

# Développement

- [Git](#)
  - [Signer ses commits Git et transférer son gpg-agent sur un serveur distant](#)
- [Vim](#)
  - [Gestion native des plugins](#)
- [Divers](#)
  - [Ajouter des couleurs à ses scripts shell](#)

# Git

Git

# Signer ses commits Git et transférer son gpg-agent sur un serveur distant

[GPG](#), c'est bien. C'est encore ce qu'on a trouvé de mieux pour que tout un chacun puisse s'assurer de l'authenticité d'un message (non-modification de celui-ci et que son auteur est bien celui annoncé) et chiffrer ses messages.

OK, c'est pas un exemple d'ergonomie, OK, c'est pas ma mère qui va s'en servir sciemment tous les jours (sous le manteau, si, puisque les paquets de sa Debian sont signés avec GPG). Mais d'un autre côté, ma mère ne risque pas non plus (et je dirais même encore moins) de se payer un certificat X509 pour signer ses mails. Ne parlons pas de mon père, à côté de lui, ma mère fait figure de [hax0r](#).

Bon, d'accord, c'est un bon gros truc de geek. OSEF, c'est cool quand même, c'était pour dire que c'était robuste et que tout le monde peut l'utiliser sans bourse délier.

Que vous le sachiez (*dans la colle*) ou pas, on peut [signer ses commits git avec GPG](#), histoire d'ajouter encore une couche de sécurité aux modifications qu'on apporte à un logiciel. Des forges comme Gitlab permettent d'[ajouter une clé GPG à son profil](#) permettant ainsi de vérifier les signatures des commits d'un projet. Voyez sur [ce commit](#) le joli petit bouton « *Verified* ».

**Ceci n'est pas un cours sur GPG, on va donc considérer que vous avez déjà une clé GPG.**

## Signer ses commits Git

Rien de plus simple. Il faut tout d'abord déclarer à Git quelle clé doit être utilisée pour signer ses commits (changez l'empreinte, ça c'est la mienne) :

```
git config --global user.signingkey EA868E12D0257E3C
```

Maintenant, soit vous ajoutez `-S` quand vous committez :

```
git commit -S
```

Soit vous configurez Git pour signer tous vos commits :

```
git config --global commit.gpgsign true
```

Voilà, c'est bon. Pour vérifier un commit :

```
git verify-commit cce09ca
```

C'est bien beau, mais ça m'arrive de développer directement sur des serveurs, et surtout, je développe généralement dans une machine virtuelle sur mon PC. Je ne vais certainement pas aller copier ma clé privée sur les-dits serveurs ou dans la machine virtuelle ! C'est là qu'intervient le [transfert du gpg-agent sur le serveur distant](#).

**Ceci n'est toujours pas un cours sur GPG, on va donc considérer que vous avez déjà un gpg-agent fonctionnel sur votre ordinateur.**

# Transférer son gpg-agent sur un serveur distant

**ATTENTION** On ne transfère son agent gpg que sur une machine dans laquelle on a confiance, et dont on sait que les personnes y ayant accès ne s'amuseront pas à utiliser votre agent (rien de plus simple si on a un accès root à la machine). Cela vaut aussi pour l'agent ssh !

Premièrement, on va dire à l'agent de créer un [socket](#) supplémentaire en mettant dans

```
~/.gnupg/gpg-agent.conf
```

 (remplacez `<user>` par votre login) :

```
extra-socket /home/<user>/.gnupg/S.gpg-agent.extra
```

Ce *socket* a des restrictions que n'a pas le *socket* habituel (ne me demandez pas lesquelles) mais surtout, le logiciel qui vous demandera votre mot de passe (*pinentry* de son petit nom générique) vous présentera la demande de mot de passe différemment d'habitude. Moi, il m'a dit en gros « Cette demande provient d'une machine distante », ce qui permet de repérer d'où vient la demande (déjà pas de votre machine pour déchiffrer un mail par exemple) et de réfléchir à si c'est bien vous qui avez fait une action demandant la clé GPG.

On redémarre l'agent pour prendre en compte la nouvelle configuration :

```
gpg-connect-agent /bye
```

Ensuite, il va falloir modifier sa configuration SSH. Comme un bon adminSys est fainéant, vous avez bien sûr utilisé [concierge](#) pour gérer votre fichier `~/.ssh/config`. Il suffit d'ajouter dans le bloc

de configuration du serveur souhaité la ligne (remplacez `uid` par votre *uid* (`id` pour le connaître) et `<user>` par votre login):

```
RemoteForward /run/user/<uid>/gnupg/S.gpg-agent /home/<user>/.gnupg/S.gpg-agent.extra
```

Enfin, il faut ajouter ceci dans le `/etc/ssh/sshd_config` du serveur distant (et redémarrer son démon ssh après) :

```
StreamLocalBindUnlink yes
```

C'est fini ! Vous pouvez maintenant utiliser votre clé GPG sur un serveur distant en vous y connectant en SSH, sans copier votre clé sur le serveur

Connectez-vous en SSH et testez avec

```
echo "test" | gpg2 --clearsign
```

Si, lorsque vous tentez de signer un commit sur votre serveur distant, cela échoue, assurez-vous que git utilise bien `gpg2` :

```
git config --global gpg.program gpg2
```

Merci à [Thomas Citharel](#) pour avoir mis la signature GPG des commits sur le tapis d'une discussion, ce qui m'a poussé à me pencher sur le transfert de l'agent GPG.

# Vim

Vim

# Gestion native des plugins

Depuis Vim 8, il est extrêmement facile d'installer des plugins dans Vim.

Tout d'abord, on crée un dossier `~/.vim/pack`

```
mkdir -p ~/.vim/pack/
```

Dans ce dossier, on peut créer différents dossiers, qui seront autant de *packs*, des ensembles de plugins.

On pourra ainsi avoir un pack qui contient les plugins spécifiques aux thèmes, à un langage de programmation, etc.

Création d'un *pack* appelé `base`, avec les dossiers qui vont bien dedans :

```
mkdir -p ~/.vim/pack/base/{opt,start}
```

Les plugins qu'on placera dans le dossier `~/.vim/pack/base/start/` seront chargés au démarrage et les plugins qu'on placera dans le dossier `~/.vim/pack/base/opt/` seront chargés à la demande avec la commande `packadd le_nom_du_plugin`.

Voir <https://vimhelp.org/repeat.txt.html#packages> pour creuser le sujet.

# Divers



# Ajouter des couleurs à ses scripts shell

Repris de <https://stackoverflow.com/a/28938235>.

Parce que la vie est plus agréable avec des couleurs, j'aime bien agrémenter mes scripts avec des couleurs.

Outre le fait que ce soit plus joli, cela permet aussi de rendre plus lisible la sortie d'un script (un texte en rouge, on se doute que tout ne s'est pas bien passé avant même de lire le texte).

Je note donc ici les codes pour écrire en couleur dans des scripts shell, en tant que pense-bête.

## Les codes

## Remise à zéro du formatage

```
NC='\033[0m'
```

## Couleurs de base

```
BLACK='\033[0;30m'  
RED='\033[0;31m'  
GREEN='\033[0;32m'  
YELLOW='\033[0;33m'  
BLUE='\033[0;34m'  
PURPLE='\033[0;35m'  
CYAN='\033[0;36m'  
WHITE='\033[0;37m'
```

## En couleur et en gras

En gras, c'est *bold* en anglais, d'où le préfixe `B`.

```
BBLACK='\033[1;30m'  
BRED='\033[1;31m'  
BGREEN='\033[1;32m'  
BYELLOW='\033[1;33m'  
BBLUE='\033[1;34m'  
BPURPLE='\033[1;35m'  
BCYAN='\033[1;36m'  
BWHITE='\033[1;37m'
```

# Utilisation

```
RED='\033[0;31m'  
NC='\033[0m'  
echo -e "${RED}Hello world${NC}"
```

On notera ici trois choses :

1. l'encadrement de la variable par des accolades, pour la séparer du texte (sinon le shell essaierait d'interpréter la variable `$REDHello`, qui n'existe pas) ;
2. l'utilisation de l'option `-e` d'echo : selon le shell utilisé, le comportement sera différent (les shells bash, dash, zsh (et les autres aussi, sans doute) utilisent leur propre commande `echo`). L'option `-e` est nécessaire pour activer l'interprétation des backslash pour `/usr/bin/echo`, pas pour le `echo` de zsh. Dans le doute, il faut utiliser cette option.
3. l'utilisation de la variable de remise à zéro à la fin de la phrase. Certains shells vont conserver le changement de formatage de texte après le `echo` (bash, dash), d'autres non (zsh). Encore une fois, dans le doute, on remet le formatage à zéro.

On [m'a signalé](#) que `printf` était plus standardisé, donc au comportement plus constant entre les shells. Avec `printf`, ça donne :

```
RED='\033[0;31m'  
NC='\033[0m'  
printf "${RED}Hello world${NC}\n"
```

À noter, le `\n` final, car `printf` ne fait pas de retour à la ligne automatiquement.