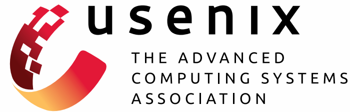


On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities

Santiago Torres-Arias, *New York University;* Anil Kumar Ammula and Reza Curtmola, *New Jersey Institute of Technology;* Justin Cappos, *New York University*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/torres-arias

This paper is included in the Proceedings of the
25th USENIX Security Symposium
August 10–12, 2016 • Austin, TX
ISBN 978-1-931971-32-4

Open access to the Proceedings of the
25th USENIX Security Symposium
is sponsored by USENIX

On omitting commits and committing omissions:

Preventing Git metadata tampering that (re)introduces software vulnerabilities

Santiago Torres-Arias[†] Anil Kumar Ammula[‡], Reza Curtmola[‡] , Justin Cappos[†]
santiago@nyu.edu *aa654@njit.edu* *crix@njit.edu* *jcappos@nyu.edu*

[†]*New York University, Tandon School of Engineering*
[‡]*Department of Computer Science, New Jersey Institute of Technology*

Abstract

Metadata manipulation attacks represent a new threat class directed against Version Control Systems, such as the popular Git. This type of attack provides inconsistent views of a repository state to different developers, and deceives them into performing unintended operations with often negative consequences. These include omitting security patches, merging untested code into a production branch, and even inadvertently installing software containing known vulnerabilities. To make matters worse, the attacks are subtle by nature and leave no trace after being executed.

We propose a defense scheme that mitigates these attacks by maintaining a cryptographically-signed log of relevant developer actions. By documenting the state of the repository at a particular time when an action is taken, developers are given a shared history, so irregularities are easily detected. Our prototype implementation of the scheme can be deployed immediately as it is backwards compatible and preserves current workflows and use cases for Git users. An evaluation shows that the defense adds a modest overhead while offering significantly stronger security. We performed responsible disclosure of the attacks and are working with the Git community to fix these issues in an upcoming version of Git.

1 Introduction

A Version Control System (VCS) is a crucial component of any large software development project, presenting to developers fundamental features that aid in the improvement and maintenance of a project's codebase. These features include allowing multiple developers to collaboratively create and modify software, the ability to roll back to previous versions of the project if needed, and a documentation of all actions, thus tying changes in files to their authors. In this manner, the VCS maintains a progressive history of a project and helps ensure the integrity of the software.

Unfortunately, attackers often break into projects' VCSs and modify the source code to compromise hosts who install this software. When this happens, an attacker can introduce vulnerable changes by adding (e.g., adding a backdoor), or removing certain elements from a project's history (e.g., a security patch) if he or she acquires write access to the repository. By doing this, attackers are usually able to compromise a large number of hosts at once [42, 27, 13, 21, 15, 4, 45, 18, 44]. For example, the Free Software Foundation's repository was controlled by hackers for more than two months, serving potentially backdoored versions of GNU software to millions of users [16].

The existing security measures on VCSs, such as commit signing and push certificates [19, 2], provide limited protection. While these mechanisms prevent an attacker from tampering with the contents of a file, they do not prevent an attacker from modifying the repository's metadata. Hence, these defenses fail to protect against many impactful attacks.

In this work, we reveal several new types of attacks against Git, a popular VCS. We collectively call these attacks *metadata manipulation attacks* in which Git metadata is modified to provide inconsistent and incorrect views of the state of a repository to developers. These attacks can be thought of as *reconcilable fork attacks* because the attacker can cause a developer's version of the repository to be inconsistent just for a finite window of time — only long enough to trick a developer into commiting the wrong action — and leave no trace of the attack behind.

The impact of an attack of this nature can be substantial. By modifying the right metadata, an attacker can remove security patches, merge experimental code into a production branch, withhold changes from certain users before a release, or trick users and tools into installing a different version than the one requested to the VCS. To make matters worse, the attacker only requires a few resources to achieve his or her malicious goals.

We have submitted a vulnerability disclosure to CERT and the GitHub security team describing the following scenario: an attacker capable of performing a man-in-the-middle attack between a GitHub [3] server and a developer using pip to install Django (a popular website framework) can trick the developer into installing a vulnerable version simply by replacing one metadata file with another. Even though Git verifies that the signature in Git objects is correct, it has no mechanism to ensure it has retrieved the correct object. This type of attack enables a malicious party to strike any system that can retrieve packages from Git repositories for installation, including Node's NPM [22], Python's pip [11], Apache Maven [34], Rust's cargo [35], and OCaml's OPAM [33]. As such, it could potentially affecting hundreds of thousands of client devices.

To mitigate metadata manipulation attacks, we designed and implemented a client-only, backwards-compatible solution that introduces only minimal overhead. By storing signed reference state and developer information on the server, multiple developers are able to verify and share the state of the repository at all times. When our mechanism is in place, Git metadata manipulation attacks are detected. We have presented these issues to the Git developer community and prepared patches — some of which are already integrated into Git — to fix these issues in upcoming versions of Git.

In summary, we make the following contributions:

- We identify and describe metadata manipulation attacks, a new class of attacks against Git. We show these attacks can have a significant practical impact on Git repositories.
- We design a defense scheme to combat metadata manipulation attacks by having Git developers share their perception of the repository state with their peers through a signed log that captures their history of operations.
- We implement the defense scheme and study its efficiency. An evaluation shows that it incurs a smaller storage overhead than push certificates, one of Git's security mechanisms. If our solution is integrated in Git, the network communication and end-to-end delay overhead should be negligible. Our solution does not require server side software changes and can be used today with existing Git hosting solutions, such as GitHub, GitLab, or Bitbucket.

2 Background and related work

2.1 Overview of Git

In order to understand how Git metadata manipulation attacks take place, we must first define Git-specific terminology, as well as some usage models of the tool itself.

Git is a distributed VCS that aids in the development of software projects by giving each user a local copy of the relevant development history, and by propagating changes made by developers (or their history) between such repositories. Essential to the version history of code committed to a Git repository are `commit` objects, which contain metadata about who committed the code, when it was committed, pointers to the previous commit object, (the `parent` commit) and pointers to the objects (e.g., a file) that contain the actual committed code.

Branches serve as "pointers" to specific commit objects, and to the development history that preceded each commit. They are often used to provide conceptual separation of different histories. For example, a branch titled "update-hash-method" will only contain objects that modify the hash method used in a project. When a developer adds a new commit to the commit chain pointed to by a branch, the branch is moved forward.

Inside Git, branches are implemented using "reference" files, that only contain the SHA1 hash of a target commit. The same format is used for Git tags, which are meant to point to a static point in the project's history. Both tags and branches live in the `.git/refs` folder.

Git users `commit` changes to their local repositories, and employ three main commands to propagate changes between repositories: `fetch`, to retrieve commits by other developers from a remote repository; `merge`, to merge two changesets into a single history; and `push`, to send local commits from a local repository to a remote repository. Other common commands may consist of two or more of these commands performed in conjunction (e.g., *pull* is both a `fetch` and a `merge`). Consider the following example:

Alice is working on a popular software project and is using Git to track and develop her application. Alice will probably host a "blessed" copy of her repository in one provider (e.g., GitHub or Gitlab) for everyone to clone, and from which the application will eventually be built. In her computer, she will keep a *clone* (or copy) of the remote repository to work on a new feature. To work on this feature, she will create a new **branch**, `#5-handle-unicode-filenames` that will diverge from the `master` branch from now on. As she modifies files and updates the codebase, she commits – locally – and the updates will be added to the new branch in her local clone. Once Alice is done adding the feature, she will *push* her local commits to the remote server and request a colleague to review and *merge* her changes into the master branch. When the changes are merged, Alice's commits will become part of the master history and, on the next release cycle, they will be shipped in the new version of the software.

2.1.1 Git security features

To ensure the integrity of the repository's history, Git incorporates several security features that provide a basic defense layer:

- Each commit object contains a cryptographic hash of its parent commit. In addition, the name of the file that contains the commit object is the cryptographic hash of the file's contents. This creates a hash chain between commits and ensures that the history of commits cannot be altered arbitrarily without being detected.
- Users have the option to cryptographically sign a commit (a digital signature is added to the commit object) using a GPG key. This allows an auditor to unequivocally identify the user who committed code and prevents users from repudiating their commits.
- A signed certificate of the references can be pushed to a remote repository. This "push certificate" solution addresses man-in-the-middle attacks where the user and a well-behaving server can vouch for the existence of a push operation.

2.2 Related work

VCS Security. Wheeler [39] provides an overview of security issues related to software configuration management (SCM) tools. He puts forth a set of security requirements, presents several threat models (including malicious developers and compromised repositories), and enumerates solutions to address these threats. Gerwitz [17] provides a detailed description of creating and verifying Git signed commits. Signing commits allows the user to detect modifications of committed data. Git incorporates protection mechanisms, such as commit signing and commit hash chaining. Unfortunately, they do not prevent the attacks we introduce in this work.

There have been proposals to protect sensitive data from hostile servers by incorporating secrecy into both centralized and distributed version control systems [1, 29]. Shirey et al. [32] analyzes the performance tradeoffs of two open source Git encryption implementations. Secrecy from the server might be desirable in certain scenarios, but it is orthogonal to our goals in this work.

The "push certificate" mechanism, introduced in version 2.2.0 of Git, allows a user to digitally sign the reference that points to a pushed object. However, push certificates do not protect against most of the attacks we describe in this work. Furthermore, push certificates were designed for out-of-band auditing (*i.e.* they are not integrated into the usual workflow of Git and need to be fetched and verified by a trusted third party using out-of-band mechanisms). As a result, push certificates are rarely used in practice.

Fork Consistency. A problem that could arise in remote storage used for collaborative purposes is when the untrusted storage server hides updates performed by one group of users from another. In other words, the server equivocates and presents different views of the history of operations to different groups of users. The *fork consistency* property seeks to address this attack by forcing a server that has forked two groups in this way to continue this deception. Otherwise, the attack will be detected as soon as one group sees an operation performed by the other group after the moment the fork occurred.

SUNDR [26] provides fork consistency for a network file system that stores data on untrusted servers. In SUNDR, users sign statements about the complete state of all the files and exchange these statements through the untrusted server. SPORC [14] is a framework for building collaborative applications with untrusted servers that achieves fork* consistency (*i.e.*, a weaker variant of fork consistency). Our solution seeks to achieve a similar property and shares similarities with SUNDR in that Git users leverage the actual Git repository to create and share signed statements about the state of the repository. However, the intricacies and usage model of a VCS system like Git impose a different set of constraints.

Other work, such as Depot [28], focuses on recovering from forks in an automatic fashion (i.e., not only detecting forks, but also repairing after they are detected). Our focus is on detecting the metadata manipulation attacks, after which the affected users can perform a manual rollback procedure to a safe point.

Caelus [25] seeks to provide the same declared history of operations to all clients of a distributed key-value cloud store. Caelus assumes that no external communication channel exists between clients, and requires them to periodically attest to the order and timing of operations by writing a signed statement to the cloud every few seconds. The attestation schedule must be pre-defined and must be known to all clients. Our setting is different, since Git developers usually communicate through multiple channels; moreover, a typical team of Git developers cannot be expected to conform to such an attestation policy in practice.

3 Threat model and security guarantees

We make the following assumptions about the threat model our scheme is designed to protect against:

- Developers use the existing Git signing mechanisms whenever performing an operation in Git to stop an attacker from tampering with files.
- An attacker cannot compromise a developer's key or get other developers to accept that a key controlled by an attacker belongs to a legitimate developer. Alternatively, should an attacker control such a key (e.g., an insider attack), he or she may not

want to have an attack attributed to him- or herself and would thus be unwilling to sign data they have tampered with using their key.

- The attacker can read and modify any files on the repository, either directly (*i.e.* a compromised repository or a malicious developer) or indirectly (*i.e.*, through MITM attacks and using Git's interface to trick honest users into doing it).
- The attacker does not want to alert developers that an attack has occurred. This may lead to out-of-band mechanisms to validate the attacked repository [30].

This threat model covers a few common attack scenarios. First of all, an attacker could have compromised a software repository, an unfortunately common occurrence [42, 27, 13, 21, 15, 4, 45, 18, 44, 16]. Even if the repository is not compromised, an attacker could act as a man-in-the-middle by intercepting traffic destined for the repository (e.g., by forging SSL certificates [23, 31, 8, 37, 43, 7, 41, 6, 38]). However, an attacker is not limited to these strategies. As we will show later, a malicious developer can perform many of the same attacks *without using their signing key*. This means that it is feasible for a developer inside an organization to launch these attacks and not be detected.

Note that in all cases, the developers have known signing keys to commit, push, and verify information.

3.1 Security guarantees

Answering to this threat model, the goal of a successful defensive system should be to enforce the following:

- **Prevent modification of committed data**: If a file is committed, an attacker should not be able to modify the file's contents without being detected.
- **Ensure consistent repository state**: All developers using a repository should see the same state. The repository should not be able to equivocate and provide different commits to different developers.
- **Ensure repository state freshness:** The repository should provide the latest commits to each developer.

As we will show later, Git's existing security mechanisms fail to handle the last two properties. The existing signing mechanism for Git does enable developers to detect modification of committed data, because the changed data will not be correctly signed. However, due to weaknesses in handling the other properties, an attacker can omit security patches, merge experimental features into production, or serve versions of software with known vulnerabilities.

An attacker is successful if he or she is able to break any of these properties without being detected by the developers. So, an attacker who controls the repository could block a developer from pushing an update by pretending the repository is offline. However, since the developer receives an error, it is obvious that an attack is occurring and therefore is easy to detect. Similarly, this also precludes *irreconciliable fork attacks* where two sets of developers must be permanently segregated from that point forward. Since developers typically communicate through multiple channels, such as issue trackers, email, and task management software, it will quickly become apparent that fixes are not being merged into the master branch. (Most projects have a tightly integrated team, usually a single person, who integrates changes into the master branch, which further ensures this attack will be caught.) For these reasons, we do not focus on attacks that involve a trivial denial of service or an irreconcilable fork because they are easy to detect in practice.

4 Metadata manipulation attacks

Even when developers use Git commit signing, there is still a substantial attack surface. We have identified a new class of attacks that involve manipulation of Git metadata stored in the `.git/refs` directory of each repository. We emphasize that, unlike Git commits that can be cryptographically signed, there are no mechanisms in Git to protect this metadata. As such, the metadata can be tampered with to cause developers to perceive different states of the repository, which can coerce or trick them into performing unintended operations in the repository. We also note that a solution that simply requires users to sign Git metadata has serious limitations (as described in Sec. 5.2).

Unlike many systems where equivocation is likely to be noticed immediately by participants, Git's use of branches hides different views of the repository from developers. In many development environments, developers only have copies of branches that they are working on stored locally on their system, which makes it easy for a malicious repository to equivocate and show different views to different developers.

In Git, a branch is represented by a file that contains the SHA1 checksum of a commit object (under benign circumstances, this object is the latest commit on that branch). We will refer to such files as *branch references*. All the branch references are stored in the directory `.git/refs/heads/`, with the name of the branch as the filename. For example, a branch "hotfix" is represented by the file `.git/refs/heads/hotfix`.

We discovered that it is straightforward for an attacker to manipulate information about branches by simply changing contents in a reference file to point to any

other commit object. *Modifying the branch reference can be easily performed with a text editor and requires no sophistication.* Specifically, we show three approaches to achieve this, all of them being captured by our adversarial model. *First*, an attacker who has compromised a Git repository and has write access to it, can directly modify the metadata files. *Second*, an attacker can perform an MITM attack by temporarily redirecting a victim's traffic to a fake repository serving tampered metadata, and then reestablishing traffic so the victim propagates the vulnerable changes to the genuine repository (in Appendix A, we describe a proof-of-concept attack against GitHub based on this approach). *Third*, a malicious developer can take advantage of the fact that Git metadata is synchronized between local and remote repositories. The developer manipulates the Git metadata in her local repository, which is then propagated to the (main) remote repository.

It is also possible to extend these attacks for Git tags. Although a Git tag is technically a Git tag object that can be signed the same way as a commit object, an attacker can target the *reference* pointing to a tag. Tag references are stored in the directory `.git/refs/tags/` and work similarly to branch references, in that they are primarily a file containing the SHA1 of a Git tag object that points to a Git commit object. Although Git tags are conceptually different — they only represent a fixed point (e.g., a major release version) in the projects history — they can be exploited in the same way, because Git has no mechanism to protect either branch or tag references.

We have validated the attacks against a standard Git server and also the GitHub, GitLab and other popular Git hosting services.

Based on their effect on the state of the repository, we identify three types of metadata manipulation attacks:

- **Teleport Attacks:** These attacks modify a Git reference so that it points to an arbitrary object, different from the one originally intended. The reference can be a branch reference or a tag reference.

- **Rollback Attacks:** These attacks modify a Git branch reference so that it points to an older commit object from the same branch, thus providing clients with a view in which one or more of the latest branch commits are missing.

- **Deletion Attacks:** These attacks remove branch or tag references, which in turns leads to the complete removal of an entire branch, or removal of an entire release referred to by a tag.

We use the following setup to present the details of these attacks. A Git server is hosting the main repository and several developers who have their own local repositories have permission to fetch/push from/to any branch of the main repository, including the master branch. For commit objects, we use a naming convention that captures the temporal ordering of the commits. For example, if a repository has commits C0, C1, C2, this means that they were committed in the order C0, C1, C2.

4.1 Teleport attacks

We identified two teleport attacks: *branch teleport* and *tag teleport* attacks.

Branch Teleport Attacks. These attacks modify the branch reference so that it points to an arbitrary commit object on a different branch. Although we illustrate the attacks for the master branch, they are applicable to any branch, since none of the branch reference metadata is protected.

Fig. 1(a) shows the initial state of the main Git repository, which contains two branches, "master" and "feature." The local repository of developer 1 is in the same state as shown in Fig. 1(a). The "feature" branch implements a new feature and contains one commit, C2. The code in C2 corresponds to an unstable, potentially-vulnerable version that needs to be tested more thoroughly before being integrated into the master branch. Commit C1 is the head of the master branch. This means that the file `.git/refs/heads/master` contains the SHA1 hash of the C1 commit object.

After developer 2 pulls from the master branch of the main repository (Fig. 1(b)), the attacker changes the master branch to point to commit C2 (Fig. 1(c). The attacker does this by simply changing the contents of the file `.git/refs/heads/master` to the SHA1 hash of the C2 commit. Any developer who clones the repository or fetches from the master branch at this point in time will be provided with the incorrect repository state, as shown in Fig. 1(c). For example, developer 2, who committed C3 into his local repository (Fig. 1(d)), now wants to push this change to the main repository. Developer 2 is notified that there were changes on the master branch since his last fetch, and needs to pull these changes. As a result, a merge commit C4 occurs between C3 and C2 in the local repository of developer 2, as shown in Fig. 1(e). The main repository looks like Fig 1(e) after developer 2 pushes his changes. If developer 1 then pulls changes from the main repository, all three repositories will appear like Fig 1(e).

Normally, the master branch should contain software that was thoroughly tested and properly audited. However, in this incorrect history, the master branch incorporates commit C2, which was in a experimental feature branch and may contain bugs. The attacker tricked a developer into performing an action that was never in-

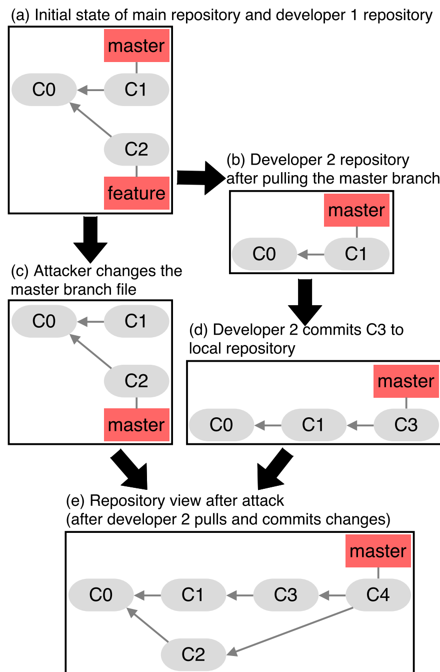


Figure 1: The Branch Teleport attack

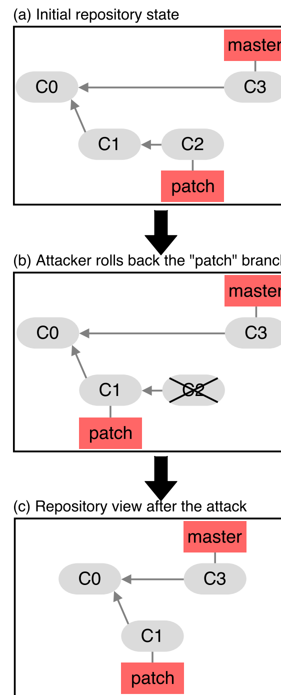


Figure 2: The Branch Rollback attack

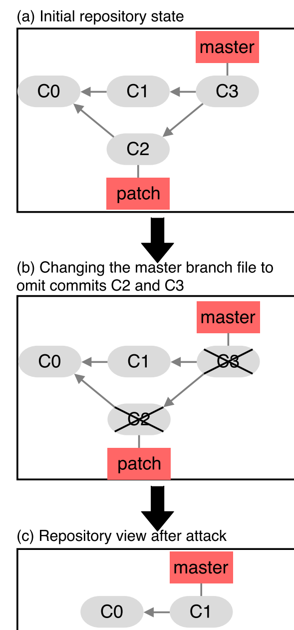


Figure 3: The Global Rollback attack

tended, and none of the two developers are aware that the attack took place.

Tag Teleport Attacks. These attacks modify a tag reference so that it points to an arbitrary tag object. Surprisingly, a tag reference can also be made to point to a commit object, and Git commands will still work.

One can verify whether a tag is both signed and a valid tag object by using the `git tag --verify` command. However, if an attacker were to modify a tag reference to point to an older tag (e.g., if the tag for release 1.1 is replaced by the tag for the vulnerable release 1.0), the verification command is successful.

Modifying tag metadata could be especially impactful for automated systems that rely on tags to build/test and release versions of software [36, 20, 10, 12]. Furthermore, package managers such as Python's pip, Ruby's RubyGEMS, and Node's NPM, among many others support the installation of software from public Git repositories and tags. Finally, Git submodules are also vulnerable, as they automatically track a tag (or branch). If a build dependency is included in a project as a part of the submodule, a package might be vulnerable via an underlying library.

4.2 Rollback attacks

These attacks modify a Git branch reference so that it points to an older commit object from the same branch. This gives clients a view in which one or more of the latest branch commits are missing. The attacker can cause commits to be missing on a permanent or on a temporary basis.

4.2.1 Permanent rollback attacks

Based on the nature of the commits removed, we separate permanent rollback attacks in two groups: Branch Rollback attacks and Global Rollback attacks.

Branch Rollback Attacks. Consider the repository shown in Fig. 2(a), in which the order of the commits is C0, C1, C2, C3. Commits C0 and C3 are in the master branch, and commits C1 and C2 are security patches in a "patch" branch. The attacker rolls back the patch branch by making the head of such branch point to commit C1, as shown in Fig. 2(b). This can be done by simply replacing the contents of the file `.git/refs/heads/patch` with the SHA1 hash of the C1 commit. As a result, all developers that pull from the main repository after this attack will see the state shown in Fig. 2(c), in which commit C2 (that contains a security patch) has been omitted.

Note that the attack can also be used to omit commits on any branch, including commits in the master branch.

Global Rollback Attacks. As opposed to a Branch Rollback attack, which removes commits that happened prior to one that remains visible, in a Global Rollback attack, no commits remain visible after the commits that are removed. In other words, the attacker removes one or more commits that were added last to the repository.

Consider the initial state of a Git repository as illustrated in Fig. 3(a), in which C2 is a commit that fixes a security bug and has been merged into the master

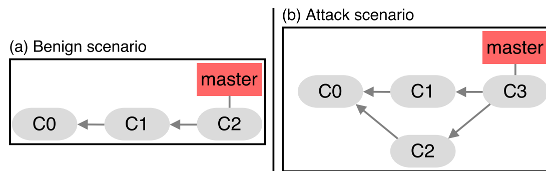


Figure 4: The Effort Duplication attack

branch. The file `.git/refs/heads/master` contains the SHA1 hash of the C3 commit object.

By simply changing the contents of the file `.git/refs/heads/master` to the SHA1 hash of the C1 commit, the attacker forges a state in which the repository contains the history of commits depicted in Fig. 3(b). This effectively removes commits C2 and C3 from the project's history, and a developer who now clones the project will get a history of commits as shown in Fig. 3(c). This incorrect history does not contain the commit C2 that fixed the security bug.

Note that the Global Rollback attack *removed the latest two commits from the repository*. This is different than the effect of a Branch Rollback attack which *removes one or more commits that happened before a commit that remains visible*.

4.2.2 Temporary rollback attacks

Effort Duplication Attacks. The Effort Duplication attack is a variation of the Global Rollback attack, in which the attacker temporarily removes commits from the repository. This might cause developers to unknowingly duplicate coding efforts that exist in the removed commits.

Consider a main Git repository with just a master branch which contains only one commit C0. Two developers D1 and D2 have pulled from the main repository, so their local repositories also contain C0. After the following sequence of actions by D1 and D2, the repository should look as shown in Fig. 4(a):

1. D1 commits C1 to her local repository & pushes to the main repository.
2. D2 pulls from the main repository.
3. D2 commits C2 to her local repository & pushes to the main repository.

However, when D2 pulls in step 2, the attacker can temporarily withhold commit C1, keeping D2 unaware of the changes in C1. As a result, D2 works on changes that already exist in C1. The following attack scenario results in a repository shown in Fig. 4(b):

1. D1 commits C1 to her local repository & pushes to the main repository.
2. D2 pulls from the main repository, but the attacker withholds C1. Thus, D2 thinks there are no changes.
3. D2 makes changes on top of C0 and commits these changes in her local repository as commit C2. C2 duplicates (some or all of) D1's coding effort in C1.
4. D2 tries to push changes to the main repository. This time, the attacker presents C1 to D2 (these are the changes that were withheld in step 2). Thus, D2 has to first pull changes before pushing.
5. D2 pulls changes from the main repository, and this results in a merge commit C3 between C1 and C2. As part of the merge, the developer has to solve any merge conflicts that appear from the code duplication between C1 and C2.

In this case, D2 re-did a lot of D1's work because D1's commit C1 was withheld by the attacker. Note that unlike a Global Rollback attack, in which commits are removed permanently from the repository, in the Effort Duplication attack commits are just removed temporarily. This is a more subtle attack, since the final state of the repository is the same for both the benign and attack cases. The effect of applying commits C1 and C2 in Fig. 4(a) on the files in the repository is the same as applying commits C1, C2, C3 in Fig. 4(b). However, D2 unknowingly (and unnecessarily) duplicated D1's coding effort, which may have negative economic consequences. Adding to this, an attacker can slow down developers of a specific project (e.g., a competitor's project) by delivering previously-withheld changes to them when they will cause merge conflicts and hamper their development progress.

4.3 Reference deletion attacks

Since the branch metadata is not protected, the attacker can hide an entire branch from the repository by removing a branch reference. Similarly, since the tag metadata is not protected, the attacker can remove a tag reference in order to hide a release from the repository history.

When an attacker performs a reference deletion attack, only the users who previously held a copy of the reference will be able to know of its existence. If this is not the case, a developer would be oblivious of the fact that other developers have worked on the deleted branch (similar to a fork attack), or be tricked into retrieving another version if the target tag is not available. Furthermore, some projects track work in progress by tying branch names to numbers in their issue tracker [9], so two developers could be tricked into working on the same issue by hiding a branch (similar to an effort duplication attack).

4.4 Summary of attacks

Metadata manipulation attacks may lead to inconsistent and incorrect views of the repository and also to corruption and loss of data. Ultimately, this will lead to merge conflicts, omission of bug fixes, merging experimental code into a production branch, or withholding changes from certain users before a release. All of these are problems that can impact the security and stability of

the system as a whole. Table 1 summarizes the attacks impact.

Attack	Impact
Branch Teleport	Buggy code inclusion
Branch Rollback	Critical code omission
Global Rollback	Critical code omission
Effort Duplication	Coding effort duplicated
Tag Rollback	Older version retrieved

Table 1: Impact of metadata manipulation attacks.

5 Defense framework

5.1 Design goals for a defense scheme

We designed our defense scheme against metadata manipulation attacks with the following goals in mind:

*Design Goal 1 (**DG1**):* Achieve the security goals stated in Sec. 3.1. That is, prevent modification of committed data, ensure a consistent repository state, and ensure repository state freshness.

*Design Goal 2 (**DG2**):* Preserve (as much as possible) current workflows and actions that are commonly used by developers, in order to facilitate a seamless adoption.

*Design Goal 3 (**DG3**):* Maintain compatibility with existing Git implementations. For example, Git has limited functionality when dealing with concurrency issues in a multi-user setting: it only allows atomic push of multiple branches and tags after version 2.4. Following Git's design philosophy, backwards compatibility is paramount; a server running the latest Git version (*i.e.*, 2.9.0) can be cloned by a client with version 1.7.

5.2 Why binding references with Git objects is not enough

Adding reference information (i.e., branch and tag names) inside the commit object might seem like a sufficient defense against metadata manipulation attacks. This would bind a commit to a reference and prevent an attacker from claiming that a commit object referred to in a reference belongs somewhere else.

Unfortunately, this simple approach has important drawbacks. It does not meet our *DG1* because it does not defend against rollback and effort duplication attacks. Furthermore, adding new reference information in a commit object requires updating an existing commit object. When this happens, the SHA-1 hash of the commit object will change, and the change will propagate to all new objects in the history. In other words, when a new branch is created and bound to a commit far earlier in the history, all commit objects need to be rewritten and, thus, sent back to the remote repository, which could add substantial computational and network overhead.

5.3 Our defense scheme

The fundamental cause of metadata manipulation attacks is that the server can respond to users' `fetches` with an incorrect state and history of the main repository that they cannot verify. For example, the server can falsely claim that a branch points to a commit that was never on that branch or to a commit that was the location of that branch in an earlier version. Or, the server can falsely claim that the reference of a tag object points to an older tag.

In order to stop the server from falsely claiming an incorrect state of the repository, we propose that every Git user must include additional information vouching for their perceived repository state during a push or a fetch operation. To achieve this, we include two pieces of additional information on the repository:

- First, upon every push, users must append a *push entry* to a *Reference State Log (RSL)* (Sec. 5.3.1). By validating new entries in this log with each push and fetch operation, we can prevent teleport, permanent rollback, and deletion attacks.

- Second, when a Git user performs a fetch operation and receives a new version of files from the repository, the user places a random value into a *fetch nonce bag* (Sec. 5.3.2). If the Git user does not receive file updates when fetching, the user replaces her value in the bag with a new one. The bag serves to protect against temporary rollback attacks.

During our descriptions, we assume that a trusted keychain is distributed among all developers along with the RSL. There are tools available to automate this process [24, 5], and the RSL itself can also be used to distribute trust (we elaborate more on this in Sec 6.1).

5.3.1 The Reference State Log (RSL)

For a developer to prevent the server from equivocating on the location of the references, the developer will sign a *push entry*, vouching for the location of the references at the time of a push. To do this, she must execute the Secure_push procedure, which has the following steps:

First, the remote RSL is retrieved, validated, and checked for the presence of new push entries (lines 3-11). If the RSL is valid and no push entries were added, a new RSL push entry is created (lines 13-14). The newly created entry will contain: (1) the new location of the reference being pushed; (2) the nonces from the fetch nonce bag; (3) a hash of the previous push entry; and (4) the developer's signature over the newly created push entry. The newly created entry is then appended to the RSL (line 16), and the nonce_bag is cleared (line 15).

Once this is done, the remote RSL must be updated and local changes must be pushed to the remote reposi-

PROCEDURE: Secure_push
Input: LocalRSL; related_commits; pushed reference X
Output: *result: (success/fail/invalid)*

```
1:  repeat
2:    result ← fail
3:    (RemoteRSL, nonce_bag) =
          Retrieve_RSL_and_nonce_bag_from_remote_repo
4:    if (RSL_Validate(RemoteRSL, nonce_bag) == false)
      then
5:      // Retrieved RemoteRSL is invalid
6:      // Must take necessary actions!
7:      return invalid
8:    if (new push entries for reference X in RemoteRSL) then
9:      // Remote repository contains changes
10:     // User must fetch changes and then retry
11:     return fail
12:   else
13:     prev_hash = hash_last_push_entry(RemoteRSL)
14:     new_RSL_Entry = create_push_Entry(prev_hash,
                                          nonce_bag, X)
15:     nonce_bag.clear()
16:     RemoteRSL.addEntry(new_RSL_entry)
17:     result = Store_in_remote_repo(RemoteRSL,
                                      nonce_bag)
18:     if (result == success) then
19:       // The remote RSL has no new entries
20:       push related_commits
21:       LocalRSL = RemoteRSL
22:       return success
23: until (result == success)
```

tory (lines 17-20). Notice that these steps are performed under a loop, because other developers might be pushing, which is not an atomic operation in older versions of Git (this is required to meet *DG3*).

Depending on the result of the Secure_push procedure, a developer's actions correspond to the following:

- `success`: the push is successful. No further actions are required from the user (line 22).
- `fail`: the push fails because there are changes in the remote repository that must first be fetched and merged locally before the user's changes can be pushed (line 11).
- `invalid`: the RSL validation has failed. The algorithm detects a potential attack and notifies the user, who must then take appropriate measures (line 7).

Note that these actions mirror a user's actions in the case of a regular Git push operation, as suggested by *DG2*. By doing this, we effectively follow the existing Git workflows while providing better security guarantees at the same time.

5.3.2 The Nonce Bag

When retrieving the changes from a remote repository, a developer must also record her perceived state of the repository. Our scheme requires that all the user fetches be recorded in the form of a fetch *nonce bag*, *i.e.*, an unordered list of nonces. Each nonce is a random number that corresponds to a fetch from the main repository. Every time a user fetches from the main repository, she updates the nonce bag. If the user has not fetched since the last push, then she generates a new nonce and adds it to the nonce bag; otherwise, the user replaces her nonce in the nonce bag with a new nonce.

Each nonce in the nonce bag serves as a proof that a user was presented a certain RSL, preventing the server from executing an Effort Duplication attack and providing repository freshness as per *DG1*. To fetch the changes from the remote repository, a developer must execute the Secure_fetch procedure.

The first steps of the Secure_fetch procedure consist of retrieving the remote RSL, performing a regular `git fetch`, and ensuring that the latest push entry in the RSL points to a valid object in the newly-fetched reference (lines 4-11). Note that this check is performed inside a loop because push operations are not atomic in older versions of Git (lines 2-12). A user only needs to retrieve the entries which are new in the remote RSL and are not present in the local version of the RSL.

If this check is successful, the nonce bag must be updated and stored at the remote repository (lines 14-20). Note that all these steps are also in a loop because other developers might update the RSL or the nonce bag since it was last retrieved (lines 1-21).

Finally, the RSL is further validated for consistency (line 22), and the local RSL is updated. We chose to validate the RSL at the end of Secure_fetch and outside of the loop in order to optimize for the most common case. Once Secure_fetch is successfully executed, a developer can be confident that the state of the repository she fetched is consistent with her peers. Otherwise, the user could be the victim of one of the attacks in Sec. 4.

5.3.3 RSL validation

The RSL_Validate routine is used in Secure_push and Secure_fetch to ensure the presented RSL is valid. The aim of this routine is to check that push entries in a given RSL are correctly linked to each other, that they are signed by trusted developers, and that nonces corresponding to a user's fetches are correctly incorporated into the RSL.

First, the procedure checks that the nonce corresponding to the user's last fetch appears either in the nonce bag or was incorporated into the right push entry (*i.e.*, the first new push entry of the remote RSL) (lines 1-2). The algorithm then checks if the new push RSL entries from the

PROCEDURE: Secure_fetch
Input: reference X to be fetched
Output: *result: (success/invalid)*

```
1:  repeat
2:      store_success ← false
3:      repeat
4:          (RemoteRSL, nonce_bag) =
                Retrieve_RSL_and_nonce_bag_from_remote_repo()
5:          fetch_success ← false
6:          // This is a regular "Git fetch" command.
7:          // Branch X's reference is copied to FETCH_HEAD
8:          fetch reference X
9:          C ← RemoteRSL.latestPush(X).refPointer
10:         if (C == FETCH_HEAD) then
11:             fetch_success ← true
12:     until (fetch_success == true)
13:     // Update the nonce bag
14:     if NONCE in nonce_bag then
15:         nonce_bag.remove(NONCE)
16:     save_random_nonce_locally(NONCE)
17:     nonce_bag.add(NONCE)
18:     // Storing the nonce bag at the remote repository
19:     //  might fail due to concurrency issues
20:     store_success = Store_in_remote_repo(nonce_bag)
21: until (store_success == true)
22: if (RSL_Validate(RemoteRSL, nonce_bag) == false)
    then
23:     // Retrieved RemoteRSL is invalid
24:     // Must take necessary actions!
25:     return invalid
26: else
27:     LocalRSL = RemoteRSL
28:     return success
```

PROCEDURE: RSL_Validate
Input: LocalRSL (RSL in the local repository); RemoteRSL; nonce_bag
Output: `true` or `false`

```
1:  if (NONCE not in nonce_bag) and (NONCE not in RemoteRSL.push_after(LocalRSL) then
2:      return false
3:  // Verify that the ensuing entries are valid
4:  prev_hash = hash_last_push_entry(LocalRSL)
5:  for new_push_entry in RemoteRSL do
6:      if new_push_entry.prev_hash != prev_hash then
7:          // The RSL entries are not linked correctly
8:          return false
9:      prev_hash = hash(new_push_entry)
10: if verify_signature(RemoteRSL.latest_push) == false then
11:     // this RSL is not signed by a trusted developer
12:     return false
13: return true
```

remote RSL are correctly linked to each other and that the first new remote push entry is correctly linked to the last push entry of the local RSL (the check is based on the prev_hash field) (lines 5-9). Finally, the signature on the last RSL push entry is verified to ensure it was signed by a trusted developer; since all RSL entries are correctly linked, only the last entry signature needs to be verified.

How to handle misbehavior? If the RSL validation fails due to a misbehaving server, the user should compare the local RSL with the remote RSL retrieved from the remote repository and determine a safe point up to which the two are consistent. The users will then manually roll back the local and remote repositories to that safe point, and decide whether or not to continue trusting the remote repository.

6 Discussion

6.1 Trust and revoke entries

Developers' keys may be distributed using *trust/revoke RSL entries*. To use these entries, the repository is initialized with an authoritative root of trust (usually a core developer) who will add further entries of new developers in the group. Once developers' public keys are added to the RSL, they are allowed to add other trust entries.

A trust entry contains information about the new developer (i.e., username and email), her public key, a hash of the previous push entry and a signature of the entry by a trusted developer. Revocation entries are similar in that they contain the key-id of the untrusted developer, the hash of the push entry, and the signature of the developer revoking trust.

6.2 Security analysis

Our defense scheme fulfills the properties described in Sec. 3.1 as follows:

- Prevent modification of committed data: The existing signing mechanism for Git handles this well. Also, RSL entries are digitally signed and chained with each other, so unauthorized modifications will be detected.
- Ensure consistent repository state: The RSL provides a consistent view of the repository that is shared by all developers.
- Ensure repository state freshness: The Nonce Bag provides repository state freshness because an attacker cannot replay nonces in the Nonce Bag. Also, if no newer push entries are provided by the repository, then the attack becomes a fork attack.

The attacks described in Section 4 are prevented because, after performing the attack, the server cannot provide a valid RSL that matches the current repository

	Possible attacks	Time window of attack	Vulnerable commit objects
Commit signing	all attacks	Anytime	Any object
RSL (full adoption)	no attacks	None	No object
RSL (partial adoption)	all attacks	After the latest RSL entry and before the next RSL entry	Objects added after the latest RSL entry

Table 2: Security guarantees offered by different adoption levels of the defense scheme

state. Since she does not control any of the developers' keys, she can not forge a signature for a spurious RSL entry. As a result, a user who fetches from the main repository after the attack will notice the discrepancy between the RSL and the repository state that was presented to her. Each metadata manipulation attack would be detected as follows:

- *Branch Teleport and Deletion Attacks:* When this attack is performed, there is no mention of this branch pointing to such a commit, and the RSL validation procedure will fail.

- *Branch/Global Rollback and Tag Teleport Attacks:* These attacks can be detected because the latest entry in the RSL that corresponds to that branch points to the commit removed and the RSL validation procedure will fail.

  An attacker can attempt to remove the latest entries on the RSL so that the attacks remain undetected. However, after this moment, the server would need to consistently provide an incorrect view of the RSL to the target user, which would result in a fork attack. Finally, the attacker cannot remove RSL entries in between because these entries are chained using the previous hash field. Thus, the signature verification would fail if this field is modified.

- *Effort Duplication Attack:* This attack will result in a fork attack because the RSL created by the user requesting the commit will contain a proof about this request in the form of a nonce that has been incorporated into an RSL push entry or is still in the Nonce Bag. Any ensuing RSL push entry that was withheld from the user will not contain the user's nonce.

6.3 Partial adoption of defense scheme

It is possible that not all developers in a Git repository use our solution. This can happen when, for example, a user has not configured the Git client to sign and update the RSL. When this is the case, the security properties of the RSL change.

To study the properties of using the RSL when not everyone is using the defense, we will define a commit object as a *"secure commit"* or an *"insecure commit."* The former will be commits made by users who employ our defense, while the latter are made by users who do not use our defense (*i.e.*, they only use the Git commit signing mechanism). Consider that supporting partial adoption requires changing the validation during fetches to consider commits that are descendants of the latest secure commit, for users might push to branches without using the defense. For simplicity, we do not allow users to reset branches if they are not using the defense.

Compared to commit signing only, when our scheme is adopted by only some users, a user who writes an RSL entry might unwittingly attest to the insecure commits made by other users after the latest secure commit. However, this situation still provides a valuable advantage because the attacker's window to execute Metadata Manipulation attacks is limited in time. That is, when our defense is not used at all, an attacker can execute Metadata Manipulation attacks on any commits in the repository, (e.g., the attacker can target a forgotten branch located early in the history). This is not possible with our scheme, where an attacker can only attack the commit objects added after the latest RSL entry for that branch. The differences between the three alternatives are summarized in Table 2.

6.4 Comparison with other defenses

In Table 3, we examine the protections offered by other defense schemes against metadata manipulation attacks. Specifically, we studied how Git commit signing, Git's push certificate solution, and our solution (listed as RSL) fare against the attacks presented in this paper, as well as other usability aspects that may impact adoption.

Feature	**Commit signing**	**Push certificate**	**RSL**
Commit Tampering	✓	✓	✓
Branch Teleport	X	✓	✓
Branch Rollback	X	X	✓
Global Rollback	X	X	✓
Effort Duplication	X	X	✓
Tag Rollback	X	✓	✓
Minimum Git Version	1.7.9	2.2.0	1.7.9
Distribution Mechanism	in-band	(no default) or Additional server	in-band

Table 3: Comparison of defense schemes against Git metadata manipulation attacks. A ✓indicates the attack is prevented.

As we can see, Git commit signing does not protect against the vast majority of attacks presented in

this paper. Also, Git's push certificate solution provides a greater degree of protection, but still fails to protect against all rollback and effort duplication attacks. This is primarily because (1) a server could misbehave and not provide the certificates (there is no default distribution mechanism), and (2) a server can replay old push certificates along with an old history. Basically, this solution assumes a well-behaving server hosting push certificates.

In contrast, our solution protects against all attacks presented in Table 3. In addition to this, our solution presents an in-band distribution mechanism that does not rely on a trusted server in the same way that commit signing does. Lastly, we can see that our solution can be used today, because it does not require newer versions of Git on the client and requires no changes on the server, which allows for deployment in popular Git hosting platforms such as GitHub and Gitlab.

7 Implementation and evaluation

We have implemented a prototype for our defense scheme. This section provides implementation details and presents our experimental performance results.

7.1 Implementation

To implement our defense scheme, we leveraged *Git custom commands* to replace the push and fetch commands, and implemented the RSL as a separate branch inside the repository itself. To start using the defense, a user is only required to install two additional bash scripts and use them in lieu of the regular fetch and push commands. Our client scripts consist of less than a hundred (86) lines of code, and there is no need to install anything on the server.

RSL and Nonce Bag. We implemented the RSL in a detached branch of the repository, named "RSL." Each RSL entry is stored as a Git commit object, with the entry's information encoded in the commit message. We store each entry in a separate commit object to leverage Git's pack protocol, which only sends objects if they are missing in the local client. Encoding the Git commit objects is also convenient because computing the previous hash field is done automatically.

We also represent the Nonce Bag as a Git commit object at the head of the RSL branch. When a nonce is added or updated, a new commit object with the nonces replaces the previous nonce bag, and its parent is set to the latest RSL entry. When a new RSL entry is added, the commit containing the nonce bag is garbage collected by Git because the RSL branch cannot reach it anymore.

When `securepush` is executed, the script first fetches and verifies the remote RSL branch. If verification is successful, it then creates an RSL entry by issuing a new commit object with a NULL tree (i.e., no local files), and a message consisting of the fields described in Table 4. After the new commit object with the RSL push entry is created, the RSL branch is pushed to the remote repository along with the target branch.

Field	Description
Branch	Target branch name
HEAD	Branch location (target commit)
PREV_HASH	Hash over the previous RSL entry
Signature	Digital signature over RSL entry

Table 4: RSL push entry fields.

A `securefetch` invocation will fetch the RSL branch to update or add the random nonce in the Nonce Bag. If a nonce was already added to the commit object (with a NULL tree also), it will be amended with the replaced nonce. In order to keep track of the nonce and the commit object to which it belongs, two files are stored locally: NONCE_HEAD, which contains the reference of the Nonce Bag in the RSL branch, and NONCE, which contains the value of the nonce stored in it.

Atomicity of Git operations. The *securepush* and *securefetch* operations require fetching and/or pushing of the RSL branch in addition to the pushing/fetching to/from the target branch. Git does not support atomic fetch of multiple branches, and only supports atomic push of multiple branches after version 2.4.0 [1].

In order to ensure backwards compatibility, we designed our solution without considering the existence of atomic operations. Unfortunately, the lack of atomic push opens the possibility of a DoS attack that exploits the 'repeat' loop in Secure_fetch (lines 3-12), that makes the algorithm loop endlessly. This could happen if a user executes Secure_push and is interrupted after pushing a new RSL entry, but before pushing the target branch (e.g., caused by a network failure). Also, a malicious user may provide an updated RSL, but an outdated history for that branch. However, this issue can be easily solved if the loop is set to be repeated only a finite number of times before notifying the user of a potential DoS attempt.

If atomic push for multiple branches is available, the Secure_push procedure can be simplified by replacing lines 17-22 with a single push. Availability of atomic push also eliminates the possibility of the endless loop mentioned above.

7.2 Experimental evaluation

Experimental Setup. We conducted experiments using a local Git client and the GitHub server that hosted the main repository. The client was running on an Intel Core i7 system with two CPUs and 8 GB RAM. The client software consisted of OS X 10.11.2, with Git 2.6.2 and

[1] Note that **both Git client and server** must be at least version 2.4.0 in order to support atomic push.

the GnuPG 2.1.10 library for 1024-bit DSA signatures.

Our goal was to evaluate the overhead introduced by our defense scheme. Specifically, we want to determine the additional storage induced by the RSL, and the additional end-to-end delay induced by our `securefetch` and `secure push` operations. For this, we used the five most popular GitHub repositories [2]: bootstrap, angular.js, d3, jQuery, oh-my-zsh. We will refer to these as R1, R2, R3, R4, and R5, respectively. In the experiments, we only considered the commits in the master branch of the these repositories. Table 5 provides details about these repositories.

Repo.	Number of commits	Number of pushes	Repo. size	Repo. size with signed commits
R1	11,666	1,345	73.04	78.85
R2	7,521	26	66.96	69.79
R3	3,510	255	32.91	34.65
R4	6,031	194	15.79	19.98
R5	3,841	1,170	3.52	4.01

Table 5: The repositories used for evaluation (sizes are in MBs).

We used the repositories with signed commits as the baseline for the evaluation. We evaluated three defense schemes:

- Our defense: This is our proposed defense scheme.
- Our defense (light): A light version of our defense scheme, which does not use the nonce bag to keep track of user fetches. This scheme sacrifices protection against Effort Duplication attacks in favor of keeping the regular Git `fetch` operation unchanged.
- Push certificates: Push certificates used upon pushing.

For our defense and our defense (light), the repositories were hosted on GitHub. Given that GitHub does not support push certificates, we studied the network overhead using a self-hosted server on an AWS instance, and concluded that push certificates incur a negligible overhead compared to the baseline. Thus, we only compare our scheme with push certificates in regard to the storage overhead.

Storage overhead. Table 6 shows the additional storage induced by our defense, compared to push certificates. In our defense, the RSL determines the size of the additional storage. We can see that our defense requires between 0.009%-6.5% of the repository size, whereas push certificates require between 0.012%-10%. The reason behind this is that push certificates contain 7 fields in addition to the signature, whereas RSL push entries only have 3 additional fields.

[2]Popularity is based on the "star" ranking used by GitHub, which reflects users' level of interest in a project (retrieved on Feb 14, 2016).

Repo.	Our defense	Push certificates
R1	301.93	461.27
R2	6.49	8.88
R3	58.91	86.05
R4	44.34	66.27
R5	261.3	402.19

Table 6: Repository storage overhead of defense schemes (in KBs).

Communication overhead. To evaluate the additional network communication cost introduced by our `securepush` operation when compared to the regular `push` operation, we measured the cost of the last 10 pushes for the five considered repositories. To evaluate the cost of `securefetch`, we measured the cost of a fetch after each of the last 10 pushes.

Table 7 shows the cost incurred by push operations. We can see that our defense incurs, on average, between 25.24 and 26.21 KB more than a regular `push`, whereas our defense (light) only adds between 10.29 and 10.48 KB. This is because a `securepush` in our defense retrieves, updates and then stores the RSL in the remote repository. In contrast, our defense (light) only requires storing the RSL with the new push entry if there are no conflicts. Table 8 shows the cost incurred by fetch operations. A `securefetch` incurs on average between 25.1 and 25.55 KB more than a regular `fetch`, whereas our defense (light) only adds between 14.34 and 10.91 KB.

The observed overhead is a consequence of the fact that we implemented our defense scheme to respect design goal DG3, (*i.e.* no requirement to modify the Git server software). Since we implemented the RSL and the Nonce Bag as objects in a separate Git branch, `securepush` and `securefetch` require additional `push`/`fetch` commands to store/fetch these, and thus they incur additional TCP connections. Most of the communication overhead is caused by information that is automatically included by Git and is unrelated to our defense scheme. We found that Git adds to each `push` and `fetch` operation about 8-9 KBs of supported features and authentication parameters. If our defense is integrated into the Git software, the `securepush` and `securefetch` will only require one TCP session dramatically reducing the communication overhead. In fact, based on the size of an RSL entry ($\sim$325 bytes), which is the only additional information sent by a `securepush`/`securefetch` compared to a regular `push`/`fetch`, we estimate that the communication overhead of our defense will be less than 1KB per operation.

End-to-end delay. Table 9 shows the end-to-end delay incurred by push operations. We can see that our defense adds on average between 1.61 and 2.00 seconds more than a regular `push`, whereas our defense (light) only adds between 0.99 and 1.3 seconds. Table 10 shows

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	17.80	3,925.35	38.32	59.14	11.96
Our defense	44.01	3,950.87	63.56	84.71	37.65
Our defense (light)	28.28	3,935.71	48.61	69.52	22.28

Table 7: Average communication cost per push for the last 10 push operations, expressed in KBs.

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	20.68	3,896.98	40.93	65.85	13.67
Our defense	46.18	3,922.40	66.48	91.27	38.77
Our defense (light)	35.19	3,911.81	55.84	80.67	28.01

Table 8: Average communication cost per fetch for the last 10 fetch operations, expressed in KBs.

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	1.29	3.27	1.17	1.31	1.51
Our defense	3.11	5.27	2.78	2.95	3.51
Our defense (light)	2.44	4.49	2.16	2.40	2.81

Table 9: Average end-to-end delay per push for the last 10 push operations, expressed in seconds.

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	0.87	1.95	0.75	0.66	0.67
Our defense	2.93	3.86	2.52	2.40	2.75
Our defense (light)	1.60	2.75	1.52	1.31	1.30

Table 10: Average end-to-end delay per fetch for the last 10 fetch operations, expressed in seconds.

the end-to-end delay incurred by fetch operations. We can see that a `securefetch` incurs on average between 1.74 and 2.08 seconds more than a regular `fetch`, whereas our defense (light) only adds between 0.65 and 0.8 seconds.

The time Git uses to do a fetch or push is dominated by the network latency when talking with the remote repository. Since our defense is designed to be backwards compatible, it uses multiple Git commands per push or fetch. This explains the additional time incurred by our implementation. If our defense scheme is integrated into Git so that additional commands (and hence network connections) are not needed, we expect the additional delay to be negligible.

8 Conclusions

In this work, we present a new class of attacks against Git repositories. We show that, even when existing Git protection mechanisms such as Git commit signing, are used by developers, an attacker can still perform extremely impactful attacks, such as removing security patches, moving experimental features into production software, or causing a user to install a version of software with known vulnerabilities.

To counter this new class of attacks, we devised a backwards compatible solution that prevents metadata manipulation attacks while not obstructing regular Git usage scenarios. Our evaluation shows that our solution incurs less than 1% storage overhead when applied to popular Git repositories, such as the five most popular repositories in GitHub.

We performed responsible disclosure of these issues to the Git community. We have been working with them to address these issues. Some of our patches have already been accepted into Git version 2.9. We are continuing to work with the Git community to fix these problems.

Acknowledgements

We would like to thank Junio C. Hamano, Jeff King, Eric Sunshine, and the rest of the Git community for their valuable feedback and insight regarding these attacks and their solutions as well as their guidance when exploring Git's internals. Likewise, we thank Lois A. DeLong, Vladimir Diaz, and the anonymous reviewers for their feedback on the writing on this paper.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. A8650-15-C-7521, and by the National Science Foundation (NSF) under Grants No. CNS 1054754, DGE 1565478, and DUE 1241976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, and NSF. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.

References

[1] Apso: Secrecy for Version Control Systems. http://aleph0.info/apso/.

[2] Git signed push. http://thread.gmane.org/gmane.comp.version-control.git/255520.

[3] Github. https://github.com.

[4] Kernel.org Linux repository rooted in hack attack. http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/.

[5] 365 Git. Adding a GPG key to a repository. http://365git.tumblr.com/post/2813251228/adding-a-gpg-public-key-to-a-repository.

[6] Ars Technica. "flame malware was signed by rogue ca certificate". http://arstechnica.com/security/2012/06/flame-malware-was-signed-by-rogue-microsoft-certificate/.

[7] Ars Technica. Lenovo pcs ship with man-in-the-middle adware that breaks https connections. http://arstechnica.com/security/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/.

[8] Beta News. Has SSL become pointless? Researchers suspect state-sponsored CA forgery. http://betanews.com/2010/03/25/has-ssl-become-pointless-researchers-suspect-state-sponsored-ca-forgery/.

[9] Briarproject. Development Workflow. https://code.briarproject.org/akwizgran/briar/wikis/development-workflow.

[10] Bundler.io. Bundler: the best way to manage your application's GEMS. http://bundler.io/git.html.

[11] Code in the hole. Using pip and requirements.txt to install from the head of a github branch. http://codeinthehole.com/writing/using-pip-and-requirementstxt-to-install-from-the-head-of-a-github-branch/.

[12] Delicious Brains. Install wordpress site with Git. https://deliciousbrains.com/install-wordpress-subdirectory-composer-git-submodule/.

[13] Extreme Tech. GitHub Hacked, millions of projects at risk of being modified or deleted. http://www.extremetech.com/computing/120981-github-hacked-millions-of-projects-at-risk-of-being-modified-or-deleted.

[14] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design & Implementation (OSDI '10)*, 2010.

[15] gamasutra. Cloud source host Code Spaces hacked, developers lose code. http://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php.

[16] Geek.com. Major Open Source Code Repository Hacked for months, says FSF. http://www.geek.com/news/major-open-source-code-repository-hacked-for-months-says-fsf-551344/.

[17] M. Gerwitz. A Git Horror Story: Repository Integrity With Signed Commits. http://mikegerwitz.com/papers/git-horror-story.

[18] Gigaom. Adobe source code breach; it's bad, real bad. https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad/.

[19] Git SCM. Signing your work. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work.

[20] M. Gunderloy. Easy Git External Dependency Management with Giternal. http://www.rubyinside.com/giternal-easy-git-external-dependency-management-1322.html.

[21] E. Homakov. How I hacked GitHub again. http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html.

[22] How To Node. Managing module dependencies. http://howtonode.org/managing-module-dependencies.

[23] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged ssl certificates in the wild. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 83–97, Washington, DC, USA, 2014. IEEE Computer Society.

[24] I2P. Setting trust evaluation hooks. https://geti2p.net/en/get-involved/guides/monotone#setting-up-trust-evaluation-hooks.

[25] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proc. of the 36th IEEE Symposium on Security and Privacy (S&P '15)*, 2015.

[26] J. Li, M. Krohn, DMazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI '04)*, 2004.

[27] LWN. Linux kernel backdoor attempt. https://lwn.net/Articles/57135/.

[28] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, 2011.

[29] J. Pellegrini. Secrecy in concurrent version control systems. In *Presented at the Brazilian Symposium on Information and Computer Security (SBSeg 2006)*, 2006.

[30] RubyGems.org. Data verification. http://blog.rubygems.org/2013/01/31/data-verification.html.

[31] Schneier on Security. Forging SSL Certificates. https://www.schneier.com/blog/archives/2008/12/forging_ssl_cer.html.

[32] R. Shirey, K. Hopkinson, K. Stewart, D. Hodson, and B. Borghetti. Analysis of implementations to secure git for use as an encrypted distributed version control system. In *48th Hawaii International Conference on System Sciences (HICSS '15)*, pages 5310–5319, 2015.

[33] Stack Overflow. How to install from specific branch with OPAM? https://stackoverflow.com/questions/25277599/how-to-install-from-a-specific-git-branch-with-opam.

[34] Stack Overflow. Loading Maven dependencies from GitHub. https://stackoverflow.com/questions/20161602/loading-maven-dependencies-from-github.

[35] Stack Overflow. Where does Cargo put the Git requirements? https://stackoverflow.com/questions/28069678/where-does-cargo-put-the-git-requirements.

[36] The Art of Simplicity. TFS Build: Build from a tag. http://bartwullems.blogspot.com/2014/01/tfs-build-build-from-git-tag.html.

[37] ThreatPost. Certificates spoofing google, facebook, godaddy could trick mobile users. https://threatpost.com/certificates-spoofing-google-facebook-godaddy-could-trick-mobile-users/104259/.

[38] US-CERT. "SSL 3.0 Protocol Vulnerabilty and POODLE attack". http://arstechnica.com/security/2012/06/flame-malware-was-signed-by-rogue-microsoft-certificate/.

[39] D. A. Wheeler. Software Configuration Management (SCM) Security. http://www.dwheeler.com/essays/scm-security.html.

[40] D. A. Wheeler. "The Apple goto fail vulnerability: lessons learned". http://www.dwheeler.com/essays/apple-goto-fail.html.

[41] Wired. Behind iphones critical security bug, a single bad goto. http://www.wired.com/2014/02/gotofail/.

[42] Wired. 'Google' Hackers had ability to alter source code'. https://www.wired.com/2010/03/source-code-hacks/.

[43] ZDNet. Gogo in-flight wi-fi serving spoofed ssl certificates. http://www.zdnet.com/article/gogo-in-flight-wi-fi-serving-spoofed-ssl-certificates/.

[44] ZDNet. Open-source ProFTPD hacked, backdoor planted in source code. http://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code/#!

[45] ZDNet. Red Hat's Ceph and Inktank code repositories were cracked. http://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked/#!

A Man In The Middle Example

This appendix contains a proof of concept of a Git metadata manipulation attack against a GitHub repository with the intention of showing how an attack could be carried out in practice.

To perform an attack of this nature, an attacker controls a server, compromises a server, or acts as a man-in-the-middle between a server and a developer. Having done this, the attacker is able to provide erroneous metadata to trick a developer into committing a tampered repository state.

We simulated a repeated line scenario, in which a Git merge accidentally results a repeated line. This can be devastating as it can completely alter the flow of a program — some researchers argue that the "goto fail;" [41] vulnerability that affected Apple devices [40] might have been caused by a VCS mistakenly repeating the line while merging.

A.1 Simulating the attack

To simulate the attack, we created a repository with a minimal working sample that resembles Figure 5(c). Also, we configured two Linux machines under the same network: one functioned as the malicious server providing tampered metadata information, while the other played the role of the victim's client machine. The specific setup is described below.

Setup. To simulate the malicious server, we set up Git server on port 443 with no authentication enabled. Then, we created an SSL certificate and installed it in the victim machine. Finally, we a bare clone (using the `--bare` parameter) of the repository hosted on GitHub is created and placed on the pertinent path.

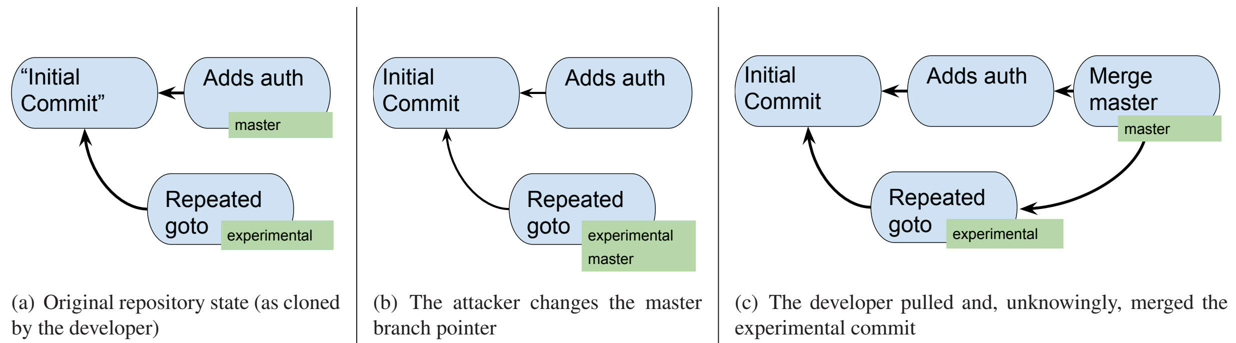
In order to redirect the user to the new branch, we modified the packed-refs file on the root of the repository so that the commit hash in the master branch matches the one for the experimental branch. Refer to Table 11 for an example.

On the client side, a clone of the repository is created before redirecting the traffic. After cloning, the attacker's IP address is added to the victim's /etc/hosts file as "github.com" to redirect the traffic.

As such, both the server and the developer are configured to instigate the attack the next time the developer pulls.

A.2 The attack

When the developer pulls, he or she is required to either merge or rebase the vulnerable changes into the working branch. These merged or rebased changes are not easy to identify as malicious activity, as they just resemble work performed by another developer on the



(a) Original repository state (as cloned by the developer)

(b) The attacker changes the master branch pointer

(c) The developer pulled and, unknowingly, merged the experimental commit

Figure 5: Maliciously merging vulnerable code

Original file
pack-refs with: peeled fully-peeled
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/experimental
3a1db2295a5f842d0223088447bc7b005df86066 refs/heads/master
Tampered file
pack-refs with: peeled fully-peeled
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/experimental
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/master

Table 11: The edited packed-refs file

same branch. Due to this, the user is likely to merge and sign the resulting merge commit.

Aftermath. Once the user successfully merges the vulnerable change, the attacker can stop re-routing the user's traffic to the malicious server. With the malicious piece of code in the local repository, the developer is now expected to pollute the legitimate server the next time he or she pushes. In this case, the attacker **was able to merge a vulnerable piece of code into production**. Even worse, there is no trace of this happening, for the target developer willingly signed the merge commit object.

Setting up an environment for this attack is straightforward; the metadata modification is easy to perform with a text editor and requires no sophistication.