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 21, 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
Attack models and their data requirements

This chapter introduces three attacker models and their respective goals. The required data to achieve and evaluate the goal will be listed and explained in the process.

2.1 The Employer

This attack model deals with the scenario of an employer, which wants to monitor their employees. The attacker's motivation is to spot irregularities in working behavior as well as unmotivated or unproductive employees.

Productivity of Employees
Ensure employees produces enough code. For this purpose the changes in lines of code over a specific time span will be evaluated.

Required data:

- Commits of the employer's repositories.
- Commit timestamps
- Additions of each commit
- Deletions of each commit

Compliance of Working Hours
Check if an employee is productive in the defined working hours. This is especially useful to supervise employees, which work remotely.

Required data:

- Commits of employer's repositories.
- Commit timestamps

External Projects during Working Hours Inspect if an employee is working on an external project during working hours. This only works if the employer has access to the external project, for example open source projects.

Required data:

- All commits of any available repository to come into question
- Commit timestamps

Code Quality Between Employees Compare the quality of contributed code between different employees. With this metric the quality of an employee could be measured. To compare the quality we would need an external tool for code analysis.

Required data:

- Commits of the employer's repositories.
- Complete commit patch
- Commit timestamps

2.2 The Individual

This scenario describes a single person, which wants to harm, monitor or gain information about an open source developer.

An example goal of an attacker could be to either stalk the victim, harm him in any way or to manipulate him or one of his acquaintances. The motivation of this attacker is mostly personal and on an emotional level.

Another non emotional attacker could be a robber trying to find the perfect time window to rob a house or the tracking of a high profile target.

A third attacker could be a headhunter which tries to get information about the skills and reliability of a developer.

Sleeping Rhythm and Daily Routine Learn about the persons sleep rhythm and obvious patterns in his daily routine. This attack aims to understand and predict the victim's behaviour.

Required data:

- Victim's commits.
- Commit timestamps.

Personal Relationships to Various Programmers Predict possible friends and the likely grade of strength of their personal relationship. This could be crucial information for further social engineering attacks or to find similar skilled developer for headhunting.

Required data:

- Victim's commits.
- Commit timestamps.

Sick Leave and Holiday Detect breaks in his typical work behaviour, which could represent holiday breaks or sick leave. This attack could give information about whether a developer is at home right now or if he tends to be sick alot.

Required data:

- Victim's commits.
- Commit timestamps.

2.3 The Industrial Spy

This attack model covers the scenario of an external person, which wants to gain as much private or malicious information about a company as possible. The attacker's motivation is either to harm the company, gain an advantage as an competitor or in the stock market or to sell secret information to a third party. This attack vector only works if the targeted company is providing their product or at least parts of their product as open-source software.

Company Employees
The most important target is to detect the company's employees as three other goals for this attacker model depend on this information. Another motivation could be to detect company members for further social engineering attacks or to headhunt the company's employees.

Required data:

- All commits of the company's repositories.
- Commit history graph.
- As much meta data about the company's employees as possible for evaluation.

Employee History
Detect the timespan for which an employee worked at a given company. This could be interesting, as it shows the average employment duration and the employee amount over the history of the company, which could be an indicator of its current financial growth. Social engineering or headhunting could be a motivation here as well.

Required data:

- Company Employees
- Commit timestamps of the company's repositories

Global Workforce Distribution Detect the timezone of all employees and create a global distribution graphic by timezones. This graphic allows you to guess the location of a company's workforce. It is also possible to create this statistic for all contributer, which could show a trend which countries or at least continents are interested the most for the company's product.

Required data:

- Company Employees

- All Commits
- Commit timestamps of the company's repositories

Internal Team Structures Try to predict different teams, the role of each team and the respective team members.

Required data:

- Company Employees.
- Commits of the employer's repositories.
- Commit history graph.

Status of the Product Compare the quality of contributed code between different employees. With this metric the quality of an employee could be measured. To compare the quality we would need an external tool for code analysis.

Überlegen, ob das mit reinkommt. Ist wahrscheinlich ein bisschen weit gefasst.

Required data:

- Commits of the employer's repositories.

Chapter 3
Git

In this chapter the **vcs!** (**vcs!**) *Git* will be introduced. As Git is the foundation of this thesis, I will explain user roles, technologies, internal data representations and other relevant parts of Git.

3.1 Introduction to Git

Git is a tool, which is used to manage different versions of files in a specific directory. Each version of the project is saved as a so called *commit*. Users are able to meticulously specify the files or changes in files that should be added to a commit. It is also capable of showing the exact changes between different commits, which is called a *diff*.

Quote

Git is the currently most popular tool to control a project's code. It enables to work with multiple developers on a single code base, as it provides several different techniques, the *history tree*, the *branch* and the *merge*. The versioning history of Git is internally represented as an directed, non-cyclic, connected graph of commits or a *tree*. The commits act as *nodes* and the connection to their parent commits as *edges*. Every time two edges leave a single node, a new *branch* is created. In Git, every branch has its own name, whereby the main branch is usually named *master*.

In case two different people want to work on the same files, they can each create their own branch on which they can work unimpeded. After they finished and want to add their work to the master branch, they can now *merge* their changes. Git then tries to automatically resolve any conflicts which might have emerged from editing the same lines in a file. If that is not possible, it marks the conflicts and allows the user to manually correct them.

With this methodology it is possible to work with many people or teams on the same project without accidentally overwriting changes of another developer whilst maintaining a clear history of all changes of the project.

Another important feature of Git is the *remote*. A remote is usually located on a distinct server, which is attached to some kind of network, which is accessible by developers. A remote acts as a single source of truth a developer can *push* their changes to or *pull* changes from other developers. Git supports several protocols such as **http!** (**http!**) or **ssh!** (**ssh!**) to connect to the remote and to provide a simple user management layer.

3.2 Git User Roles

There exist two roles in Git, namely the *committer* and the *author*. Every commit in Git contains the email addresses and the names of these two people. The *author* of a commit is the person which actually contributed the changes in the files. The *committer* is the person, which created the git commit. This is important to keep track of the original author of the changes. Lets look at the case of an author contributing code to a project in an email with an attached patch file. If a maintainer of the project now applies the patch file and commits without setting the *author*, the information about the original author would be lost. Although in most cases the *author* and the *committer* are the same person.

Prozentzah auf Basis der gesammelten Daten.

3.3 Internal Representation

Git provides a collection of high level abstraction tools to work with it's underlying **fs!** (**fs!**). In the following I'll explain the structure and management of Git's **fs!**.

The most basic structure in Git is a *blob* object. A *blob* object is any file, which has been added to a Git **fs!**. It is compressed and saved in the `.git/objects` directory under the respective **sha1!** (**sha1!**) hash of the uncompressed file. As follows there exists a blob object for every version of every file of the project.

The **sha1!** hashing for unique file identifier might seem unsafe, but the probability of a **sha1!** collision is really low, roughly 10^{-45}. Lately Google managed to force a collision in an controlled environment in 2017, but it is really unlikely to encounter a collision under normal circumstances. [1]. This feature of **sha1!** hashing become quite important in the design of the database later on.

As mentioned in the introduction**??** Git is used to store the state of a specific directory on any underlying **fs!**. To represent a **fs!** or to simply bundle multiple Git *blob* objects together, Git uses the *tree* object.

A *tree* object is a file, which has a **sha1!** hash reference to all underlying *blob* and *tree* objects as well as their names and file permissions. To represent a subdirectory a *tree* simply holds a reference to another *tree* object.

```
1    100644 blob 11d1ee77f9a23ffcb4afa860dd4b59187a9104e9        .gitignore
2    040000 tree ac0f5960d9c5f662f18697029eca67fcea09a58c        expose
3    100644 blob 61b5b2808cc2c8ab21bb9caa7d469e08f875277a        install.sh
4    040000 tree 8aaf336db307bdcab2f082bd710b31ddb5f9ebd4        thesis
```

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

Listing 1: *tree* file example.

As stated before the *commit* is utilized to provide an exact representation of a state of the repository's files and directories.

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

Listing 2: *commit* file example.

As you can see in listing **??**, 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, which represents the root directory of the 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 **utc!** (**utc!**) timestamps with **utc!** offset for the commit and author date.

Just as the *blob* object the *tree* and *commit* files are also stored in the `.git/objects` directory under their respective hash. With these methods it is now possible to jump between different versions by calling `git checkout` COMMIT_HASH with the version commit hash or to create a *diff* between two commits with `git diff` $\text{COMMIT}_HASH_1 COMMIT_HASH_2$.

Chapter 4
Data from Github

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.

I chose Github for this purpose, as it hosts one of the biggest collection of open source projects and provides a great **api!** (**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.

I decided to use Github as a data source, as it is not only convenient to find **url!**s (**url!**s) 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.

Chapter 5
The Aggregator

As mentioned in **??**, 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.

5.1 Database layout

To store and represent the gathered Information I chose a **sql!** (**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 **orm!** (**orm!**) allows me to write highly specific queries and

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

5.2 Gitalizer

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 **??**. For interaction with the Github **api!** the *pygithub* library is used, which provides a convenient abstraction layer for requests and automatically maps **json!** (**json!**) responses to python objects.

5.3 Methods

The data aggregation module of Gitalizer is capable of several scanning methods. In the following we will look at these approaches in detail.

5.3.1 Stand-alone Repository

Gitalizer can scan any git repository from a **ssh!** or **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.

5.3.2 Github Repository

To tackle the problems in **??**, 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.

5.3.3 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 multi-processing pool of workers. Each worker process scans a single repository as described in **??**.

5.3.4 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 **??**.

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.

5.4 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

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