Universal Cryptographic Signing for Git

—

Jimit Bhalavat - Hyperledger 2020 Mentee
David Huseby - Hyperledger Security Maven

Project Goals

- Add to Git the ability to support any signing/verification tools.
  - Create ability to support signed patches submitted by email.
- Simplify and unify config for signing/verification.
- Normalize command line switches.
- Maintain backwards compatibility.

Project Approach

- Abstract away signing/verifying tool specifics so that Git can use any tool, not just GPG.
  - Remove GPG specific code, config, and naming (e.g. `--gpg-sign` command line switch).
  - Reorganize and normalize config settings to support multiple tools (i.e. `sigs.openpgp.*`, `sigs.x509.*`, `sigs.minisign.*`, etc)
  - Make signatures stored in Git objects "self-describing" and optionally "self-verifying".
    - Remove string matching/regex code to find and extract signatures from objects.
    - Add tagged fields for signature type, signature data, optional verification options for verification tool, and optional public key used for verification.
- Preserve backward compatibility.
  - Deprecate, but maintain support for, old config and command line switches; with warnings.
  - "Legacy" signatures on objects and tags are detected and assumed to be either GPG or GPGSM (X.509) and processed with sane defaults even without new-type config settings.

Currently, How Signing Works

1. Git builds a command line to execute a signing tool from its config and command line switches.
    - Command line includes the signer identity/key identifier for the signing tool to use.
2. Git pipe-forks the command line (default: `gpg -bsau "Jane Hacker <jane@h.com>"`)
3. Git sends the data to be signed over the signing tool's stdin pipe.
4. Git reads the detached, ascii encoded signature over the signing tool's stdout pipe.
5. If failure (e.g. signing tool process exits >0) stderr messages are printed to Git's stderr/log.
6. If success, signature data is stored tagged under "**`gpgsig:`**" in Git commit object or as raw, un-tagged signature appended to Git tag.

Currently, How Verification Works

1. Git uses string matching to find the "`-----BEGIN SIGNATURE-----`" and "`-----END SIGNATURE-----`" lines in a Git commit/tag and saves the signature to a temporary file.
2. Git builds a command line to execute the verification tool from the config and command line switches.
    - Command line includes the path to the detached signature file (default: `gpg --verify /tmp/gitXXXX-signature -`)
3. Git pipe-forks the command line.
4. Git passes the signed data over the verification tool's stdin pipe.
5. Git reads the signature status from the verification tool's stderr pipe.
6. Git parses the status looking for "`GOODSIG`" and other data to show to the user (e.g. `git log --show-signature`)

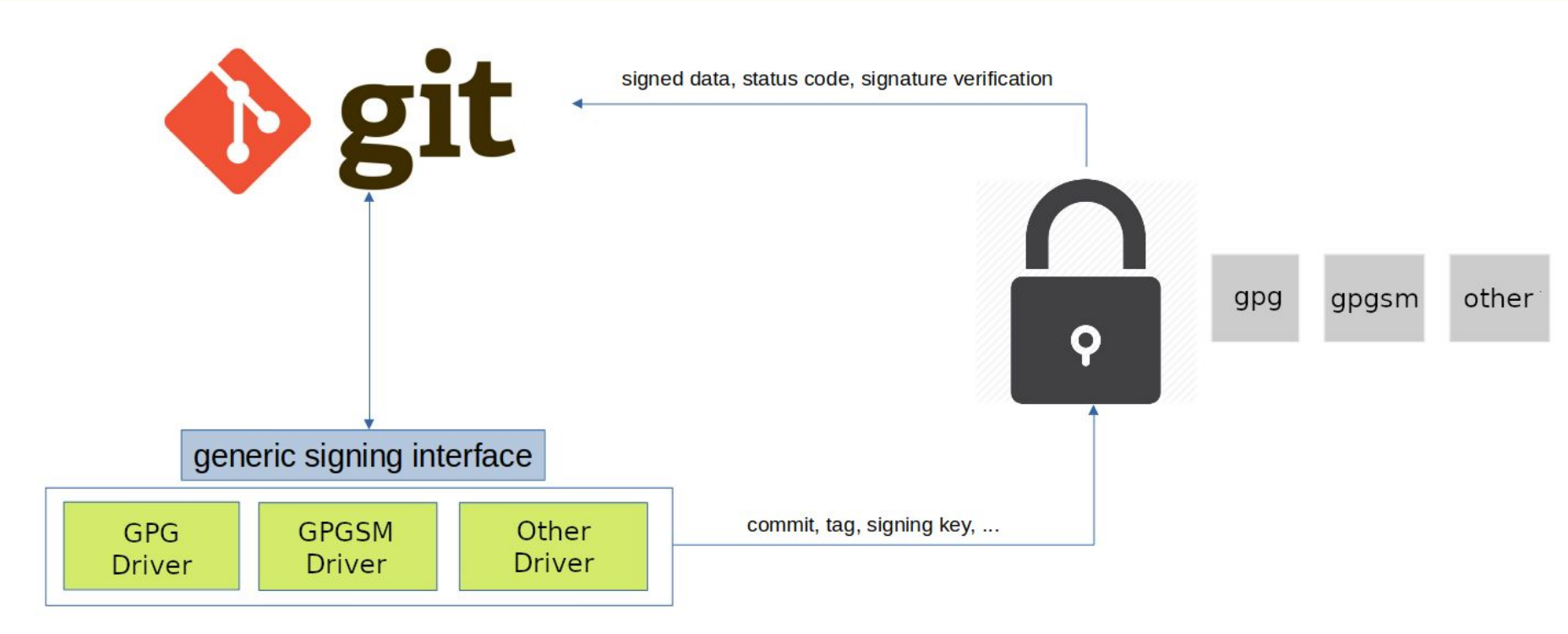
2019 Hyperledger Mentorship Project

Ibrahim el Rhezzali

Ibrahim's Approach (Hyperledger 2019 Mentee)

- Abstract current GPG-specific interface into a generic API with "drivers" for each signing tool.
- Reorganize Git config into **`signing.*`** structure with tables for each signing tool: **`signing.openpgp.*`**, **`signing.x509.*`**, etc.
- Preserve backwards compatibility for a limited time by deprecating old config fields and having OpenPGP as default signing/verifying tool.
- **Rejected** because the "drivers" for signing tools were written in C and the Git developers didn't want to have to review a bunch of tool-specific C code and were also worried about ongoing maintenance.
- New signing interface API with pluggable drivers
  - [Patch 0/5], [Patch 1/5], [Patch 2/5], [Patch 3/5], [Patch 4/5], [Patch 5/5]

Ibrahim's Approach (Hyperledger 2019 Mentee)



Source: [Git Commit Signing Mainpage 2019.]

Ibrahim's Approach (Hyperledger 2019 Mentee)

```
[signing]
     default = "openpgp"

[signing."openpgp"]
     program = "/usr/bin/gpg"
     identity = "Jane Hacker <jane@h.com>"

[signing."x509"]
     program = "/usr/bin/gpgsm"
     identity = "0123456789ABCDEF"
```

2020 Hyperledger Mentorship Project

Jimit Bhalavat

Options Going Forward

- Standard Pipe-Fork Interface and Config Based
  - Complicated config for specifying how each and every tool is to be called.
  - This solution would have to deal with all of the legacy complication of signing tools.
- Protocol-based Approach Inspired by GPGme (i.e. Assuan Protocol)
  - Cleanly separates Git from tool-specific details and code.
  - Makes .gitconfig for each tool simple.
  - Stores verification options—and optionally the pubkey—inside of the Git object.
    - Requiring pubkey for all commits/tags makes the Git repo *self-verifiable*. Enforceable with existing hook script feature.
  - Allows for `git format-patch` and `git am` to support signed emailed patches.
  - Externalizes signature creation and verification
    - Done over any IPC (e.g. pipe, shmem, socket) to local/remote crypto services.
    - Allows for integration into enterprise identity systems while allowing external parties the ability to verify signatures via verification options and pubkey in Git objects.

Protocol Based Approach - Config Simplification

```
[sigs]
      default = “openpgp”                                                    # override with ‘git --sign=<format>’

[sigs.”openpgp”]
      program = “/usr/bin/gpgme-server”                                      # example of single tool command
[sigs.”openpgp.options”]
      identity = “Jane Hacker <jane@h.com>”
      min_trust_level = “marginal”
      armored = true
      detached = true

[sigs.”openssl”]
      sign = “/usr/bin/openssl -sign”                                        # example of separate tool commands
      verify = “/usr/bin/openssl -verify”
[sigs.”openssl.options”]
      privkey = “/etc/letsencrypt/live/hyperledger.org/privkey.pem”
      digest = sha256
      encoding = base64

[sigs.”x509”]
      program = “socat stdio openssl-connect:10.0.0.100:1337,cert=/etc/client.pem” # TLS socket to X.509 HSM
[sigs.”x509.options”]
      cert = “hyperledger.org”
```

Protocol Based Approach - Signing

1. Git gets executable from config/command line and pipe forks.
2. Git issues `OPTION` commands to pass options from the config/command line to signing tool.
3. Git issues `SIGN` command and passes data to be signed.
4. Git reads back signature type, verification options (optional), verification public key (optional) and the signature data.
5. Git stores `sigtype`, `sigoptions` (optional), `sigkey` (optional), and `sig` in the Git object.

Protocol Based Approach - Signing

```
S: OK
C: OPTION identity=Jane Hacker <jane@h.com>
S: OK
C: OPTION min_trust_level=marginal
S: OK
C: OPTION armored=true
S: OK
C: OPTION detached=true
S: OK
C: SIGN
C: D <lines of object data>
C: D …
C: END

S: D sigtype:openpgp
S: D sigoptions:min_trust_level=marginal
S: D sig:<lines of signature data>
S: D …

S: OK
C: BYE
S: OK
```

Data stored inside of the Git object for the verification tool to process. The **`sigtype`** identifies which verification tool to use.

Protocol Based Approach - Verification

1. Git parses the **`sigtype`**, **`sigoptions`** (optional), **`sigkey`** (optional), and **`sig`** data from the Git object.
2. Git uses the **`sigtype`** to get the verification tool executable from the config file.
    - If there is no **`sigtype`** field, the signature is parsed to determine if it is an **`openpgp`** or **`X.509`** signature. The **`sigoptions`** field that doesn't exist is given sane defaults for verification. This is to maintain backward compatibility with current GPG/GPGSM sigs.
    - If there is a **`sigtype`** field and the local installation doesn't have a config for the **`sigtype`** then the signature verification is skipped for that object with a warning.
3. If there are **`sigoptions`**, Git uses **`OPTION`** commands to pass the options to verification tool.
4. If there is a **`sigkey`**, Git issues **`KEY`** command to pass the public key to the verification tool.
5. Git issues **`SIGNATURE`** command to pass the signature data to the verification tool.
6. Git issues **`VERIFY`** command to passes the signed data to the verification tool and trigger verification.
7. Git reads back the verification status from the verification tool.

Protocol Based Approach - Verifying

```
S: OK
C: OPTION min_trust_level=marginal
S: OK
C: SIGNATURE
C: D <lines of signature data>
C: D …
C: END
S: OK
C: VERIFY
C: D <lines of signed object data>
C: D …
C: END
S: D Signature made Sun 18 Oct 2020 03:14:17 AM PDT using RSA key ID DFBBCC13
S: D Good signature from “Jane Hacker <jane@h.com>”
S: OK
C: BYE
S: OK
```

Protocol Based Approach - Signing `git format-patch`

Konstantin's email: https://lore.kernel.org/git/20190910121324.GA6867@pure.paranoia.local/

```
S: OK
C: OPTION key=/home/user/.minisign/minisign.key
S: OK
C: OPTION verified_comment="\s*Signed-off-by:.*"
S: OK
C: SIGN
C: D <lines of patch data to sign>
C: D …
C: END
S: D sigtype:minisign
S: D sigkey:RWT9fcUvSnHPLqqyfLbkGBMEscBWciFFp2iBj2XnZPzW69OVIoYwZ25q
S: D sig:RWT9fcUvSnHPLiqWgXEnn98sgk8nl4FteDRkD+9lVK+He//eLOxNZ5QjCROoKJgPGpL4uzoHicN+f6gB54qmtO1cQtfvjS+++QU=
S: OK
C: BYE
S: OK
```

Minisign supports cryptographically verified comments. This regex specifies which data should be extracted as the verified comment. This approach means that all Signed-off-by lines are cryptographically bound to the sig.

Signature data stored in the formatted patch email sent to the mailing list.

Protocol Based Approach - Verifying `git am`

Konstantin's email: https://lore.kernel.org/git/20190910121324.GA6867@pure.paranoia.local/

```
S: OK
C: KEY
C: D RWT9fcUvSnHPLqqyfLbkGBMEscBWciFFp2iBj2XnZPzW69OVIoYwZ25q
C: END
S: OK
C: SIGNATURE
C: D RWT9fcUvSnHPLiqWgXEnn98sgk8nl4FteDRkD+9lVK+He//eLOxNZ5QjCROoKJgPGpL4uzoHicN+f6gB54qmtO1cQtfvjS+++QU=
C: END
S: OK
C: VERIFY
C: D <lines of signed emailed patch>
C: D …
C: END
S: D Signature and comment signature verified
S: D Trusted comment: Signed-off-by: Konstantin Ryabitsev <konstantin@linuxfoundation.org>
S: OK
C: BYE
S: OK
```

Signature data parsed from the emailed patch.

Cryptographically trusted comment is the "Signed-off-by" message in the email.

Jimit Bhalavat's Work (2020)

- Technical Design Document (Deprecated in lieu of the protocol design)
- Technical Design Proposal (Deprecated in lieu of the protocol design)
- Git Commit Signing Mainpage 2020