UVSource

End to end encrypted distributed version control system

Problem and motivation

Private repositories on github, bitbucket, gitlab, ... are NOT Private in any reliable way.

Attackers can take control of github infrastructure and thus gain control of anyone's repository, including source code, experimental features, issues, documentation, unresolved bugs, etc etc. These could contain valuable trade secrets.

Additionally, many govts have the power to compel github or its data centers to grant access to the physical infrastructure. when counting all the intelligence partnerships that exist, the list of governments around the world with such power is much too long.

Both the integrity and confidentiality of the private repos are at risk.

Problem contd

Apart from github infrastructure being a potential risk, the transport protocol (i.e. https) may not be as safe as many ppl think it is. (DigiNotar example)

If https communication is not secure, users can “push/pull” to/from malicious man-in-the-middle nodes instead of github, bitbucket, ...

All good with https, right?

DigiNotar

Dutch based Certificate Authority that was widely trusted by major browsers. On Sep 2011 it became clear that DigiNotar root certificates had been stolen for several years, most likely by Iranian intelligence services, who used it to set up fake gmail, yahoo, ... servers and launch man-in-the-middle attacks against Iranian users of those services. https did nothing for these ppl except provide a false sense of security.

A team of dvcs users can ask themselves, if we can very well encrypt our own files before sending them to the cloud, why should we trust git's https implementation (which probably gets updated even less frequently than the browser). All it takes to fool git into thinking it is connected to github (when in fact it is connected to a malicious node) is a compromised root certificate of a CA that git trusts, like DigiNotar.

While https is probably pretty good at stopping low and medium level criminals, how hard is it for most governments to compel CAs to grant them access? We know the FBI has already publicly demanded access to Apple's root certificates. It would be extremely naive to not presume, they have done so much more in secret.

If the government of Iran of all places, has the capability to break https, we should all be worried.

On the other hand, If all you are sending to a dvcs hosting like github, is ciphertext to begin with, the loss of transport layer security is not as catastrophic. The worst that can happen is you lose your paid github account, but the confidentiality and integrity of your repository is maintained.

Alternative to github, manage own infrastructure

First problem, what is meant by "my own infrastructure"? An EC2 instance is just as vulnerable as github to both criminals and governments.

I would need physical control of the servers to declare them as safe as my development machine. But now I have to manage availability, redundant storage, redundant power supply, redundant network access and much more to setup git hosting and make it be as available as github.

On top of that now I have to set up vpn/ssh or something else to allow my team to securely connect to this machine from home, office, elsewhere. Then you have to tell your team members to use openvpn, use ssh, install putty, install cygwin, upload your keys, set up ssh-user agent, ...

Not easy, not convenient and not in every team's time/money budget.

Its makes sense for a 3rd party like GitHub, to capitalize on the economies of scale, and provide high reliability and availability dvcs hosting services on cheap.

End to end encrypted dvcs

The idea is simple. Allow the dvcs users to put a password on their repositories. The owner sets a repository password at initialization time. This password is required for all future interactions with that particular repository and must be kept secret.

Encrypt everything the user puts in the repository (i.e. when users make a new commit) on the end user's machine with a secure AEAD scheme (i.e. XChaCha20-poly1305, or AES256-GCM or AES256-CBC-HMAC-SHA256)

When the user pushes commits to a high availability 3rd party cloud. send only ciphertext.

When pulling data from the 3rd party cloud, decrypt everything again on the end user machine and give it out. Hence "end to end encrypted dvcs"

No need to trust the 3rd party cloud with plaintext.

Confidentiality in the context of encrypted dvcs

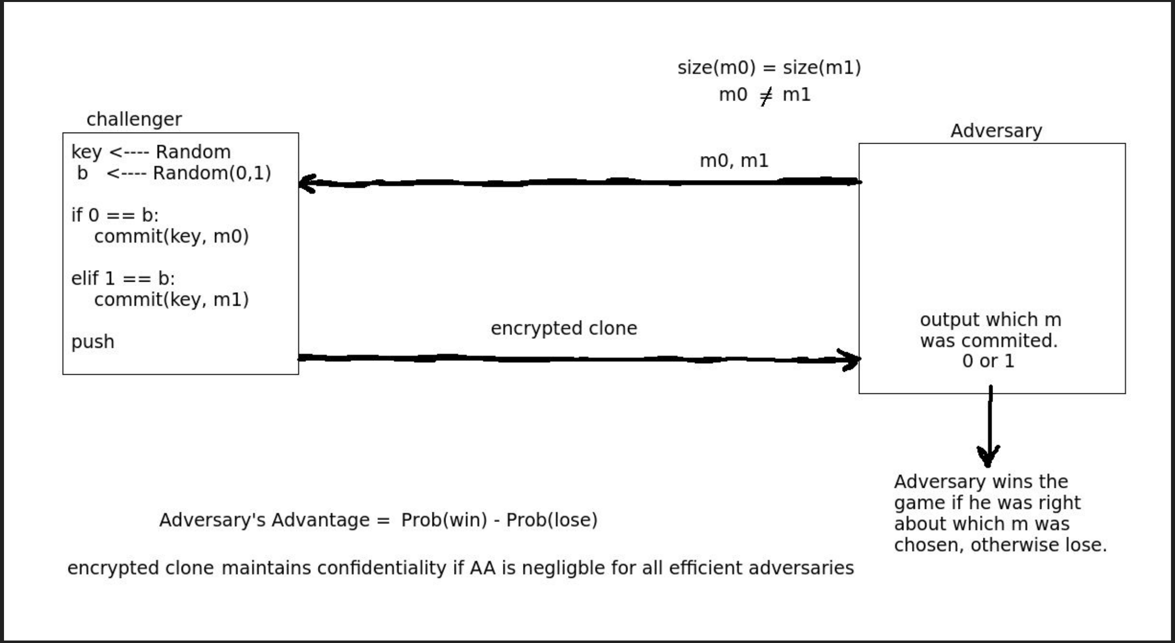
End to end encrypted dvcs provides cryptographic confidentiality and integrity. Lets now discuss what we mean by these.

Confidentiality in the context of end to end encrypted dvcs:

- Intuitively it means that an adversary with access to an encrypted remote clone (in git terminology this is equivalent to a headless clone) learns nothing about what the user put in the repository (files, directories, commit messages, …).
- More precisely it means all computationally efficient adversaries, have no better than a negligibly small[1] advantage at winning the commit distinction game (next slide) .

1. For example 2^-90 is negligibly small, 1/100 is not.

Commit distinction game


size(m0) = size(m1)
m0 ≠ m1
challenger
Adversary
m0, m1
encrypted clone
output which m was commited. 0 or 1
Adversary wins the game if he was right about which m was chosen, otherwise lose.

```
key <---- Random
 b   <---- Random(0,1)

if 0 == b:
    commit(key, m0)

elif 1 == b:
    commit(key, m1)

push
```

Adversary's Advantage = Prob(win) - Prob(lose)

encrypted clone maintains confidentiality if AA is negligble for all efficient adversaries

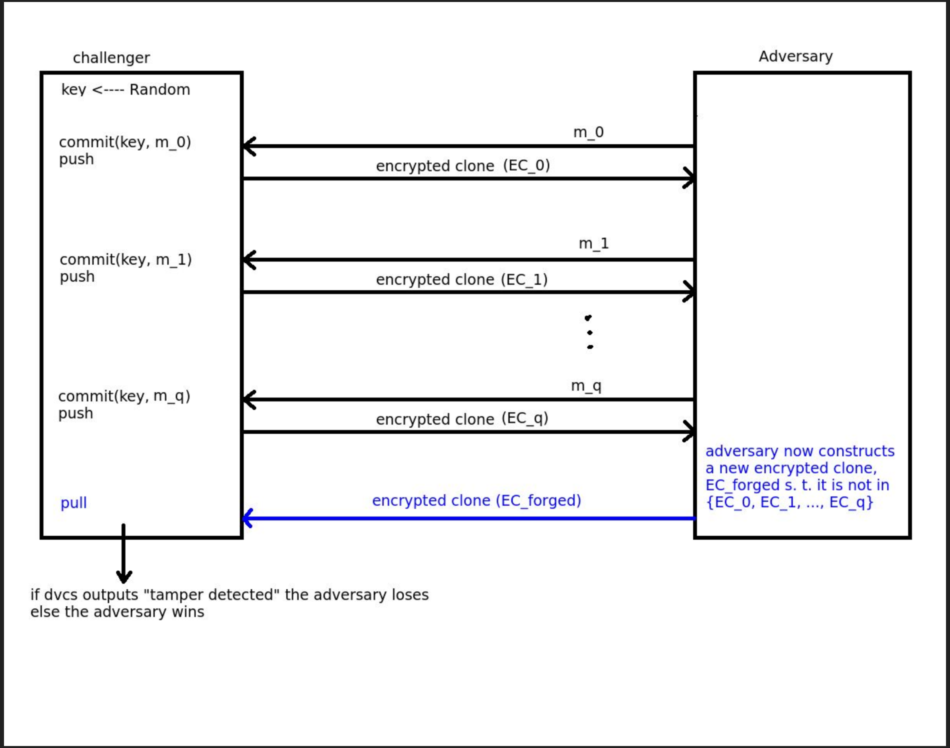
Integrity in the context of encrypted dvcs

What is meant by integrity in the context of end to end encrypted dvcs?

- Intuitively it means that an adversary with full control of the underlying infrastructure behind a remote encrypted clone (for example a git equivalent would be: the adversary controls github servers) can not modify the encrypted clone in any way without being detected. (examples of modifying include but are not limited to: reordering files, file contents, change commit messages, introduce garbage bit patterns in the middle of a file etc etc)

- More precisely it means all computationally efficient adversaries, have no better than a negligibly small chance of winning the repository existential forgery game.

Repository existential forgery game



UVS crypto

UVSource is in alpha as of this writing. So internal details and API can change in the future. So far

- kdf[1] is PBKDF2-SHA256-HMAC with iteration count of 120k
- per repository random salt generated by OS kernel for each repo at init time.

kdf derives 3 keys:

- key_enc: actual encryption key use with AES (so far encryption[2] is AES in CBC mode w/ random IVs, from OS)
- key_mac: key used for generating Message Authentication Codes for integrity (sha256-hmac as of now)
- key_sefp: storage engine fingerprinting key.

---

1. Support for argon2, for key derivation function is planned for near future.
2. Support for XChaCha20-poly1305 and AES-GCM is planned for near future.

UVS crypto

For every new object seen by the version control system two things are generated and stored in the encrypted history.

- A ciphertext version of that object (this is randomized encryption for ciphertext indistinguishability under CPA) encryption mode as of right now is AES-CBC-SHA256-HMAC (encrypt then MAC for CCA security) with random IVs derived from OS kernel random source
- A deterministic fingerprint of that object (called sefp, for storage engine fingerprint) these are equivalent to git's sha1sums. The storage engine needs a deterministic and collision resistant way of fingerprinting previously seen objects so they need not be stored again. In UVS these are keyed hashes.

SEFP = SHA3_256(key_sefp || object) or

SEFP = HMAC-SHA256(hmac_key= key_sefp, message=object)

The choice of which hash function is used is configurable on a per repository basis. Any one of the SHA3 era hash functions are good, if sha3 is unavailable hmac versions of sha2 era functions are used instead.

Additional Application: password manager

Once a dvcs becomes encrypted, it unlocks new use cases. A lot of problems benefit from a confidentially remote backed up, replicated and versioned filesystem.

UVS is inherently a password manager. It has some benefits over using a traditional password manager like keepass + dropbox or bittorrent to sync the keepass db file.

- Keepass + dropbox/bittorrent setup is definitely as confidential as UVS, but managing the keepass db file can be a problem.
- What if a corrupted version of the db was saved on a device (either accidentally or by an attacker)?
- The sync mechanism could overwrite all other devices with this corrupted file. Hence the replicated copies are still a single point of failure.
- In an encrypted dvcs this problem does not exist due to versioning. Also the user can choose when to pull/push new commits from one clone to another.

Additional Application: secure replicated backups

- Let's say bob wants weekly back ups of his home folder. All he needs:

```
$ cd ~
$ uvs init
```

His home folder is now a uvs repository, all he needs is to schedule weekly jobs to run:

```
$ uvs commit
$ uvs push
```

If he wants replication, just add more remotes:

```
$ uvs push  _remote1_in_the_cloud_
$ uvs push  _remote2_local_NAS_
```

Secure replicated backups contd

The encrypted dvcs already takes care of only storing the changes from a previous commit (snapshot of bob’s home folder in this case) rather than full copies every time. This is superior to using tarballs and cron jobs for this reason.

Assume bob’s home folder was 100 GB in size but his changes are only 10 MB per week on average.

Storage cost for one year worth of weekly backups, in uvs:

100GB + (10MB * 52) = 200 GB approx.

Cron job + tarball backups:

100GB * 52 = 5 TB approx.

Additional Application: Encrypted distributed discussion board / issue tracker

Subversion and CVS needed a central server to coordinate everything. For them this was pretty critical piece of the software. Back then it would have been very hard to imagine subversion without a central server. However git showed that a central server is not needed, for a version control system to work (in fact it works even better with the distributed model).

Lets think about discussion boards / issue trackers (like Piazza, myBB, phpBB, bugzilla, …..) what do these tools fundamentally do? They allow users to:

- Create new threads/issue.
- Append new replies to these threads/issues.

Nothing about these operations requires a central server let alone storing plaintext version of everyone’s posts.

UVBoard: distributed discussion board/issue tracking

- Model the underlying the state of the discussion board as regular files sitting on top of a dvcs (like git or uvs)
- Make one file per thread/issue, when users post to threads, append to this file. Let the dvcs version these files.
- The system simply reads these files and projects them a web page as a discussion board / issue tracker UI.
- When users reply to posts via the web ui, append to these files, and commit them.
- login/userid system replaced by who has commit access to the underlying repository.