionally made available to other gem users. A typical use case for such a model would be a vendor that does code or security audits of publicly released Gems, providing a publicly usable record of that audit. As such a user base grows, this may provide non-auditing end users with the ability to benefit from this information, such as to "only install gems that have passed an audit by vendors X and Y". Two such potential vendors discussed today include Redhat and Google, with many others potentially forthcoming. While this discussion item is largely out of scope for the core trust model, it may be sufficiently important that it is to be later included, and may want to be included in as much as not being excluded by some future system designs or proposals.

It is important to maintain mirror replication semantics (HTTP cache and rsync), as such embedded signatures in valid Gems should not be required to change as a result of revocations, expiration or other invalidations of publisher credentials. This poses some significant challenges to a revocation model, particularly as the model cannot be tied directly to versions or connections (unlike SSL and some other systems). No direct solution to this problem has yet been proposed. Discussions recommended two options that can help with this problem: a publicly distributed ledger, and double signing (author and server signatures). Double signing has some complexities in the event that a Gem is to be published through multiple channels (e.g. through rubygems.org and also some competing service). It is currently suggested that such a workflow would involve shipping a gem through rubygems.org first, fetching the double-signed copy, and republishing that to alternative services (which may or may not add additional signatures). It is recommended that any such model again conform to the "do not modify published Gem files" strategy, in order to provide for inter-system verifications based on raw file data. This is not seen as essential, but merely important for consideration.

It is expected that a high degree of publisher credentials will be lost, and therefore be reissued or recreated. In these events, the system should be resistant to end user frustration, while still providing best-effort trust models. This potentially adds credence to some central signing or

double-signing models, as the central signatures can be used for first-level or common use verification. More secure environments would need to go through additional verification steps in order to assert true end-to-end security in these cases. It is possible that rubygems.org could keep a record of author public keys, through the existing authorization model, although the potential futility of such a centralized model in the event of a breach is recognized by many. Again some form of replicated public ledger may provide additional layers of verification options in this case, however, it has also been discussed that ledgers and CRLs both have some eventual scaling concerns.

Current authorization mechanisms exist within rubygems.org to control publishing rights to publish gems with particular names (sometimes referred to as gem namespaces elsewhere). This system could be migrated to utilize the cryptographic signatures for authorization. Doing so would significantly enhance uptake among publishers. There are also arguments that rubygems.org should not be directly involved in any signing processes, however these processes may be able to be separated by dead-drop systems or other means. In any case, splitting the threats to rubygems.org as a first point of publishing and distribution, from some system that provides any cryptographic verifications, signatures or validations has been strongly recommended.

Threat Model Items

Attacks and Threats

- Man in the middle
- Denial of service
- Trust poisoning
- Implementation poisoning
- Dependency attacks
- Coincidental attacks / corruptions
- Social engineering
- Trojan horse
- Time
- Disclosure
- Embargo
- Theft
- Entropy
- Destruction
- False Alarms

- Bugs
- Misuse
- Law
- Culture
- Malicious gem author(s)/team(s)

Assets

- Primary .gem servers
- Mirror .gem servers
- Primary upload servers
- User computer
- Intermediate networks
- Rubygems team(s)
- Users/Distributors/Vendors
- Rubygems code
- rubygems.org code
- Threat model designers
- Existing gems
- Supported platforms

History

Rubygems was recently hit by a Proof of Concept attack that highlighted deficiencies in various areas of both the social and technical aspects of the system. The event also highlighted significant deficiencies in user understanding of the current trust and threat models. While the attack had a very limited real damage, the post-attack due diligence took a significant amount of time. It has been realized that there are a number of mechanisms to significantly improve in these areas. Since the incident, it has also been observed that selection of policy and technology to form an improved trust model is non-trivial, and requires some organization.

Details of the recent incident are available at the following locations:

- http://blog.rubygems.org/2013/01/31/data-verification.html
- https://github.com/rubygems/rubygems-verification
- https://docs.google.com/document/d/10tuM51VKRcSHJtUZotraMlrMHWK1uXs8qQ6Hmguyf1g

Some key points of interest from the event are:

- Mirror data being available for use was not by design
- No backups were available for .gem files
- Existing signature mechanisms were useless due to the lack of public key distribution
- External (non-ruby) tools were critical to the early phases of first response

- The ability to compare data sets seems important for worst-case scenarios
- Phased incident response to build risk profiles worked very well
- Out-of-band / non-normal communication channels are important (blog down, etc)

Definitions

- Gem / Gems refers to “ruby packages in the .gem format”. (this is not yet well specified, assume “current implementation” and detail extensions)
- Servers refers to any service or server providing infrastructure relating to distribution, enforcement and/or trust of Gems.
- Rubygems refers to the command line gem management software, as well as the runtime library that manages the ruby load path.
- rubygems.org refers to the primary central distribution Servers, that are also the central authority for the Gem name namespace.

Potential Sources

- http://blog.infobytesec.com/2010/10/evilgrade-20-update-explotation.html
- https://www.updateframework.com/browser/specs/tuf-spec.txt
- http://isis.poly.edu/~jcappos/papers/samuel_tuf_ccs_2010.pdf
- https://github.com/rubygems-trust/rubygems.org/issues
- https://metacpan.org/module/CPAN#Cryptographically-signed-modules
- http://docs.python.org/2/distutils/uploading.html
- http://wiki.debian.org/SecureApt
- http://www.rpm.org/max-rpm/ch-rpm-pgp.html
- Amoroso, Edward et al. "Toward an approach to measuring software trust." *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on* 20 May. 1991: 198-218.
- Gallivan, Michael J. "Striking a balance between trust and control in a virtual organization: a content analysis of open source software case studies." *Information Systems Journal* 11.4 (2008): 277-304.
- Ali Babar, Muhammad, June M Verner, and Phong Thanh Nguyen. "Establishing and maintaining trust in software outsourcing relationships: An empirical investigation." *Journal of Systems and Software* 80.9 (2007): 1438-1449.
- Carmel, Erran. *Global software teams: collaborating across borders and time zones*. Prentice Hall PTR, 1999.
- Stewart, Katherine J, and Sanjay Gosain. "The impact of ideology on effectiveness in open source software development teams." *Mis Quarterly* (2006): 291-314.

TODOs

- Actual Threat Model(s)
- Document structure

- Document status
- Introduction
- Schedule
- Proposal(s)
- Future Paths