Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor thesis

Privacy implications of exposing Git meta data

presented by

Arne Beer
born on the 21th of December 1992 in Hadamar
Matriculation number: 6489196
Department of Computer science

submitted on January 6, 2018

Supervisor: Dipl.-Inf. Christian Burkert
Primary Referee: Prof. Dr.-Ing. Hannes Federrath
Secondary Referee: Prof. Dr. Dominik Herrmann

Abstract

Even if you’re not doing anything wrong, you are being watched and recorded.

Edward Snowden

Contents

Acronyms

API Application programming interface

FS file system

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notification

ORM Object-Relational Mapping

SHA-1 Secure Hash Algorithm 1

SSH Secure Shell

SQL Structured Query Language

URL Uniform Resource Locator

UTC Coordinated Universal Time

VCS version control system

Chapter 1
Introduction

Git is a code version control system which is used by most programmers on a daily basis these days. According to the Eclipse Community Survey about 42.9% of professional software developers used git in 2014 with an upward tendency [1]. It is deployed in many if not most commercial and private projects and generally valued by its users. It allows quick jumps between different versions of a project's code base and to manage and merge code from different sources to one upstream.

Several million users send new commits to their Git repositories every day. On Github alone, the currently biggest open source platform, there exist about 25 million active repositories, a total of 67 million repositories and about 24 million users [2].

Some well known projects and organizations use Git, for example Linux[3], Google[4], Adobe[5] and Paypal[6]. Every repository contains the complete contribution history of every contributing user. Each commit contains the full directory structure, a link to a blob for every file, a timestamp, a commit message from the author and more additional metadata.

This raises the question how much information is hidden in the metadata of a Git repository and which attack vectors could be introduced by mining this information, regarding a contributer or the owner of the repository.

The newly gained knowledge could be utilized by employers to spy on their employees. It could be used by an unknown attacker who aims to obtain sensitive information about a company and its employees trough their open-source projects. It is even possible that a privat person wants to monitor another person that regularly contributes to open-source repositories.

As there have not been any papers published about this specific topic or at least no public paper and Git plays such a crucial role in todays information technology, I want to investigate and evaluate this potential threat.

[1] Ian Skerret. Exclipse Community Survey 2014 Results. `https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/` Retrieved Oct. 25, 2017

[2] The State of the Octoverse 2017, Retrieved Oct. 25 2011, `https://octoverse.github.com/`

[3] `https://github.com/torvalds/linux`, Retrieved Nov. 24 2017

[4] `https://github.com/google`, Retrieved Nov. 24 2017

[5] `https://github.com/adobe`, Retrieved Nov. 24 2017

[6] `https://github.com/paypal`, Retrieved Nov. 24 2017

1.1 Motivation

1.2 Leading Questions and Goals

Chapter 2
Data Aggregation

The biggest initial task for this thesis was the acquisition of data. The data had to be as extensive as possible, feature a high conjunction between contributers over several repositories to verify a possible connection between those and have realistic meta data. For these requirements two different solutions came up.

The first approach was to design an algorithm for automatic local generation of repositories. The main problem with this solution is, that the visualization and data mining code might be highly optimized for this specific generation algorithm. Real world data is noisy and inconsistent. Thereby the developed solution would have worked on the generated data, but might have failed on real world data.

The second solution was to collect real world data, and thereby collect data from open source projects. I chose Github for this purpose, as it hosts one of the biggest collection of open source projects and provides a great Application programming interface (API) for querying Github's meta data. A problem with this approach is that we don't have access to all important meta data, as for example the full list of members for organizations or the internal team structure of organizations. Another problem is old email addresses, which are not related to any account anymore, because all commits made with this email address are irrefutable. Even though some ground truth is missing, I decided to use this approach as it was still the most promising way to gather as much ground truth and real world noise as possible.

2.1 Structure of the Data

Before we get to the data aggregator, I want to briefly explain the internal Git storage data structure and mechanisms, which are important for the purpose of this thesis [1].

Git, as most programmers know it, is a collection of high level abstraction tools to work with it's underlying file system (FS). The most basic structure in Git is a *blob* object. A *blob* object is a file, which has been added to a Git FS. It is compressed and saved in the `.git/objects` directory under the respective Secure Hash Algorithm 1 (SHA-1) hash of the uncompressed file. The probability of a SHA-1 collision is really low, roughly 10^{-45}, even though Google managed to force a collision in an controlled environment in 2017 [1].

[1] Announcing the first SHA1 collision: `https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html` Retrieved Dec. 16, 2017

To represent a UNIX FS or to simply bundle multiple Git *blob* objects together, Git uses the *tree* object. A *tree* object is a file, which has a SHA-1 hash reference to all underlying *blob* and *tree* objects as well as their names and file permissions. If a *tree* holds a reference to another *tree* it could be interpreted as a subdirectory.

```
100644 blob 11d1ee77f9a23ffcb4afa860dd4b59187a9104e9        .gitignore
040000 tree ac0f5960d9c5f662f18697029eca67fcea09a58c        expose
100644 blob 61b5b2808cc2c8ab21bb9caa7d469e08f875277a        install.sh
040000 tree 8aaf336db307bdcab2f082bd710b31ddb5f9ebd4        thesis
```

Listing 1: *tree* file example.

Now we come to the probably most important Git feature for this thesis; the *commit*. The commit is utilized to provide an exact representation of a state of the repository's files and directories.

```
tree cd7d001b696db430b898b75c633686067e6f0b76
parent c19b969705e5eae0ccca2cde1d8a98be1a1eab4d
author Arne Beer <arne@twobeer.de> 1513434723 +0100
committer Arne Beer <arne@twobeer.de> 1513434723 +0100
```

Listing 2: *commit* file example.

As you can see in listing 2, the *commit* is just another kind of file utilized by Git, which contains some meta data about a repository version:

- The reference to a *tree* object. This is practically the root directory of the Git project
- A reference to one or multiple parent commits, to maintain a version history
- The name and email address of the author
- The name and email address of the committer
- The exact commit and publish Coordinated Universal Time (UTC) timestamp with timezone

Just as the *blob* object the *tree* and *commit* files are also stored in the `.git/objects` directory under their respective hash.

With these simple methods Git manages to create a robust version control system (VCS). Git also provides tools to easily switch between commits of a project (checkout), show the changes between two different commits (diff) and to resolve conflicts between two different commits and merge them together. There are a lot more features available, but those mentioned are the most important for this project.

2.2 Github Meta Data

I decided to use Github as a data source, as it is not only convenient to find Uniform Resource Locators (URLs) for cloning repositories, but also provides some other useful meta data, which can be used to evaluate the precision of any extracted knowledge.

Github offers some features, which are convenient to find repositories a specific user contributed to and to find other contributer which are likely related to each other.

The first feature is *starring*. Every user can *star* a repository to show that he likes a project. The Github *api* doesn't provide a method to get all repositories a user ever contributed to, it only allows to query the repositories owned by a user and the repositories *starred* by a user. With this feature it is possible to get some repositories a user contributed to, even though he doesn't own these repositories, as users tend to star repositories they contributed to .

Find a quote

Another feature is *following*. Every user can *follow* another user to get informed, if they do specific things like creating new repositories or *starring* repositories. As user tend to *follow* friends or colleagues, we can locate repositories of people, which are somehow personally related or work together.

Find a quote

The third feature are *organizations*. An organization is used to host projects under an account which is not necessarily led by a single natural person, but rather supports roles with different permissions and team structures. This feature provides us with some important ground truth, but sadly a lot of information is not visible, as users have to actively opt-in, if they want to be publicly displayed as a member of an organization. Additionally team structures can only be examined, if one is a member of the organization. Despite not knowing all members of an organization, we still get some useful information to estimate the tendency of precision of our knowledge extraction algorithms.

2.3 Data structure

To store and represent the gathered Information I chose a Structured Query Language (SQL) based solution. To be exact I chose PostgreSQL as it provides excellent tools to provide a high consistency, namely check constraints, and a great support for working with times and time zones. The usage of a SQL database and the combination of a Object-Relational Mapping (ORM) allows me to write highly specific queries and

Ordentliche Beschreibung der Datenstruktur und deren Vorteile. SChreiben nach den ersten schwierigen Queries.

2.4 The Aggregator

As mentioned in 2, I decided to get data from Github and wanted to utilize their *Github APIv3* for this purpose. This API is publicly available and can be used by anyone registered on Github. There is a rate limit of 5000 requests per user per hour.

The program I wrote for this thesis is named Gitalizer and features data aggregation, preprocessing, knowledge extraction and visualization. Gitalizer uses a PostgreSQL database for data storage and data consistency checks as described in 2.3. For interaction with the Github API the *pygithub* library is used, which provides a convenient abstraction layer for requests and automatically maps JavaScript Object Notification (JSON) responses to python objects. The data aggregation module of Gitalizer is capable of several scanning methods. In the following we will look at these approaches in detail.

2.4.1 Stand-alone Repository

Gitalizer can scan any git repository from a Secure Shell (SSH) or Hypertext Transfer Protocol (HTTP) URL as long as it has access to it. At first the repository is cloned into a local directory. When the cloning is done the scan process begins. During the scan, we checkout the *HEAD* of the current default branch for this repository and walk down every commit of the Git history. The program saves all available meta data for each commit in its database, namely the emails, timestamps and names of the committer as well as additions and deletions to the project in lines of code.

After this scan we are still missing a lot of information. The unique identifier of an author or committer is their email address, as names may change or can be ambiguous. The problem with the simplicity of Git is that there doesn't exist the concept of an user. Thereby we cannot easily link email addresses to a specific contributer.

2.4.2 Github Repository

To tackle the problems in 2.4.1, I used the Github API to get some of the missing meta data. The general approach is the same as in the previous scan method. The repository is cloned and locally scanned. However, a request to Github is issued every time a new email is found that we do not already have linked to a contributer. Github allows to link multiple email addresses with a single user account and automatically references the respective user in their own API commit representation. With this additional meta data we gain ground truth about the identity of an author or committer.

Anyway this approach does not work, if the user of a commit removed the email used for the commit from his account, or if the user deleted his account. In this case there is nothing that can be done and these commits need to be handled later on in the preprocessing of the data.

2.4.3 Github User

To get all repositories of a specific user, I implemented a new functionality utilizing the Github API. At first several requests are issued to get all repositories of the specified user, as well as all *starred* repositories of this user. For each *starred* repository we check if the user contributed to this repository, which is quite easy as the API provides an endpoint for this query. During the repository exploration, every relevant repository is added to a shared queue, lets call it “repo-queue”, which is then processed by a multiprocessing pool of workers. Each worker process scans a single repository as described in 2.4.2.

2.4.4 Github connected User

For detection and analysis of connections between contributers over multiple repositories, I needed to gather as many repositories of related users as possible. Gitalizer is able to achieve this by not just scanning a single user, but rather scanning the repositories of the specific user, as well as the repositories of all *following* and *followed* users. For this task two different worker pools are utilized. The user pool is initialized with a shared queue, lets call it “user-queue”, of all users we need to look at. This pool simply searches for relevant repositories of a single user and passes them to a second shared queue. The second pool then processes the “repo-queue” as described in 2.4.2.

For organizations it’s nearly the same approach. Initially all repositories, which are owned by the organization, are added to the “repo-queue”. All publicly visible organization members are then added to the “user-queue” and processed as described above.

2.5 Problems

During the development of the data aggregator I experienced a few problems and edge cases which needed to be handled. The earliest and most delaying problem was the rate limit of the Github API. The first version of the aggregator didn’t clone and scan the repository locally, but rather gathered all information from the Github API endpoints. This approach worked well until the aggregator hit the official repository of Nmap, which has about 11.000 commits and took over three hours to scan. Soon I realized that this would severely slow down my research and I then started to continuously minimize the amount API calls. A user scan with remotes of my own Github account led to about 600.000 commits, to provide you with a reference of scale.

After implementing multiprocessing, I managed to hit the rate limit again, as I was now issuing requests with sixteen threads. I needed to implement a wait and retry clause around every single function call or object access, which internally triggered a call to the Github API, to fix this issue, otherwise the worker processes would silently die and the collected data would be incomplete.

Another problem occurred during continuous data mining. Gitalizer only scans repositories until it hits a commit it has already scanned in a previous run. This rule only applies to repositories, which have once been scanned completely. In this scenario I needed to handle edge cases such as force pushing of commits. Force pushes can alter the history of a git repository significantly, which can lead to a split in the Git history and leaves dangling commits. As the complete history of a repository is stored inside the database, I needed to detect a force push and truncate the old commits of the history, which were now outdated and irrelevant.

Grafik zu dieser Problematik

List of Figures

List of Listings

List of Tables

Bibliography

[1] Scott Chacon and 2nd Edition Ben Straub. *Pro Git*. Apress, 2014. ISBN: 978-1484200773.

Eidesstattliche Erklärung

„Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.“

Ort, Datum Unterschrift