UntrustedDebs - Debian Wiki

14-17 minutes

So this is just a set of ideas I want to braindump. It's not a formal proposal (yet?) or a flame festival preparation.

Contents

Purpose

Right now, dpkg is very trusting with the packages it installs. It runs all the maintainer scripts as root and installs files anywhere in the hierarchy with any set of permissions (suid, /etc files, etc). This is somewhat expected for some packages (say `passwd`) but less so of others (say `coreutils`).

In fact, I would be curious to try and figure out how many packages actually *don't* need root permissions to install themselves (short of having the files written to root-only directories). How many packages need suid or root-level maintainer scripts, really? And can we cut down how that number with a certain set of conventions?

The idea here is to protect users that install `sources.list` lines from random locations (say ?PPAs and unofficial package repositories). Such repositories shouldn't be able to install any random package at any time.

The other attack vector is the "malicious or compromised package maintainer". As much as we like to trust our multi-theist faith in Debian Developers, the attack surface is so wide that it's almost surprising that no one has yet introduced its own little backdoor in a random package's postinst file. As much as we want reproducible builds and follow package debdiffs (because everyone reads that religiously all the time too, of course), it's always possible to sneak something in there and the failure mode is just too catastrophic to not do anything to prevent this.

Here are my thoughts...

Threat model

The concerns here lie mainly into the idea that a determined attacker may be able to take over an APT repository and upload arbitrary packages. On the official Debian repositories, this could be done in one of those ways:

- compromise a Debian Developer's machine to steal secret key material and upload a malicious package
- compromise a DSA machine to bypass PGP key checks and upload a malicious package
- compromise a DSA archive-signing key to replace an existing package with a malicious package

On unofficial package repositories, the scope of this is expanded significantly, as they can replace any package they want freely, and there's no clear trust path between the user and the developer: such repositories often recommend running `curl | bash` style shell commands to setup their repositories, which in itself brings its own slew of attacks.

There are probably other ways of doing this I can't think of right now, but that gives a rough idea of the problem. The package doesn't need to be a critical package (like the Linux kernel or libc) to have significant impact, since any package runs maintainer scripts as root. An update wouldn't necessarily be noticed by the maintainer if the attack is against the infrastructure, because a package could be silently replaced.

Prior art

Secure apt

SecureApt was a pioneer and brought us pretty much end-to-end authentication of packages, but transitively, and introduced the above single point of failure. We can do better than this.

.deb signatures

dpkg supports signed binary packages through debsig-verify, which supports custom policies based on several parameters. The signatures on the binary packages can be performed with debsigs, but there are other compatible implementations. There are also other alternative implementations not currently supported by dpkg. None of this is currently being enforced or checked on install, as far as I know.

dpkg's builtin features

dpkg is actually pretty careful with the packages it installs. It will not overwrite another package's file with a package that's being installed, as long as the other package does not have a Replaces field, or uses pathname diversions. Maintainer scripts could replace pathnames based on triggers. Actually, that's pretty much all I can think of.

Ubuntu's "snappy"

Ubuntu made its own packaging format for phones, the .snap. Packages are constrained by a security policy that keep the packages from doing nasty stuff. It includes everything from capabilities, services, apparmor profiles, sockets, and so on.

Of course, it's a complete rewrite of a package management system, so hardly interesting for us here except for inspiration.

Android

Android is similar to Ubuntu's packaging system. In theory, Android "apps" all run in a sandbox (?) and explicitly request certain permissions to the user when installing. Recent versions of Android have a "Privacy guard" tool that actively monitors what apps do and can allow or disallow certain actions (access address book, camera, network, etc). The problem with this is the typical "warning fatigue" problem where you constantly get asked for permission and end up accepting whatever you get asked for.

Android also authenticate packages based on the developer's keys: when a package is installed the first time, Android remembers which key signed the package and will only allow updates to that package based on the same signing key, on top of the repository key.

I believe this may be running in some sort of JVM, but I'm not an android dev so I don't actually know (or care much right now).

This is also not usable for us other than for inspiration.

other sandboxes

insert your favorite sandbox idea here. the only one I can think of is an obscure project of rewriting the FreeBSD packaging format to run within the TCL sandbox, called `libh`, that never came to fruitition.

NixOS is similar to this: the install script is in large part declarative and you can install packages systemwide or per-user. But it's a different OS so we can't use this directly either, short of redoing the whole Debian packaging stack.

Reproducible builds

Reproducible builds are useful to make sure the binaries we have match the source code published. While this is very useful in ensuring integrity when we make an audit, its impact is limited for users, short of rebuilding all packages instead of installing binaries (which obviously brings into question how the source is trusted...)

TUF

The Update Framework is a "framework for securing software update systems". It defines a specification and a ?https://github.com/theupdateframework/tuf that does something equivalent to SecureApt, but with some improvements for certain attacks that we are still vulnerable for. This paper examines YUM and APT and determines which attacks are effective at, for example, denying upgrades or faking dependencies.

APT could be extended to support TUF metadata, although it is not clearly said if it would scale to the size of the Debian archive.

in-toto

in-toto is a "framework to secure the integrity of software supply chains". A complement to reproducible builds, it defines (another) specification (PDF) that defines ways to announce who did what to a piece of software when it was written, compiled, rebuilt, linted, tested, etc.

Implementation ideas

Here we cover various ideas, designs and concepts that could be used to improve the security of untrusted Debian packages.

Some conventions:

Those are ideas that are implemented.

Those are ideas with a design that need a first implementation.

Scoped secure apt

Apt could be better at enforcing which repository can install what. Sure, the official Debian repo can ship any package whatsoever, but why would a random PPA be able to replace the `passwd` binary without my consent?

In my mind, when I add an OpenPGP certificate to SecureApt, it should also certify which `Origin` (or `Label`? see DebianRepository/Format for that bikeshed) is allowed in the Release file. Then specific pinning could be enforced: by default, the `Debian` origin would be unrestricted (pin 990?) while others would be prioritised *only* if they do not replace an existing package (pin 200?). Even stricter policies could be enforced by saying that a specific key can only install a given package.

Package-specific scoping could also be implemented directly in `sources.list`.

That would solve part of the problem: a given repo could only install some package, and that's it. (In fact, there's a `Origin` field in the `CONTROL` file, does dpkg check that against the APT `Origin` field at all? I am guessing it's not.)

Note: the 1.1 release of apt now supports a `[signed-by...]` option in the `sources.list` to enforce at least which key signs which repository. There is a server-side config as well: DebianRepository/Format#Signed-By.

This can be successfully deployed right now on Debian stretch and later to enforce which packages can be installed or upgraded by a given repository. See RepositoryInstructions for the best practices in that regard.

A variation of this design was detailed in 858406, where a `pinmark` option is introduced in the `sources.list` file format that can then be used to uniquely pin certain packages in the preferences field.

End-to-end signatures with TOFU

The .deb signatures support could be leveraged to allow APT to check packages signatures themselves to confirm they really come from what they should be coming from. For example, packages could be checked against keys in the `debian-keyring` package on install. It could also "remember" which key signed with package and forbid untrusted keys (say outside of the debian keyring) from replacing packages already present. This could obviously present problems for NMUs and key rotations, but is certainly an interesting approach to the problem, as it would remove the single point of failure of trusting the central repository.

Permission checking

dpkg could refuse (or warn when) installing certain files with "bad" permissions. I think `suid` files are basically what the problem is, but we already mentioned overwriting other packages files, and there are definitely sensitive locations that should be avoided (`{/usr,}/lib`? `/home`? `/root`? others?).

User creation standardisation

Creating users from packages should be standardized. Right now, postinst scripts will generally call `adduser --system foo`. Or is it `adduser --system --disable-password foo`? Or `adduser --system debian-foo`? It changes from one package to the other!. What about the `UID`? Will it stay consistent across platforms? And so on...

Having a metadata in (say?) the package's `control` file would allow `dpkg` to prompt the operator for creating the user, and then safely create it, consistently.

The point of this is to reduce the number of things needing root in the postinst scripts.

There's already 685734 requesting the feature of dpkg to manage users and groups. There is also a debhelper extension to manage system users called dh-sysuser.

Services auto-configuration

Right now, services are installed and started as part of the postinst script, basically by hand, by fiddling with `update-rc.d` (for sysvinit) and by dropping the right file (in `init.d` for sysv, `.service` files for systemd), and then starting the service with `service foo start`.

In itself, starting the service isn't a bad thing if it is running in a user sandbox (assuming no privilege escalation here, please bear with me). But we may want to ask the user for such a thing as well: installing a text editor shouldn't start a web server (unless you're in some weird Ignucious church, in which case it

may also be a freaking window manager, apparently).

The point of this is also to reduce the number of things needing root in the postinst scripts.

Apparmor

Apparmor profiles are great and very useful, but basically, it still gives the package all the permissions it wants to do whatever it wants. Maybe those could be maintained as a central directory of "things that packages are allowed to do", outside of the main package definition. Of course, that means moving the problem elsewhere, but it's easier to review changes to such a meta-data-only package than when it is lost amongst piles of other diffs.

The right to install apparmor profiles itself could be a specific permission as well.

Limiting arbitrary code execution

Packages that still require arbitrary code execution (e.g. I believe the `dash` package maintainer scripts are compiled to binary code to avoid bootstrapping issues?) would still be necessary, but that could be a specific permission as well. Such arbitrary scripts could also be run in a package-specific user sandbox (defined above) to limit potential damage, making this two permissions ("run as root", "run as the package user").

Heck, while we're at it: why wouldn't all of the dpkg run under a given apparmor profile? There could be a default apparmor profile, then exceptions in the package's `control.tar.gz` that could be reviewed by the user prior to installing the package...

Debconf

Debconf is a pile of stuff that would need to be worked on. I really don't know how to deal with it, but it seems to me that it could be somewhat suid so that it would run unprivileged somehow. Maybe the debconf protocol could be abused to provide some sort of separation here as well... not sure.

Declarative maintainer scripts

An idea from liw: http://blog.liw.fi/posts/declarative-deb-maintainer-scripts/

Then there's also declarative diversions, a SoC project with similar ideas, but for dpkg-divert: SummerOfCode2011/DeclarativeDiversions.

There are plans in motion for more declarative operations in dpkg (see https://lists.debian.org/debian-dpkg/2015/08/msg00031.html).

User experience

I am afraid of what this would look like for the user. Unfortunately, all I can think of right now is some stupid and incomprehensible dialog asking the user weird questions like "do you want to allow package X to do Y?" But it beats nothing at all...

We could specify a certain set of permissions, granted by package name and/or origins. A set of predefined policies, like apparmor profiles, could help with this.

History

This was first written in august 2015 by TheAnarcat at a time where flames were missing, I guess.

Please consider commenting inline or editing the proposal to enhance it before commenting here. -- TheAnarcat

See also

- SecureApt
- RepositoryInstructions

CategoryPackaging CategorySystemSecurity