KeyMan: Trust Networks for Software Distribution

Ben Laurie <ben@algroup.co.uk>
Matthew Byng-Maddick <mbm@aldigital.co.uk>

February 28, 2002

1 The Problem

Software, particularly open source software, is vulnerable to attack by distribution of maliciously altered versions of it.

The standard solution to this attack is to either distribute checksums (traditionally MD5 but SHA-1 would be better these days) or to PGP sign the distribution or its checksum.

Clearly distributing checksums is a very weak solution, particularly since these are usually obtained by downloading them from the same site as the software itself, which makes the checksum as vulnerable as the software, and hence of no value.

PGP signing is a better solution, but neither of the current ways of associating the signing key with the project are particularly strong.

One is to use identifiers with the appropriate email domain, or some kind of role email address. This is not a particularly strong way of connecting the package signer, because you can never be sure of the key signers' own authority to mark a key as authorised to sign the particular software. This is especially true where one authority has many projects that it manages.

Another common scheme is to include a list of valid signing keys in the software distribution - this does have some merit for validating future distributions from the same source, but clearly is of no value whatsoever in validating the first one someone downloads.

In both cases, the sheer difficulty of actually checking PGP signatures, which is a tedious manual process, is a major barrier to any rigour in this scheme.

2 Why is it hard to validate the signer?

As one of the authors is a director of the Apache Software Foundation (ASF), it will be used as an example throughout this paper.

In the ASF, there are around 50 different projects. Each one has a number of contributors, and typically any of those contributors can build any particular release of the software. Clearly, only the builder of the release is in a position to sign the software. This means that there are literally hundreds of people who can sign software that is distributed by the ASF.

But how does one check that they are entitled to? This is really quite troublesome. You might check that their key has been signed by some well-known member of the ASF, perhaps. But what does that actually mean? It means that well known person has validated their identity and does not imply that they can necessarily sign a release of Apache!

A solution that has been suggested to solve this problem is to have an ASF-wide signing key. Although this clearly helps, it just changes the problem from one of validation to one of management. Who has control of the key? If it is too many people, then the key is both reduced in value, because we have the original problem - there are people empowered to sign things they should not be - and also a new one - the key will presumably have to be revoked and replaced

on a regular basis, as the pool of eligible signers changes, if too few, then the problem is that the signer may no longer be the person who made the release.

It was thinking about these problems that led to the idea of KeyMan.

3 What is KeyMan?

KeyMan is a piece of software that permits the management of keys, certificates and signatures in a distributed and exportable network of trust.

The idea is relatively simple. First, there's the concept of domains. Every object can have its trust evaluated in a domain, the domain being an indication of who controls it. Sub-domains are considered to be contained within their super-domains, so, for example, the domain "apache" might be the domain for the whole ASF, and the domain "httpd.apache" the domain for the Apache Webserver.

The next idea is a trust level - this is simply a number between 0 and 1 inclusive, where 0 means "no trust" and 1 means "complete trust". Numbers in between do not have any strict meaning, for example .5 doesn't mean "I half trust this" - they are just meant to be an indication of degree of trust, providing at least some rough idea of how much trust should be placed in a thing. Their interpretation will probably be a matter of personal preference (the most obvious being that only things with a trust of 1 should be trusted).

The only other thing KeyMan uses is a trust depth. This is used when making a certificate of a key - one signs the key with a domain (or domains), a trust metric, indicating one's personal trust in that key in that domain and a depth, indicating how far the trust extends in signatures or certificates made by the key being signed. It acts rather like a Time-To-Live metric, in that things can reduce it by arbitrary amounts, but must always decrement it.

KeyMan objects are implemented as XML data structures, parts of which are signed, and parts of which are not. These will be described in a future paper. Being XML, they can easily be integrated into a larger XML export document, so multiple items (either objects or certificates) can appear as one document.

3.1 Why use a non-1 trust metric?

The principle reason we have thought of for using a metric that isn't 0 or 1 (clearly 0 is the same as not signing at all, of course) is to indicate that you have some confidence in a key, perhaps through circumstantial evidence, but not complete confidence.

How might you get circumstantial evidence about a key? One way is through repeatedly seeing it over a period of time signing software you use. Another is to see a certificate or set of certificates from keys you trust in other domains or even in the same domain but with inadequate depth. Of course, should this happen, it might be better to re-evaluate your trust in those certificates (including domains and depths), rather than signing the key itself with partial trust.

3.2 Why use a non-infinite trust depth?

The most obvious reason is because you want to give someone the ability to certify objects in a domain without giving them the power to delegate that ability. In this case, a depth of 1 is appropriate.

Another case is where an organisation wants to limit complexity in signing and certification, by only granting limited power to delegate - for example, the ASF board might sign the keys of the management committee for one of its projects with depth 2, allowing only people directly signed by that committee to do releases within that domain.

An infinite depth is appropriate where a key is trusted completely in a domain. For example, the ASF board members would probably sign each others keys in the Apache domain with infinite depth.

3.3 Evaluating Trust

In each installation of KeyMan, there exists a "root key", which is a key used by the owner of the installation. Any non-zero trust in any object will be traceable by a chain of signatures and certificates back to this root key. There are other models for doing this, but we feel that this is the most appropriate.

So, to evaluate the trust in an object, we must first know the domain of that object. We then find all possible paths of signatures or certificates, back from that object to the root key, where every certificate is in the domain of the object or a super-domain of it, and the depth on each certificate is sufficient to reach the object. Then all the trust metrics on each path are multiplied together, giving the path trust for that path. The highest path trust is then the trust in the object.

The existence of multiple paths at reasonable trust levels can be an example of the circumstantial evidence mentioned above, which might make the user more likely to make their own certification of a given object.

4 KeyMan: the software

So now we know how KeyMan works, what does the software do? It manages all of the above objects, and their associated signatures and certificates, calculates the trust in supplied domains, allows you to sign and export objects, and also to download KeyMan objects from URLs and check them in one operation (note that the object can be a software distribution plus associated KeyMan signatures).

It displays the chains of signatures and certificates leading to trust on an object graphically, with helpful information available for all of the objects involved.

It allows you to sign things, and to export your certificates, signatures and objects themselves in a format suitable for other KeyMan users to import.

And it does this all both graphically and from the command line.

Incidentally, all signatures are currently actually PGP signatures, using GnuPG to manage them, but there is no reason they should not be any kind of public key signature. The architecture has been designed to allow for this. It is also possible to import pre-existing PGP signatures to be checked as part of the certification chain. Because these are signatures rather than certificates, they default to a trust metric of 1, an infinite depth, and a domain of ".", the root-level domain.

5 Usage Scenarios

How KeyMan would be used in practice is probably best illustrated by considering four different types of user, who conveniently span more or less all possible uses of KeyMan. Even more conveniently, each type of user builds on the previous one, each using the software in a slightly different way.

These four types of user are software developers (assumed to be members of a large team), sophisticated end users, package maintainers and naive end users.

5.1 Software Developers

Software developers really have two things they have to do with KeyMan - the first is to sign each others keys. This is normally best managed by first signing in the usual PGP web-of-trust way, indicating their trust in the key being owned by its claimed owner, then signing in KeyMan with appropriate domain (i.e a sub-domain of the overall project), trust metric (1, usually) and depth (depending on project's policies) in order to include them as a member of a project.

In the case of the ASF, the way this would probably work is that the ASF board would all sign each other's keys in the "apache" domain with infinite depth, and get as many of the other ASF members and contributors as possible to sign their keys in the same way (and, indeed, anyone else that can be persuaded). The board members would then sign the keys of each project management committee (PMC) in that PMC's domain (which would be a sub-domain of "apache", for example, "httpd.apache"). The PMC would, in turn sign keys of the project maintainers in appropriate domains (which might be sub-domains of the PMC domain, or the PMC domain itself). These maintainers can then sign project releases.

The net effect of all this activity is that anyone who can find a path of trust to any ASF developer should end up being able to trust all software released by the ASF (via developer → Board → PMC → maintainers → developer), but in a way that is scalable and maintainable.

The other thing a developer has to do is release software. This is really just a matter of signing the tarball, setting its domain and setting a URL where it (and future versions and the certificates) can be downloaded, and then putting the resulting KeyMan exports at that location.

5.2 Sophisticated End Users

A sophisticated end user will want to make their own decisions, control their own life as far as trust is concerned (see Naïve End Users). This means that they will try to find chains of trust that lead to the software they need to check, through friends or colleagues. It is typically this kind of user that will end up signing keys with partial trust, being unable to find routes that give them complete trust.

Of course, they can use KeyMan as a tool to help them explore the trust network and see if there are keys they may be able to validate to improve their position.

And if they have friends or colleagues who have trust that improves their situation (assuming they trust them, of course), then they can get them to export their trust graphs and send them (by email for example).

5.3 Package Maintainers

A package maintainer is in much the same position as a sophisticated end user, from the point of view of checking signatures and certificates, except that they may have better contacts through their organisation to enable trust paths to be found.

Once they've verified the source, most package maintainers will then apply patches and other tweaks appropriate to their way of doing things, and then will probably produce some kind of rolled-up version of the modified/configured software. They will then sign this with a key that has been certified in the domain of their organisation. The idea behind this being, of course, that the distribution can have a single root key which, if trusted, will automatically certify all software in that distribution.

Naturally, the distribution will have hierarchy of domains similar to the ones maintained by software authors - reflecting their own internal network of packagers and their managers.

5.4 Naïve End Users

The typical naive user will most likely be using vendor-supplied packages on their system, and will be compiling very little from the original source. They would set themselves up to trust the vendor root key, to a depth such that they would trust the package maintainers of the vendor's packages (probably, in practice, infinite depth, since they should trust the vendor to manage their keys and certificates correctly anyway).

The vendor's key could well be distributed with the installation media for their distribution - including KeyMan, of course. KeyMan would then be used as part of their installation and upgrade process, and would be largely transparent to the user. The only thing they'd have to do in a normal situation is to generate their own root key and sign the vendor's root key with it - of course, the distribution would probably largely automate this process as part of the install.

6 Future work

One of the things we are still working on in KeyMan is certificate revocation. In essence this is simple, but there are a number of issues.

The main issue is that we would like to not invalidate all past certificates made by a key that has been revoked at a particular time. This requires us to have timestamps, signed by third parties[1]. In essence, timestamps are relatively simple - all one needs to do is send an object to someone, containing a hash of the thing to be timestamped and a claimed time[2], which the timestamper then signs if they believe the claimed time, including a time at which they signed[3].

It is clear that in the worst case we can abandon the idea of timestamps and simply rely on the time the object actually arrived at the relying party (that is, if there is a revocation that claims a key was revoked at time t_X, and we first saw an object signed by that key at time t_Y, then if $t_Y < t_X$, we can still believe the signature). Obviously this gives an attacker a denial of service attack, but it does prevent them from forging valid signatures.

Another thing we'd like is to allow anyone to revoke anyone else's certificates - this makes sense with the model of the network of trust - whether you believe the revocation depends on whether you trust its author, rather than the simplistic model where only the holder of the key can revoke it.

In any case, there are a number of complications resulting from revocation and timestamping which we are still working on, and which, no doubt, will be the subject of further papers.

7 Closing Remarks

An implementation of KeyMan written in Perl is available from an anonymous CVS download, described in `http://keyman.aldigital.co.uk/`.

KeyMan was designed and written by Ben Laurie and Matthew Byng-Maddick for A.L. Digital Ltd., and is an effort sponsored by the Defense Advanced Research Project Agency (DARPA) and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0537.

[1] this is needed otherwise a compromised key could obviously then be used to claim any time in the past

[2] this is needed because if the timestamp signer is human, they may not respond immediately

[3] this, we believe, is a good idea, because it allows the relying party to make their own decisions about how large a time window with which to trust the signer