Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 170 (2020) 618–625

Procedia Computer Science

www.elsevier.com/locate/procedia

The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40), April 6 - 9, 2020, Warsaw, Poland

Software Release Patterns
When is it a good time to update a software component?

Solomon Berhe[a,*], Marc Maynard[b], Foutse Khomh[c]

[a] *releasetrain.io, Stuttgart, 70376, Germany*
[b] *Data Independence LLC, Ellington, 06029, Connecticut, United States*
[c] *Polytechnique Montréal, Montréal, QC, H3C 3A7 H3C 3A7, Canada*

Abstract

Over the past decade the industry 4.0 witnessed a trend towards an increasing number of software components, dependencies towards third party software components, and software component release cycles. Industry 4.0 teams building software products are more frequently impacted by third party software component updates. Due to this dependency, updating a single third party software component can break an entire software product. Reasons include parallel conflicting updates of third party software components, updating to an unstable version, or updating to a major stable version without an impact analysis. The objective of this paper is to reduce the risk of breaking updates by reviewing software release patterns and proposing update scheduling recommendations.

© 2020 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.

Keywords: Software Updates, Software Dependencies, Software Release Schedule

1. Introduction

Many sources estimate that the number of connected devices is going to increase from 15 billion in 2015 up to estimated 40 billion by 2020 and 75 billion by 2025 [19]. Two main reasons for this trend are the evolution of the internet of things (IoT) and quantified self (QS) [17, 8]. In both areas more and more devices get connected to servers. For example, with the help of connected scales, patients are able to automatically send their daily weight to clinical servers, which automatically checks patient health status. This trend not only affects the clinical domain, but also the mobility domain, the smart home domain, the smart city domain, and many more domains that build connected software products.

* Corresponding author. Tel.: +49-157-5240011.
E-mail address: solomon.berhe@gmail.com

1877-0509 © 2020 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2020.03.142

With the number of software components increasing, their usage as part of third party software components has increased as well. Over the past decade, customers expect a base set of features and functionalities next to a product's core service feature. Examples include easy sign-up using an existing account, standardized user interface, mobile application versions for Android and iOS, automatic data synchronization, or legal compliance. To meet these types of requirements, it is often faster to reuse existing industry standard third party software components. Another usage of third party software components derives from the interaction with an application programming interface (API) to external software components. For example, mobile web applications often run on an underlying browser version, operating system version, firmware version, or sensor version [14]. However, these third party software components can be updated independently of the main mobile web applications. This complicates maintaining a stable software product. While third party software components often allow for faster software development, their maintenance costs are continually rising [20].

Next to an increasing number of software components and their dependencies, the third software update challenge over the past decade is the trends towards faster release cycles [16]. Many software components have shorten their release cycles and release a version, for example, on a monthly basis. From an agile development team point of view, the motivation is often the need to respond more quickly to changing customer requirements and to build a stable product by testing it early in the market [18].

If the three trends are considered together, a rising number of dependent software components paired with faster release cycles lead to more security updates, version upgrades, version downgrades, or version deprecation [2, 1]. This increased version release volatility automatically increases the risk of a single version release update breaking an entire software product [9, 20]. To illustrate this issue, below is a sample of real user feedback regarding software updates:

- "I put the iOS 12 beta on my phone and it stopped receiving calls. It was a major pain to downgrade, too. iOS 13 hasn't been terrible on my iPad but I wouldn't recommend updating anything you rely on!" [21].
- "Upgraded to Rails 6.0.0.rc1 and all tests are passing except a few that depend on Active Job executing jobs inline. It seems like http://config.active job.queue adapter = :inline' has no effect whatsoever. Has anyone else experienced the same?" [22].
- "CVE-2018-15664: docker (all versions) is vulnerable to a symlink-race attack" [23].
- "Seems like Retrofit 2.7.0 will require API 21 - Burning bridges down, huh." [26].
- "sigh. Xcode 11.2 is already deprecated due to some serious bugs. I spent ages trying to get it like 3 weeks ago due to another bug in the osx App store app where xcode wouldn't even download. Now back to square 1 with 11.2.1 and the app store app again." [25].
- An Apple user recommending not to update three dependent software components at the same time [24].

These examples show that updating a single software component can result in breaking parts of a software product or an entire product. The decision of when to update a software component is therefore becoming very critical. Some agile software product teams decide not to update a software component unless it is required and others decide to update software components in a more intuitive manner [10, 15]. With faster release cycles, software components will be deprecated faster and efforts to fix delayed or intuitive updates is only postponed until the software product breaks. As a result, two tasks are required for each update (1) an early and cautious evaluation of the impact of the update on its dependent components [5, 12] and (2) based on the evaluation result, a coordinated update of all affected software components [9, 13]. Performing these tasks is not without risks [24]. Generally, the evaluation and coordination of updating a single software component is becoming more and more difficult and time consuming. For software products that depend on many third party software components, it is practically impossible to comprehensively evaluate the impact of single component version updates [12, 13]. This paper describes an approach that aims to help development teams reduce the risk of breaking updates. The proposed approach leverages software component update release patterns. A rash and ill-timed update to a breaking software component version often leads an agile team to re-prioritize a planned sprint backlog. To propose a more risk free update time, this work looks at the aggregate release distribution of many software components, and to what extent potential software release patterns are helpful in deciding when the risk is lower to update to a non-breaking software component version.

The remainder of this paper is organised as follows. Section 2 presents the data and setup used for the work in this paper. Section 3 presents the questions to be answered, that may assist in recommending when to perform a software component update. In Section 4 the results will be presented. Section 5 will propose recommendations based on the results. Section 6 concludes the paper with a look ahead to future research.

2. Data Setup

To potentially assist industry 4.0 software product teams with decisions of when to perform a software component update, release patterns of software component releases in general may be of assistance. To perform the software release pattern study, software release databases of components that are very frequently used and are very up-to-date significantly improve the quality of the study results. The data setup in this section contains three parts, including 1) the data sources; 2) the data scope; and 3) the resulting data set.

2.1. Data Source

As the number of software components is increasing the number of release channels are increasing as well. Presently, software component releases are accessible via app stores, meetings, e-mails, firmware web sites, product websites, Slack channels, Skype, software repositories, blog posts, posts, Common Vulnerabilities and Exposures (CVE) sites, among many others.

For the work in this paper, three release channel criteria were critical. First, it was critical to access releases from channels that include software components that are used from as many software teams as possible. Second, it was important to access releases from channels that support automation, such as an API or direct access to the database. Third, it was also critical that the channel updates releases on a daily basis.

As a result, the releases used are extracted from two channels. For beta or production releases of open source projects, the data is extracted from GitHub [7]. For CVE release update the National Institute of Technology (NIST) National Vulnerability Database (NVD) was picked for data extraction [11, 10]. The data source for this work can be accessed at `https://releasetrain.io/docs/paper/edi40_2020/versionDb.json`.

2.2. Data Scope

In this work release data ranging from January 1, 2018 to July 3, 2019 was extracted. Within this time frame, the GitHub release data was further limited to the 100 most used open source projects.

While both data sources store the data in JavaScript Object Notation (JSON), they each have different schemas. Applying an extract, transform, and load (ETL) process, data from both sources is transformed into a single release version database. The database schema supports the following core attributes critical for this work [6]: versionNumber, versionProductName, versionReleaseDate, versionReleaseChannel.

2.3. Data Set

After extracting the release data (Section 2.1) and transforming it (Section 2.2), the resulting data set included 3,742 total software components and 8,537 software component version releases. The software component releases are distributed by the following channels: Beta (375), Prod (1202), CVE (6956).

3. Relevant Questions

This section will review the types of questions that may be of assistance when determining when to update a software component. The questions will not address a single software component pattern, but software components in the aggregate in order to potentially identify general patterns. The questions are primarily proposed by the authors, since their answers would have helped them to better schedule software updates in many previous projects. Towards this goal, the first question will look at the temporal aspect and distribution of software component releases by different time intervals. The second and the third questions will look at potential release channel patterns to better determine and plan the stability of a software component version.

3.1. Question 1: Releases by time intervals?

The first question will look at the release distribution by the day of the week, day of the month, and month of the year. Depending on the result, it may determine when updating a software component is the least risk free. For example, if an agile software product team decides to update a software component on a Monday, but on the same day, many third party software components also decide to perform a software update, then the risk of an update collision is higher and the root cause is less clear. Resolving such a collision requires a cross software component technical expert team evaluation, which is often difficult and time consuming to establish [12].

3.2. Question 2: How many versions are released before a stable release?

Some agile teams postpone updating a software component until it breaks the software product [10]. Other agile software product teams update a software component as soon as it is available. The first approach is risky, as by the time an update is mandatory, the impact of that update is likely very high [15].

3.3. Question 3: How reliable are major releases forecasts?

Ideally, agile software product teams release their version updates in regular intervals and with release channel rules. Such software component allow consuming agile software product teams to reliably allocate resources for a major (breaking) software component update task. However, sometimes major updates are very unpredictable and therefore must be planned in a more ad-hoc manner. This question will examine the extent to which a correlation exists between the release date, release version, and the release channel.

4. Results

This section will present the results based on the three questions in Section 3.1, 3.2, and 3.3. The goal of this section is to extract as many patterns as possible that are potentially valuable to improve a software component update time. The results will only take software release data into consideration. Other important software update scheduling aspects, such holidays, vacations, sick days, or days off are not taken into consideration.

4.1. Result 1: Releases by day of the week?

Every agile software product team decides when during the week a software update makes the most sense. Some teams may decide that the beginning of the week is a good candidate, as it gives the team the remainder of the week to address potential issues. Moreover, they can monitor the market impact of the update throughout the week. Other teams may decide to update a software component on Friday, so they have more time to run internal tests and evaluate the impact during the weekend. Moreover, if the software product is less used during the weekend and more during the week, then a potential crash could be fixed before Monday.

Each agile software product team has product specific criteria of when to perform a major software update. However, when one only looks at the number of release by the day of the week, one gets the result in Figure 2. Given the release data, in 2018 and 2019, most software updates were released on Wednesday (245 major, 78 beta) and Thursday (1483 CVE, 253 prod). The least software updates were released on Saturday (51 prod, 25 beta, 265 CVE) and Sunday (50 major).

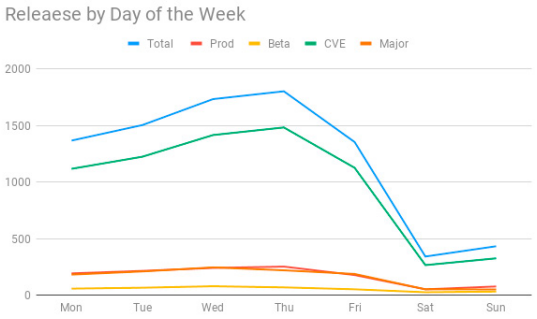

Fig. 1: Software component releases by day of the week.

4.2. Result 2: Releases by day of the month?

Many agile software product teams plan their backlog at the beginning of the month and run a two or four week sprint. Given the overall release data, when can they expect a high volume of update releases during the month? Towards this goal, the results show that on average the number of releases is the highest in the middle of the month (2129 releases) and the lowest at the end of the month (1721 releases). The highest number of updates are available on the 17th of the month (363) and the lowest number of updates are available on the 27th of the month (205). For more details view Figure 3 and Table 3. The behavior of the total number of releases is also reflected in the various subsets of releases (beta, prod, major, CVE).

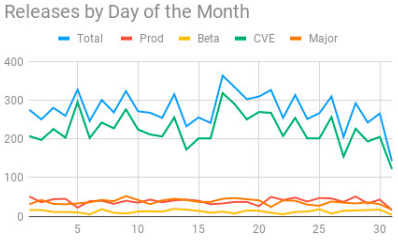

Fig. 2: Software component releases by day of the month.

4.3. Result 3: Releases by month of the year?

Some agile teams prefer to plan product updates a year ahead in order to early align dependencies with other teams, customers, and potentially reduce continuous update bug fixes as much as possible. Aligning the updates with the market release update volatility may reduce the risk of conflicting releases tremendously. The results in Figure 4 and in Table 4 show that the number of releases are the highest in the first quarter and the lowest during the third quarter.

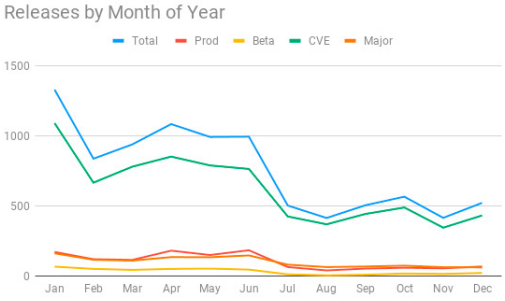

Fig. 3: Software component releases by month of the year.

4.4. Result 4: How many releases before stable?

The results in 1-3 focus on the number of releases in the aggregate without looking into the evolution of a specific software component type. The results here will look at patterns regarding release cycles of software components types. The minimum number of release cycles in this result is four which was met by 423 components (11,3%).

The first result shows that a CVE tremendously affect the stability of software components. A single CVE easily affects multiple major versions of the same software component. For example, an Ubuntu Linux CVE affects four major versions ranging from 12.0.0 - 18.0-0. A Debian Linux and a Linux kernel CVE affects three major versions ranging from 7.0.0 - 9.0.0 and 2.0.0 - 5.0.0. The same applies for other widely used more infrastructure level software components such as PHP, PHPMyAdmin, Go, Python, NX-OS, iOS, FreeBSD, Solaris, MySQL, PostgreSQL, DB2, Wordpress, Jenkins, JDK, CURL, and many others. One option to address this type of CVE stability is to update a software component to its latest version. However, updating to the latest version includes the risk that this version will be rolled back as it is not stable enough.

The second result looks at the correlation between the release channel and the version number (major and minor versions). The result is that many components map the first major and minor version to a beta channel release. Examples include operating systems, such as Linux, iOS, Android; databases, such as MySQL, SQLite, and application frameworks, such as caddy, ember.js, react, vue, electron, kubernetes, webpack. For many of these software components, a stable production version will be released after several beta versions. Therefore, updating to the latest beta version may result in bugs that do not require any action and will be automatically fixed in the next versions.

The third result shows that for some components, once a certain software version number is reached, then only stable production releases are performed. These types of components have more the scope of a library and less of an entire layer or tier. Examples for such software components include: laravel (v5.8), shadowsocks-android (v4.8), bootstrap (v4.3), electron (v7.0), next.js (v8.0, and many more.

4.5. Result 5: How reliable can releases be forecasted?

One way to generally forecast major releases is by extrapolating the previous software component releases by looking at various time intervals (see Results 1-3). Another option is to look at release patterns of component types and potentially identify release patterns. Towards this goal, this result will look at operating systems, databases, programming languages, and libraries. Operating systems have a reliable schedule with regard to releasing minor and major versions. This release cycle can potentially be affected by CVEs, but is still very reliable. On the other hand, forecasting releases from databases, such as MySQL, SQLite, or programming languages and libraries is not very reliable.

5. Recommendations

Given the release pattern results in section 4, this section will derive update time recommendations for agile software product teams. Generally, the more third party software components are included in a software product and the

more these third party software components are updated independent of the product team, the more these recommendations may be of value. The first three recommendations are prerequisites for an update to be reliably scheduled. They are parallel to software security plans and are borrowed for regular software releases. The last four recommendations focus on the update schedule and the update order.

5.1. Recommendation 1: Document dependent third party software components.

Before an update can be scheduled, compiling an overview of all third party components and their current versions and other details, allows an agile team to more reliably track and schedule updates. As a result, documenting all dependent third party software components is the first recommendation. An overview of the system version baseline could be maintained by the software architect.

5.2. Recommendation 2: Monitor dependent third party software component updates.

An update of a third party component may have an impact on the entire product. Given the number of dependent third party software components, monitoring updates on a regular basis is the second recommendation. Once updates are monitored they can be reliably scheduled. The monitoring can be done by looking up each update channel individually or by looking a single releasetrain.io dashboard.

5.3. Recommendation 3: Analyse dependent third party software component updates.

A third party software component update may range from no impact to the software product crashes or legal consequences. As a result, as soon as an update is announced, it should be technically analysed by experts.

5.4. Recommendation 4: Schedule updates around business cycles.

The first update schedule recommendation focuses on business cycles. To reduce the number of conflicting updates, the goal is to schedule updates, such that the risk is lower that they collide with other conflicting updates. The results in Section 4 show that towards the end of the week, the end of the month, and the end of the year, the number of updates is the lowest. As a result, in general, the risk of conflicting updates during these business cycles is the lowest.

5.5. Recommendation 5: Schedule updates to stable version.

As seen in Section 4, for many software components to reach a stable version, they are first tested in the market. Updating to an early release increases the risk of breaking a product due to bugs. Therefore, scheduling updates to stable version reduces the risk of breaking product due to bugs of dependent components.

5.6. Recommendation 6: Schedule future updates to stable versions.

For a limited set of software components, updates follow release guidelines that including strict release cycles. Examples include operating systems such as Linux, iOS, RIOT-OS, and others. Scheduling future updates ahead of time allows early resource allocations and timely impact analysis.

5.7. Recommendation 7: Schedule updates along dependency path.

In some cases, multiple software components must be updated around the same time [24]. Here it is recommended to update one component at a time starting from the component that has the most dependencies (likely on critical path) to the components with the least dependencies (unlikely on critical path). That way potential bugs that are discovered between the testing are fixed before one updates the next component, resulting in reduced impact of bug fixes from a component dependency point of view.

6. Conclusion

This paper presented the study of software update release patterns and the derived recommendations. Given that the number of software releases is going to increase, the demand of carefully scheduling updates will likely increase as well. Therefore, besides software product specific aspects, considering recommendations from general software update release patterns may further reduce the risk of breaking software product update. The future work includes a more granular look at software component type, topic patterns and extracting more valuable recommendations, and machine learning models to predict optimal release windows.

7. ACKNOWLEDGMENTS

We would like to thank NIST NVD and GitHub for providing public access to the software updates release data. We would like to thank [21, 22, 23, 24, 25, 26] for their valuable public feedback on released software versions.

References

[1] D. A. da Costa and S. McIntosh and U. Kulesza and A. E. Hassan. (2016) "The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues - An Empirical Study of the Mozilla Firefox Project." *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* 374–385.
[2] CVE-2019-2043. (2019) "CVE-2019-2043.", in https://nvd.nist.gov/vuln/detail/CVE-2019-2043.
[3] CVE-2019-3896. (2019) "CVE-2019-3896.", in https://nvd.nist.gov/vuln/detail/CVE-2019-3896.
[4] CVE-2019-12817. (2019) "CVE-2019-12817.", in https://nvd.nist.gov/vuln/detail/CVE-2019-12817.
[5] Dingsøyr, Torgeir and Nerur, Sridhar and Balijepally, VenuGopal and Moe, Nils Brede. (2012) "A Decade of Agile Methodologies." *Journal System Software* **85** (6): 1213–1221.
[6] El-Sappagh, Shaker H. Ali and Hendawi, Abdeltawab M. Ahmed and El Bastawissy, Ali Hamed. (2011) "Original Article: A Proposed Model for Data Warehouse ETL Processes." *Journal King Saud University Compututer Information Science* **23** (1): 91–104.
[7] GitHub. (2019) "Showing available repository results.", in https://github.com/.
[8] Khakurel, Jayden and Immonen, Mika and Porras, Jari and Knutas, Antti. (2019) "Understanding the Adoption of Quantified Self-tracking Wearable Devices in the Organization Environment: An Empirical Case Study." *Proc. of the 12th ACM International Conference on PErvasive Technologies Related to Assistive Environments*.
[9] Krishnan, Mayuram S. (1994) "Software Release Management: A Business Perspective." *Proc.ings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*.
[10] Kula, Raula Gaikovina and German, Daniel M. and Ouni, Ali and Ishio, Takashi and Inoue, Katsuro. (2018) "Do Developers Update Their Library Dependencies?." *Empirical Software Engineering* **23** (1): 384–417.
[11] Li, Frank and Paxson, Vern. (2017) "A Large-Scale Empirical Study of Security Patches." *Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security*.
[12] Ma, Wanwangying and Chen, Lin and Zhang, Xiangyu and Zhou, Yuming and Xu, Baowen. (1994) "How Do Developers Fix Cross-project Correlated Bugs?: A Case Study on the GitHub Scientific Python Ecosystem." *Proc. of the 39th Intl. Conf. on Software Eng.*.
[13] Mougouei, Davoud and Powers, David M. W. and Moeini, Asghar. (2017) "Dependency-aware Software Release Planning." *Proc. of the 39th International Conference on Software Engineering Companion*.
[14] NIST NVD. (2019) "National Vulnerability Database.", in https://semver.org/.
[15] Sassenburg, Hans and Berghout, Egon. (2006) "Optimal Release Time: Numbers or Intuition?" *Proc. of the 2006 International Workshop on Software Quality*.
[16] Skeet, John. (2019) "Versioning limitations in .NET.", in https://codeblog.jonskeet.uk/2019/06/30.
[17] Peter Shaw, Mateusz Mikusz, Petteri Nurmi, and Nigel Davies. 2019. IoT Maps: Charting the Internet of Things. Proc. of the 20th Intl. Workshop on Mobile Comp. Sys. and Appl. (HotMobile '19). ACM, New York, NY, USA, 105–110.
[18] Shore, Jim. (2004) "Fail Fast." *IEEE Software* **21** (5): 21–25.
[19] Statistica. (2019) "Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions).", in https://statista.com/statistics/471264/iot-number-of-connected-devices-worldwide.
[20] Tanabe, Yudai and Aotani, Tomoyuki and Masuhara, Hidehiko. (2018) "A Context-Oriented Programming Approach to Dependency Hell." *Proc. of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition*.
[21] Twitter. (2019) "Software Update Finding.", in https://twitter.com/michaeltbuss/status/1139659750105333764.
[22] Twitter. (2019) "Software Update Finding.", in twitter.com/andrewculver/status/1133375191961210881.
[23] Twitter. (2019) "Software Update Finding.", in twitter.com/GloriaPalmaGlez/status/1133494969661050881.
[24] Twitter. (2019) "Software Update Finding.", in twitter.com/marcoarment/status/1140683983447089154.
[25] Twitter. (2019) "Software Update Finding.", in https://twitter.com/overjeer/status/1203463879537283072.
[26] Twitter. (2019) "Software Update Finding.", in https://twitter.com/arturdryomov/status/1147947575917666305.
[27] Tom Preston-Werner. (2019) "Semantic Versioning.", in https://nvd.nist.gov/.