The Benefits of Vulnerability Discovery and Bug Bounty Programs: Case Studies of Chromium and Firefox (Appendix)

Anonymous

Anonymous

1 Data

Here, we describe the main steps of our data collection and cleaning process for both Chromium and Firefox VRPs.

2 Data Collection

2.1 Chromium Issue Tracker

We collected all data from September 2, 2008 to September 13, 2022 from the Chromium issue tracker[1] using Monorail API version 2[2]. We use three types of request from the Monorail API for data collection: (i) *ListIssues*, which returns the list of reports that satisfy the query specified in the request; (ii) *GetIssue*, which returns the details of the report corresponding to the report identification number specified in the request; and (iii) *ListComments*, which returns the list of comments posted on the report specified in the request. For each vulnerability's report, the Chromium issue tracker stores a list of comments, which includes conversations among internal employees and external parties as well as a history of changes (i.e., amendments) made to the report.

Each report contains the following fields:

- *IdentificationNumber*: A unique number that identifies the report.
- *Status*: Current state of the report (*Unconfirmed*, *Untriaged*, *Invalid*, *Duplicate*, *Available*, *Assigned*, *Started*, *ExternalDependency*, *Fixed*, *Verified*, and *Archived*).
- *Component*: Component (or components) of the Chromium project that are affected by the report.
- *Owner*: Email address of the person who currently owns the report (e.g., reporter of the vulnerability or the person who fixes or closes the vulnerability).

[1] https://bugs.chromium.org/p/chromium/issues/
[2] https://chromium.googlesource.com/infra/infra/+/master/appengine/monorail/api/README.md

- *AllLabels*: Labels associated with the report. These labels are used to categorize reports, e.g., to indicate security-severity levels (*critical*, *high*, *medium*, or *low*), impacted versions (*stable*, *beta*, or *head*), reward amount for bug bounty (e.g., `Reward-500` indicates that $500 is awarded to the reporter of the vulnerability), or CVE ID.
- *Summary*: Short description of the report.
- *ReporterEmail*: Email address of the person who reported the report.
- *CCDetails*: Email addresses of all users who are part of the conversation thread of the report.
- *OpenedTimestamp*: Date and time when the report was initially reported.
- *ClosedTimestamp*: Date and time when the report was closed.
- *BugType*: Type of the report (e.g., `Bug-Security`).
- *MergedInto*: This is an optional field that applies only to duplicate reports. This field references the original report.

Each comment posted on a report consists of the following fields:

- *CommenterEmail*: Email address of the person who posted the comment.
- *Content*: Text of the comment, images of the report, videos of how to reproduce the report, etc.
- *SequenceNumber*: Order of the comment among all comments posted on the report.
- *CommentedTimestamp*: Date and time when the comment was posted on the report.
- *Amendments*: Updating or removing the values of some fields of the report (e.g., changing the owner, status, or priority).

Each amendment added to a comment consists of the following fields:

- *FieldName*: Name of the field that the amendment changes.
- *OldValue*: List of previous values of the field. This an optional field.
- *NewOrDeltaValue*: List of new and removed values of the field.
- *AmendmentTimestamp*: Date and time when the amendment was posted.

Chrome Releases We also collected all the release notes from the Chrome Releases blog[3], which provides information regarding both the closed-source Chrome and the open-source Chromium projects. A release note is a blog post written by Google when they officially release a new version of Chrome or Chromium. Each Chromium release note contains a list of vulnerabilities that Google patches in the Chromium project when releasing the corresponding version. Each entry in this list of vulnerabilities contains the following fields:

- *IdentificationNumber*: Unique identification number of a report. We use this to join the Chromium issue tracker data with the Chrome Releases dataset.
- *ReporterName*: List of bug hunters who reported the particular vulnerability.
- *Association*: Organization (or organizations) where the reporters work.
- *ReleaseDate*: Release date of Chromium version that includes the fix for the vulnerability.

[3] https://chromereleases.googleblog.com/

Google Git For Programming Languages Analysis Another data resource that we use in our study is the Google Git repository[4]. From analyzing comments on vulnerabilities that we collected from the Chromium issue tracker, we find that most vulnerabilities that have been fixed have links to the Google Git repository, which we can use to identify the files that were changed to fix the vulnerability. For each vulnerability with a Google Git repository link, we collected the programming languages of the files that were changed.

2.2 Mozilla Firefox VRP

We collected data from two main resources, Bugzilla [5] (Firefox bug tracker), and Known Vulnerabilities from Mozilla website [6] which is a list of security advisories based on product and advisories for older products which all are listed in the Mozilla Foundation Security Advisories (MFSA) [7]. Overall, we collected security issues from January 24, 2012 to August 25, 2022.

In order to collected security vulnerabilities, we used security keywords added to the URL of Bugzilla search and scraped all URLs of reports which have at least one of the security keywords in the report *Keywords* field. Finally, all of the information related to a report was scraped.

Each report contains several fields which the collected ones are listed below:

- *BugID*: A unique identifier of the report.
- *CVE*: CVE Id of the report (does not exist for all of the reports).
- *Opened*: Date and time when the report is opened.
- *Closed*: Date and time when the report is closed.
- *Summary*: A brief summary of the report.
- *Product*: Product type which the report is related to (we are interested in the Core and Firefox).
- *Component*: Component (or components) of the Firefox that are affected by the vulnerability.
- *Type*: This field represents type of the bug. It contains three types: defect, enhancement, and task. We are only interested in the defect type.
- *Status*: This represents current status of the report and what has happened to the report. It contains UNCONFIRMED, NEW, ASSIGNED, REOPENED for reports that are open. For reports that are closed, *Status* field contain, RESOLVED and VERIFIED which each have 7 resolutions (resolution represents the approach applied to reach to the current status): FIXED, INVALID, WONTFIX, MOVED, DUPLICATE, WORKSFORME, and INCOMPLETE.
- *Reporter* Username of a person who reported the issue.
- *Keywords* Criticality of the report (critical, high, medium, low) is mentioned in this field.

[4] https://chromium.googlesource.com/
[5] https://bugzilla.mozilla.org/home
[6] https://www.mozilla.org/en-US/security/known-vulnerabilities/
[7] https://www.mozilla.org/en-US/security/advisories/

- *Duplicates* ID number of duplicates of the report.
- *Whiteboard* It contains tags, or information of a report's status. *reporter-external* tag is one of the tags which is used to identify external versus internal reporter.
- *Bug Flags* It contains *sec-bounty* value and it does not exist for all of the reports.
- *TrackingFlagsStatus* It contains vulnerability's statuses tracked by developers (does not exist for all of the reports).
- *Comments* All comments posted on the report.

Fixed Timestamp Each report that has *VERIFIED* or *RESOLVED* followed by *FIXED* in its *Status* field, has an arrow followed by *RESOLVED* ($\rightarrow$ RESOLVED) and an arrow followed by *FIXED* ($\rightarrow$ FIXED) in one of its last comments (date of that comment which equals to the close time of the report). We use that date (close time) as the date the vulnerability is fixed. There are some reports that are closed because of incomplete fix status and reopened again. In those cases, we consider the last fixed time as the fix time of that vulnerability.

Mozilla Foundation Security Advisories (MFSA) MFSA reports vulnerabilities for Mozilla products. In this paper, we focus on Firefox. To identify reports pertaining to *stable* releases, we use MFSA. For FireFox and its older versions, we scraped advisories to be able to label reports that pertain to stable releases. We also collected the *Reporter* field, which some pages in MFSA have, to identify external versus internal reporters in our cleaning process.

Firefox Modified Source Files For Programming Languages Analysis Most vulnerabilities that have been fixed have links to the Mozilla source-code repositories in their comments, which we use to identify the files that were changed to fix the vulnerability. For each vulnerability with a repository link, we collect the programming languages of the files that were changed.

Reopened In Firefox, some reports had incomplete status due to the lack of information for replication and patching. For some of them, Mozilla reopened a new report of the vulnerability, which was then completed with respect to this information, and marked the first report as a duplicate. These reports can be identified by searching a right arrow to *REOPENED* ($\rightarrow$ REOPENED) in the comments of that reports. Later in our analysis, we exclude these reports from rediscovery analysis since they are not actual rediscoveries.

2.3 CVEs and CWEs

CVEDetails We leverage CVEDetails[8] and MITRE CWE[9] to collect information regarding CVE IDs and weakness types (CWEs), when available for both

[8] https://www.cvedetails.com/
[9] https://cwe.mitre.org/

Chromium and Firefox. One of the fields associated with a report is *AllLabels* in Chromium. These labels may include a categorical parameter *Common Vulnerabilities and Exposures (CVE) Entry*, which contains an identification number called *CVE ID*. In Firefox, some reports contain *CVE ID* field which we collected them for analysis. These identifiers are used by cybersecurity product and service vendors and researchers as one of the standard methods for identifying publicly known vulnerabilities and for cross-linking with other repositories that also use CVE IDs. Using these CVE IDs, we collected CVSS scores, impact metrics, and weakness types from CVEDetails[10]. For each report with a CVE ID, we collected the following details:

– *CVSS Score*: *Common Vulnerability Scoring System* (CVSS) provides a way of capturing the fundamental characteristics of a vulnerability. This numerical score reflects the severity of the vulnerability.
– *Confidentiality Impact*: Impact of successful exploitation on information access and disclosure.
– *Integrity Impact*: Impact of successful exploitation on the trustworthiness and veracity of information.
– *Availability Impact*: Impact of successful exploitation on the accessibility of information resources.
– *Access Complexity*: Complexity of the attack required to exploit the vulnerability once an attacker has gained access to the system.
– *CWE ID*: *Common Weakness Enumeration* (CWE) is a community-developed list of weakness types. The CWE ID references the type of software weakness associated with the particular vulnerability.

Weakness Types (CWE IDs) The CWE IDs associated with the vulnerabilities represent common types of software weaknesses. We later use CWE ID collected from *cvedetails* to collect broad-type names of CWEs from MITRE for weakness type analyses. Some of these weakness types have a hierarchical relationship with other types. For example, CWE 119 denotes the error ”Improper Restriction of Operations within the Bounds of a Memory Buffer.” This weakness type is also the parent of other CWEs, including CWE 120 (Classic Buffer Overflow), CWE 125 (Out-of-bounds Read), and CWE 787 (Out-of-bounds Write). For ease of presentation, we group the CWE weakness types together based on their parent-children hierarchy.

3 Data Cleaning

In our analysis, we only consider reports that satisfy at least one of the following three conditions: (1) the report is an original report, and it has at least one security label; (2) the report is an original report, and the value of field BugType is *Bug-Security* for Chromium or has one of security labels in *Keywords* field for Firefox; and (3) the report is a duplicate report, and its original report satisfies at least one of the above conditions.

[10] https://www.cvedetails.com/

3.1 Duplicate Reports

3.1.1 Chromium A report in the issue tracker is considered to be a *duplicate* if the underlying report has already been reported to the Chromium issue tracker (i.e., if this is a rediscovery). We can determine whether a report is a duplicate or not based on the *Status* field of the report: if the *Status* field is marked as *Duplicate*, the report is a duplicate.

To facilitate studying vulnerability rediscovery, we find the original report of each duplicate report as follows. For each duplicate report D, we follow the *MergeInto* field to retrieve the report referenced by it. If that is a duplicate report, we again follow the *MergeInto* field of the referenced report. We continue this process recursively until either one of the following holds:

- We reach a report O that is not a duplicate report. In this case, report O is the *original report* of duplicate report D.
- We reach a report X that is a duplicate report but does not have any references in the *MergedInto* field (or the value of *MergedInto* field is malformed). In this case, we say that the duplicate report D does not have an original report.

We include a duplicate report D in our rediscovery analysis if report D has an original report O and report O has at least one security-related label. In order to retrieve duplicates of a vulnerability, we use *Duplicates* field which has references to the duplicate reports. For the cases that references also have a reference to other duplicates, we recursively retrieve duplicate reports until there is not any reference to a duplicate report.

3.1.2 Firefox We can determine whether a vulnerability is reported before (is a duplicate) or not based on the *Status* field. If the report is a duplicate, *Status* field contains the keyword *Duplicate* and it has reference to the original report. In some cases, the referenced report, which is supposed to be the original report, has *Duplicate* in the status and has reference to another report. In these cases, we recursively, retrieve the report which is referenced in the *Status* field until there is not any reference to a report.

3.2 Valid and Invalid Reports

For both Chromium and Firefox, if the *Status* field of an original report is not marked as *Invalid*, it is considered a *valid original report.* If a duplicate report has a valid original report, then the duplicate report is a *valid duplicate report.* If a vulnerability belongs to either valid original reports or valid duplicate reports, then the vulnerability is considered a *valid vulnerability.*

If the *Status* field of an original report is marked as *Invalid*, it is considered an *invalid original report.* If a duplicate report has an invalid original report, then the duplicate report is an *invalid duplicate report.* If a report belongs either to invalid original reports or invalid duplicate reports, then, the report is considered an *invalid report.*

In Firefox, there are other invalid statuses that we do not consider them as valid statuses for reports. We do not consider duplicate reports that their original report has *INACTIVE*, *INCOMPLETE*, *MOVED*, *WONTFIX*, *WORKSFORME*, or *UNCONFIRMED* in its' *Status* field. However, there is an exception here. By checking some of the reports with the mentioned statuses, we realized that some reports that have *WORKSFORME* or *INCOMPLETE* in their *Status* field, have *fixed* word in their *TrackingFlags* field. Therefore, we keep duplicate reports that their original has *WORKSFORME* or *INCOMPLETE* in its' status and it got fixed in a version (according to the 'fixed' word in the *TrackingFlags* field).

3.3 Type and Product

In Bugzilla (Firefox) there is a *Type* field that contains type of a vulnerability which can be task, enhancement, or defect. We only keep duplicate reports that their original report have *defect* type. As for *Product* field, we keep only duplicate reports that their original report's product contains *Core* or *Firefox*.

3.4 External and Internal Reports

3.4.1 Chromium The Chromium issue tracker contains reports either reported internally by Google or reported externally by bug hunters. For each report, we use the reporter's email address to classify the report as either an *internal* or an *external report*. However, not all email addresses fall into the internal vs. external classification; thus, we cannot always determine the reported origin based on the email address alone. For each such address, we manually check the activities of the email address, such as vulnerabilities reported and comments posted by this particular email address. We determine the reporter's origin based on the activities associated with a particular email address.

There are also cases where the email address could be misleading. First, some external bug hunters submit reports privately to Google, and internal experts then post these reports on the Chromium issue tracker. Second, sometimes internal reporters import reports from other bug-bounty programs (e.g., Firefox, Facebook). In these cases, we need to identify the actual external reporter for each replicated report by analyzing the CC email address list of the report. We further improve the data cleaning process of distinguishing internal and external reports using the data we collected from Chrome Releases. The detailed cleaning process can be described using the following steps.

Step 1: Initial Classification based on ReporterEmail Field

For each report *I*, we use the email address of the reporter to classify the report *I* as either an *internal* or an *external* report. Specifically, if the email address is *ClusterFuzz* or ends with *google.com*, *chromium.org*, or *gserviceaccount.com*, and does not contain any label stating *external_ security_ report* or label starting *reward-to-external* and contains a non google email (i.e., email address without google.com or chromium.org) then we consider report *I* to be internally reported; otherwise, we consider it to be externally reported.

Step 2: Identifying Outlier Email Addresses based on Comments

We found that some of the reporters have email addresses that do not fit the rules of Step 1. One exception is the gmail address *scarybeast*, which belongs to an internal reporter. We identified this exception by analyzing the comments posted on reports reported by this email address. Based on the comments, we determined that this reporter is one of the key persons in announcing the confirmation of the reward to external reporters. Thus, we consider reports reported by this email address as internal reports. [11]

We also find some other exceptions where the email address of the reporter *skylined* or *cnardi* end with chromium.org. When we analyze the comments on reports reported by *skylined* with chromium.org address, we realized he served as a team member of the Google Chrome Security Team from 2008 - 2013 and left Google. After leaving Google, he reported few vulnerabilities as an external reporter and received rewards. When we analyze the comments on reports reported by *cnardi* ends with chromium.org, one comment mentions "`cnardi@chromium.org` as an external reporter regardless of his email address ends with @chromium.org." Accordingly, we classify him as an external reporter. We consider the reports reported by these two reporters as external reports.

Step 3:Analyzing CCDetails Field and Identifying the Actual External Reporters

Some reports that are reported by internal-reporters are replications of reports privately reported by external reporters to Google or reports imported from other bug bounty programs (e.g., Firefox, Facebook). Google replicates most of these externally reported reports through the automated tool `ClusterFuzz`, but sometimes Google replicates them manually using internal reporters (e.g., scarybeasts@gmail.com). For each replicated report, we need to identify the actual external reporter. We use the following approach and identify the email address of the external reporter of those reports.

For each report I for which we have to identify the email address of the actual external reporter, we first extract the CC email addresses (CC_{all}) from the *CCDetails* field. From CC_{all}, we obtain a new list CC_{remain} by removing the email address where the email address belongs to an internal reporter at the end of Step 2. For each email address in the CC_{remain} list, we look into comments of the corresponding report I whether any comment has one of the following phrases "originally reported by", "thanks to", "credits to", "thanks", "credits", "reward", "congratulations" immediately followed by username or email address or full name of the reporter. If the username of the email address or the reporter's full name matches the email address, we add the particular email address to the possible-reporters list.

We repeat the same process for every email address on the list. After the process finishes, we check the possible-reporters list of the report I. If the possible-reporters list is empty, we set the *ReporterEmail* field of the report as empty (there are 50 reports for which we cannot identify the email address of the actual external reporter during this data cleaning process). If the possible-reporters list

[11] We believe this person was actually a member of the Google Chrome Security Team.

is not empty, then we set the *ReporterEmail* field of the report with the list of email addresses in the potential-reporters list.

Even though there should be only one reporter for each report (i.e., the length of the potential-reporters list should be one), we observe some reports where multiple reporters are rewarded. This may happen when multiple external bug hunters report the same report to Google (not through the issue tracker). Google replicates these reports by posting a single report on the tracker via an internal reporter. In such cases, we let the reporter's email of the report be a list instead of a single email address. Note that for some of these reports with multiple reporters, we perform an additional verification in Step 4.

Step 4: Cleaning based on Chrome Releases Data

Further, we improve the data cleaning process of internal and external reports based on data collected from Chrome Releases (Section 2.1). During the last step (Step 3), we mark the *ReporterEmail* field as empty for the reports where we are unable to determine the actual external reporter.

For each report I which marked the *ReporterEmail* field as empty in the last step (Step 3), we look for a data entry *DE* with the *IdentificationNumber* field same as the Identification Number field of report *I*. If a data entry exists in the Chrome Releases dataset, then we set the Report Email of report I with the Reporter Name in data entry *DE*. Accordingly, we are able to identify the actual external reporter details of 14 reports.

Further, during the last step (Step 3), we have more than one email address set to the *ReporterEmail* field for 13 reports. For each report I in those 13 reports, we look for a data entry *DE* with Identification Number field the same as the *IdentificationNumber* field of report *I*. If there exists a data entry (*DE*) in the Chrome Releases dataset, then we check the value in the ReporterName field of *DE*; if it indicates a single person, then we update the *ReportEmail* field with the single person. We are able to update 8 out of 13 reports with multiple reporters to the actual external reporters. At the end of this step, we were left with 22 reports to go through an additional cleaning process to identify the original *ReporterEmail* field of the report.

For each reporter name *RN* used as the *ReporterEmail* field of the above 22 reports, we list out the reports (L_{RN}) reported by the reporter *RN* based on the Chrome Releases dataset. For each report in L_{RN}, we look for report *I*, which has the same Identification Number and Reporter Email in the email address format. If we obtain the report *I*, then we map the reporter name RN with the *ReporterEmail* field of report I; otherwise, we continue the same process with the next report in the list.

Finally, we repeat Step 1 with the cleaned dataset based on Steps 3 and 4. We identify the email address of the actual external reporter for 98% of valid external reports.

3.4.2 Firefox Report in Firefox VRP are reported either internally by Firefox internal team members or by external reporters.

We use 4 steps to separate internal versus external reports. First, we use *Whiteboard* and *bug-flag* fields on the report information page. If a report has *reporter-external* in *Whiteboard* field or *sec-bounty* in *bug-flag* field, we consider that report as an externally reported report; otherwise, it is considered as an internally reported report. However, there are reports that do not have the above keywords in the mentioned fields, they are reported by external reporters. To separate them, we added three more cleaning steps. In the second step, we leverage the snow-balling technique (on the comments such as 'security@mozilla.org received the following report from') to identify reports (around 650 reports) reported by internal security members of Mozilla and do not have any bounty tag, but their original reporters are external. In the third step, we check reporters with both internal and external reports (around 50). We manually check whether they are internal or external (by reading comments and checking their social networking websites). In the last step of the cleaning process, we leverage *Reporter* field in MFSA and match the name of reporters in their profile names (we got each reporter's profile name from Bugzilla) with the name of the reporter mentioned in MFSA. By applying the above steps, we are able to separate internal versus external reports with 97% accuracy.

3.5 Vulnerability Attributes

For each duplicate report D, we clean security-severity, impacted releases, weakness types, components, and programming language attributes to make them consistent with its original report O. Accordingly, we perform the following cleaning steps,

– If the duplicate report D does not list any value for an attribute A and the original report O of duplicate report D has a value, then we set the value for attribute A of duplicate report D to the value of the original report O
– If all duplicate reports of original report O list the same value and original report O does not list any value for an attribute A, then we set the value for attribute A of original report O to the value of the duplicate reports.

Further explanations for different attributes based on the datasets are in the following sections.

3.5.1 Security-Severity

Chromium There are four severity levels in the Chromium issue tracker that describe the security severity of a vulnerability: *Critical*, *High*, *Medium*, and *Low*. For each original report, we identify its security-severity level by extracting labels that start with *Security_Severity* from the *AllLabels* field of the report. If a label in the format of *Security_Severity-L* is available in the list of labels (where L is one of the four security-severity levels), then the severity of the report is L. If no labels are available in the format of *Security_Severity-L* in the list of labels, then we consider the severity of the report to be *unclassified*. For each duplicate report D, we use the security-severity level of the corresponding original report O instead of the security-severity level of the duplicate report D. We exclude *unclassified* vulnerabilities from the security-severity analysis.

Firefox There are multiple security related keywords in *Keywords* field of a report. We only include reports in our analysis that contain one of the 6 following security tags in their *Keywords* field: *sec-critical, sec-high, sec-medium, sec-low, sec-vector, and sec-other*. If a report has one of the above keywords in its *Keywords* field, we consider that report as a security report and include it for our analysis. There are some reports that have more than one security keyword. For those reports, we keep them as security reports in the dataset but exclude them from the analysis parts related to the security severity. We also realized that most of the collected reports that are opened in 2011 and before that year, do not have any security keywords assigned. Therefore, we only consider reports that are opened in 2012 and later for our analysis. For each duplicate report D, we use its corresponding original report's keyword as the duplicate security-severity field. There are reports with other security related keywords in their *Keywords* field which we exclude them since they are not actually security reports. For instance, reports which have *sec-want* which is 'New features or improvement ideas related to security' according to Mozilla keywords explanation are excluded. [12]

3.5.2 Release Channels

Chromium Google categorizes release versions as *stable*, *beta*, and *dev*. Stable is the release that is available for end-users. Beta is the release that is available for a limited number of users to test features before releasing a stable release. Dev (commonly referred to as *head*) is the release that is based on the last successful build. We use the term *release channel* to refer to these release versions throughout the paper. Note that the term release version means the type of the release instead of the version number (e.g., Version 90 and Version 91).

Each security vulnerability I impacts one or more release channels. To identify which security channel(s) is affected by vulnerability I, we check labels in *AllLabels* field that start with *Security_Impact*. Based on these, we identify three release channels during this process: stable, beta, and head. We group beta and head as development release channels and perform our analysis.

Firefox We followed the general approach we mentioned at the beginning of this section Section 3.5.

3.5.3 Components

[12] https://bugzilla.mozilla.org/describekeywords.cgi

Chromium For each report I, we identify the components from its *Component* field. Each report I will contain a set of components C_I. For each component c in the C_I, we extract the set of the group, which indicates a list of all sub-levels from the top level to the bottom level of the component hierarchy. For example, if report I has a component `Internals>Plugins>PDF`, we extract the set of the group as `Internals,Internals>Plugins,Internals>Plugins>PDF`. We use G_I to denote the set of all groups that correspond to all the components of the report *I*. Some of the pairs of original report O and its duplicate report D have one of the following inconsistencies.

- If the *Component* field of all duplicate reports of original report O are not the same and the *Component* field of the original report O is empty. Still, all duplicate reports of the original report O contain the same set of groups of components G_D. We set the *Component* field of the original report O to the value of the *Component* field of duplicate reports.
- If the *Component* field of all duplicate reports of original report O are not the same and the *Component* field of the original report O is not empty, then each pair of original report O and duplicate report D, we check whether it satisfies at least one of the conditions. If it is satisfied, then we set the *Component* field of duplicate report D to the value of the *Component* field of original report O
  - All the components of duplicate report D (C_D) in the set (C_O) or the set (G_O).
  - All the groups of the components of duplicate report D (G_D) present either in the set (C_O) or the set (G_O).

Firefox We followed the general approach we mentioned at the beginning of this section Section 3.5.

3.6 Earliest Report and Fixed Timestamps

First reported Timestamp ($T_{earliest}$) : the date and time of the first report of a vulnerability. For each valid original report O, we compute $T_{earliest}$ based on either one of the conditions:

- If the valid original report O reported before all of its duplicate reports, then we set $T_{earliest}$ to the value of *OpenedTimestamp* field of the valid original report O.
- If the valid original report O is reported after one or more of its duplicates, we list all the timestamps from the *OpenedTimestamp* field of all the duplicate reports of the valid original report O, then set $T_{earliest}$ to the minimum timestamp from the list of all timestamps.

Fixed Timestamps In Firefox In collected data, there are 2 vulnerabilities that both original and its duplicate has fixed time. We excluded those reports from our analysis related to fix. For some reports with *WORKSFORME* status in

Firefox, there are not specific fixed time. If that report has *TrackingFlags* in its information page and it shows that the report got fixed in a version, we consider this report in our dataset as valid report but we do not consider its fixed time.

3.7 Rediscovery

There are some reports that are reported by the same origin reporter multiple times in both Chromium and Firefox. In the rediscovery analysis, we remove redundant reported reports and keep only the earliest report from the same origin. There are also reports that do not have accessible pages in Bugzilla. Since for those reports we cannot identify which report is the earliest one, we excluded them from the rediscovery analysis. In Firefox, some reports are incomplete with respect to replication and patching. For some of them, Mozilla opened a new report of the vulnerability, which was then completed with respect to this information, and marked the first report as a duplicate. We also exclude these vulnerabilities from our analysis since they are not actual rediscoveries.

3.8 Exploited Vulnerabilities

To identify exploited vulnerabilities, we first collect an initial set of exploited vulnerabilities from the Known Exploited Vulnerabilities Catalog of CISA[13]. Then, we extend this set iteratively using a snowballing method by identifying terms in comments related to exploitation (e.g., *exploited in the wild*) and gathering vulnerabilities whose comments include these terms. We manually verify the descriptions of these vulnerabilities to find false positives. Finally, we restrict the set to valid original security vulnerabilities, resulting in a set of 18 and 37 exploited vulnerabilities for Firefox and Chromium respectively.

[13] https://www.cisa.gov/known-exploited-vulnerabilities-catalog