(Hrsg.): GI SICHERHEIT 2022,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2022 1

PrivacyDates: A Framework for More Privacy-Preserving Timestamp Data Types

Christian Burkert[1], Jonathan Balack[2], Hannes Federrath[3]

Abstract: Case studies of application software data models indicate that timestamps are excessively used in connection with user activity. This contradicts the principle of data minimisation which demands a limitation to data necessary for a given purpose. Prior work has also identified common purposes of timestamps that can be realised by more privacy-preserving alternatives like counters and dates with purpose-oriented precision. In this paper, we follow up by demonstrating the real-world applicability of those alternatives. We design and implement three timestamp alternatives for the popular web development framework Django and evaluate their practicality by replacing conventional timestamps in the project management application Taiga. We find that our alternatives could be adopted without impairing the functionality of Taiga.

Keywords: Privacy by design; data minimisation; timestamps

1 Introduction

The design of software is today probably one of the biggest factors for everyday privacy. Since using software becomes virtually inescapable, it is increasingly application data modelling that decides how much of our personality and about our activities is recorded. Previous work [BF19] indicates that data models make excessive use of timestamps, the data type that adds the particularly sensitive temporal dimension to profiling. Timestamps have been previously observed to fulfil various functions in programming that not even require temporal properties. Instead, timestamps are frequently used for ordering or determining state (e. g., maintain order in which attachments were added). Function that can easily be achieved with less privacy-invasive alternatives. But also in cases where their temporal functions like universal comparability are actually used (e. g., time a bug report was filed), there appears to be room for a reduction of the typical second or even microsecond precision, to precisions that correspond more with human perception and increase privacy. Tackling the excessive use of timestamps in data models is a matter of raising awareness but also of providing ready to use alternatives. In this paper, we provide and evaluate a first such framework of timestamp alternatives. In summary, we make the following contributions:

[1] Universität Hamburg, christian.burkert@uni-hamburg.de
[2] Universität Hamburg, jonathan.balack@informatik.uni-hamburg.de
[3] Universität Hamburg, hannes.federrath@uni-hamburg.de

- We design more privacy-preserving alternatives for common use cases of timestamp data types as identified by prior work.
- We validate the design applicability with a case study of the application *Taiga*.
- We provide an implementation for the popular web application framework Django.
- We evaluate and demonstrate the practicality of those alternatives by replacing timestamps in data model of Taiga with our alternatives and observe the effects.

The remainder of the paper is structured as follows: We firstly present related work and our adversary model, then we describe the design and implementation of our alternatives, after which we provide an evaluation.

2 Related Work

In a prior case study of the Mattermost application, we systematically analysed the usage of personally identifiable timestamps in data models [BF19]. We found that timestamps of creation, last modification and deletion are included in a majority of models. However, most user-related timestamps were found to have no programmatic use and only a small fraction is displayed on the user interface. Based on the identified functions of timestamps, we proposed design alternatives that use precision reduction and context-aware counters. Otherwise, the literature on timestamp-related privacy patterns and practical data minimisation is scarce. In 2017, a literature survey of privacy patterns by Lenhard et al. [LFH17] showed that proposals are rarely verified or even implemented. Strategies to reduce the sensitivity of timestamps have been proposed by Zhang et al. [ZBY06] for log sanitization. They discuss *time unit annihilation* as a strategy to gradually reduce precision over time.

3 Adversary Model

To contextualise privacy gain through our more data-minimal timestamp alternatives, we provide the following adversary model. It follows the established honest-but-curious (HBC) notion commonly used to assess communication protocols. Paverd et al. [PMB14] define an HBC adversary as a legitimate participant in a communication protocol, who will not deviate from the defined protocol but will attempt to learn all possible information from legitimately received messages. Following the adaption of this model to the context of application software and data models [BF19], we consider an adversary to be an entity that is in full technical and organisational control of at least one component of a software system, e.g., the application server. The adversary will not deviate from default software behaviour and its predefined configuration options, but will attempt to learn all possible information about its users from the data available in the application. This especially means that an

adversary will not modify software to collect more or different data, or employ additional software to do so. However, an adversary can access all data items that are collected and recorded by the software system irrespective of their exposure via GUIs or APIs. We reason that this adversary model fits real world scenarios, because software operators in general lack the technical abilities to modify their software systems or are unwilling to do so, to not endanger the stability of their infrastructure or to not document potentially illegal behaviour. We come back to this adversary model when we employ server-side reduction later on.

4 Design

Based on alternative concepts for timestamps in the literature, we designed three data types: a generalised date with a static precision reduction (*rough date*), a context-aware counter for chronological ordering (*ordering date*), and a generalised date with temporally progressing precision reduction (*vanishing date*). The designs are targeted as replacements for the conventional timestamp data type in the Django framework, but are using only standard database features typically available in development frameworks. The following describes the design for each alternative type.

4.1 Type 1: Rough Date

Rough date is a variation of Django's standard `DateTimeField` that truncates the date to a given precision. As such, it should offer all functionality that `DateTimeField` does and maintain the same interface, to be usable as a drop-in replacement. The desired precision is given as a mandatory argument at field initialisation, either in seconds or in the style of the `timedelta` class from Python's standard library package `datetime` [Py21] as multiples of the units minutes, hours, etc. The following creates a rough date with one hour precision: `RoughDateField(hours=1)`. We deliberately do not provide a default precision to force users to consider the necessary precision for their given use case. The date part below a given precision is truncated.

4.2 Type 2: Ordering Date

The ordering date is an alternative to using timestamps for ordering items chronologically, if absolute date references and relative distances are not actually needed. `OrderingDateField` offers ordering via context-specific auto-incremented counters. Consequently, ordering date requires that objects are inserted in chronological order. As shown in Fig. 1, a context-defining string key is given for each `OrderingDateField` at model initialisation. The context label is cryptographically hashed to a 256 bit key which then uniquely identifies its corresponding `OrderingContext` which persists the actual counter state information. This

way of maintaining the relation between ordering dates and their contexts in code and not in the database is space-efficient and allows for dynamic contexts keys. And since the context label is given at initialisation, ordering contexts can be defined very flexible. For instance, a label can comprise a username and thereby create an isolation between counter contexts of different users. This can be used to increase user privacy by avoiding an otherwise global counter context that would make instances with ordering date temporally chronologically comparable across users.

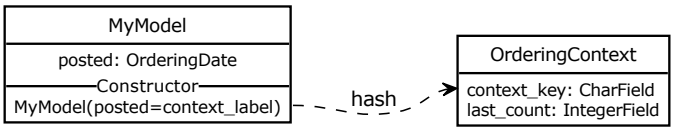

Fig. 1: Class diagram showing a user-defined model with `OrderingDateField`. The related `OrderingContext` is identified by a context label given as initial field value. The integer value of `OrderingDateField` is then set to the context's next count.

4.3 Type 3: Vanishing Date

Vanishing date implements the privacy pattern of *time unit annihilation*. This alternative offers a progressing reduction of precision according to given increments until the end precision is reached. For each step, a precision is provided like for `RoughDateField` in combination with a temporal offset, i. e., the distance from object creation that marks when the reduction step is due. A background process regularly checks for due reductions and applies them. List. 1 shows an example of a vanishing date with a three step reduction policy, the first of which is immediately on creation, whereas the second and third follow after a given time. Tab. 1 lists the resulting stored date and next reduction event for each step.

```
created_at = VanishingDateField(policy=make_policy([
  Precision(hours=1),
  Precision(days=1, after_hours=3),
  Precision(months=1, after_days=7),
]))
```

List. 1: Construction of a vanishing date with a three step reduction policy ranging from initially 1 hour to finally 1 month precision after 7 days. Helper `make_policy` ensures correct reduction progression.

Step	**Current Time***	**Stored Date**	**Next Due Date**
Creation/1st Red.	2021-11-08 15:17	2021-11-08 15:00	2021-11-08 18:00
2nd Reduction	2021-11-08 18:01	2021-11-08 00:00	2021-11-15 00:00
3rd Reduction	2021-11-15 00:03	2021-11-01 00:00	-

Tab. 1: Exemplary progression of vanishing date reduction with a 3-step policy leading to a precision of one month after seven days. (*Current times depend on the frequency and delay of periodic checks.)

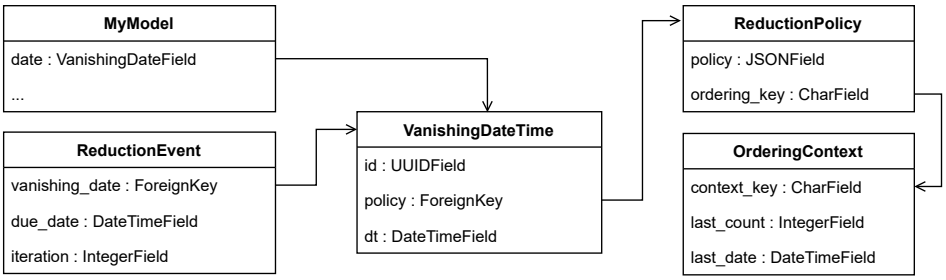

Fig. 2: Class diagram showing a custom model that uses `VanishingDateField` which references a `VanishingDateTime` object specifying the information about the reduction policy and events.

As shown in Fig. 2, the resulting design of vanishing date is more complex than for rough date and requires auxiliary models to persist information about the reduction policy and the current progress within that policy. Therefore, `VanishingDateField` sets a reference to a `VanishingDateTime` class that holds the actual, gradually reduced date, a reference to a policy instance, and to a `ReductionEvent` that represents the next due reduction step. All instances of `ReductionEvent` form a queue that can be efficiently processed by the periodic due check. Since Django lacks the ability to natively trigger periodic tasks, we offer a management command for periodic reduction that can be triggered via, e. g., *Cron*.

Note that to not leave any traces of the previously truncated time information, due dates for subsequent reduction steps are calculated on the basis of the reduced step. In Tab. 1, for instance, the second reduction is due at 18:00 instead of 18:17, to not thwart the reduction to hour precision in the first step. As a result, the time periods given between each policy step are upper boundaries. In the previous example, the hour precision is available for 17 minutes less than the full three hours given in the policy. Also note, the reduction level of a step should not be larger than the offset of the following step.

Following our adversary model in Sect. 3, the application itself holds the only record of the timestamp reduced by vanishing date. This especially means, that the software operator does not keep a mirror of the information before reduction or of earlier reduction levels.

4.4 Design Validation

The purpose of design validation is to examine whether the design mainly based on previous case studies also holds for the application we selected to evaluate our implementation. As described below, we inspected its timestamp usage following the methodology of [BF19].

Taiga is a project management software that is built on the Django framework. Taiga focuses on user interaction like the creation, processing and commenting of tasks to control and document the progress of projects. Following the agile approach, interactions occur around

planning elements like tasks, issues, epics, sprints and user stories. We chose Taiga for our evaluation because it is a popular app built on Django and it is focused on structuring and recording user interactions, which likely brings sufficiently complex requirements to test our implementation. In the following, we describe our methodology, the identified timestamp use, and any necessary modifications to our design.

4.4.1 Methodology

Following [BF19], we examine the source code of Taiga's back-end component [Ta21a] for occurrences of Django's date-related model fields `DateTimeField`, `DateField`, and `TimeField`. We assess semantic and purposes of each timestamp by examining all of their uses by the back-end, their presentation in the front-end [Ta21b] and their exposure via the API. We then use the identified purposes to select from our proposed types an alternative that provides the required functionality. If none should be available, our design would need refinement. We find that the back-end REST API typically returns every attribute (model field) related to the requested object, regardless of whether the front-end uses them or not. Therefore, it is not sufficient to access timestamp usage and purpose only based on API exposure, but actual use based on inspections of the rendered front-end are necessary. To do so, we manually examined Taiga's front-end user interface cataloguing presented timestamp information. Usage in the back-end was assessed by manually inspecting all occurrences of date-related field names throughout the back-end code. These analyses were conducted on version 6.0.7 of both back-end and front-end, as released on March 8th, 2021.

Model Field	Occurrences	Used in Models	Exposed via API
DateTimeField	170	48	41
DateField	15	3	3
TimeField	5	0	0

Tab. 2: Usage of date-related Django model fields in Taiga's back-end.

4.4.2 Identified Timestamps

Tab. 2 shows the numbers of identified uses per model field. In total, we located 190 occurrences in Taiga's back-end code of which 51 are part of data model definitions. The remaining matches occurred in database migrations and serialization code. We did not further inspect the latter occurrences as they do not contribute any usage and purpose information. Almost all definitions use the `DateTimeField`. Only 3 (6 %) use the `DateField`, whereas `TimeField` is not used in current model definitions at all.

To assess the API exposure, we inspected the source code and consulted the official API documentation for information about which fields are included in a query response. We found that all but 7 timestamps (86 %) are exposed through the API. Of those 7 timestamps,

5 are also not programmatically used on the back-end. The visual inspection of the front-end UI also revealed that at least 22 (50 %) of the timestamps fetched from the back-end API are not used there. Regarding those timestamps without a detectable usage or purpose, we can not determine a purpose-appropriate alternative. For the sake of data minimisation, they should be removed entirely. Also, not all timestamp fields are necessarily personal data. This is true for the three `DateField` uses, which are used to model due dates of planning elements (e. g., sprints) which are not directly linked to actions of users. Hence, we omit those from classification as well. For the remaining timestamps, we classified their type purpose according to their usage context in back-end and/or UI.

4.4.3 Timestamp Semantic and Purpose Classification

We classified the remaining timestamps based on the semantic given by their variable name and source code context. All models in Taiga (27) have a creation timestamp to automatically capture when a model instance was created. 17 models additionally record the time of the latest update, three the time a planning element was completed, and one when a notification was read. Regarding purpose classification, we followed [BF19] and used a bottom-up classification that inspects and labels each timestamp's programmatic use in the source code with respect the function they serve in the respective context, resulting into similar purposes: presentation for user information, sorting, and comparison. Tab. 4 in the appendix lists the identified purposes for each timestamp.

4.4.4 Design Revision

Based on the identified purposes and functional properties of our proposed alternatives, we select possible replacements for each timestamp. The selections are shown in Tab. 4 (appendix). If a timestamp is only used for sorting like the creation date of `Attachment`, the ordering date is the apparent alternative. Timestamps with a presentation or comparison purpose can equally be replaced by rough date and vanishing date. The latter should be chosen if an initial higher demand for precision exists.

We find that our proposed alternatives cover all found purposes. However, we noticed that the purpose of maintaining temporal order sometimes coincides with providing a temporal context (presentation or comparison). To replace such a timestamp, two fields are required in the initial design (e. g., `OrderingDateField` and `VanishingDateField`). Since this would both complicate usage and increase memory footprint, we decided to introduce two additional fields that combine the properties of ordering date with rough date and vanishing date respectively, which otherwise do not maintain order for dates reduced to the same value. To do so without increasing memory footprint, we use the sub-second value range available in most timestamp representations to hold the ordering counter. For a microsecond timestamp this leaves space for a 10^6 counter. The counter is incremented for

all timestamps with identical values in their end-precision (Tab. 3). Note that this approach only works for timestamps that are added in chronological order, e. g., that are automatically set to the current time, otherwise the insertion order would not reflect their temporal order.

Original	**Vanishing** 1. Iteration [5 sec]	**Vanishing+Order** 1. Iteration [5 sec]	**Vanishing+Order** 2. Iteration [30 sec]
12:20:11:673320	12:20:10:000000	12:20:10:000000	12:20:00:000000
12:20:14:313406	12:20:10:000000	12:20:10:000001	12:20:00:000001
12:20:17:248323	12:20:15:000000	**12:20:15:000002**	**12:20:00:000002**
12:20:33:040852	12:20:30:000000	12:20:30:000000	12:20:30:000000
12:20:35:917632	12:20:35:000000	12:20:35:000001	12:20:30:000001

Tab. 3: Sample sequence of vanishing date with and without added support to preserve ordering. The highlighted timestamps demonstrate that counter reset is determined by the end-precision of 30 sec which defines the counter scope.

5 Implementation

We implemented our revised concept as a Django app that can be included into other Django projects to provide our date alternatives. It is available open source on GitHub [EM22]. In the following, we describe trade-offs and limitations of this implementation. Ordering date can simply build on available counter fields and is hence omitted from description.

5.1 Rough Date

To offer `RoughDateField` as a drop-in alternative for `DateTimeField`, it also has to support the options `auto_now` and `auto_now_add`, which automatically set the field to the current time at the moment when the object is saved (not initialised). To support these options, the reduction of precision has to be integrated in the saving process, since at any earlier point the value is not yet defined. To do so, we use pre-save hooks that apply the reduction. As a consequence, date values assigned to `RoughDateField` remain in full precision until saved.

5.2 Vanishing Date

As vanishing date is the most complex type, we face three main implementation challenges.

Avoiding chronology leak with UUIDs By default, Django would use an auto-incrementing integer primary key for `VanishingDateTime` if no other primary key was specified. Auto-incremented integers would however leak information about the temporal creation order of all instances of every model that uses `VanishingDateField`. For instance,

an attacker could learn that user A logged in after user B posted their last comment but before user B closed the issue. To prevent such chronology leaks, we use randomized UUIDs as primary key. It should be noted that databases might still leak the temporal order by exposing the insertion order in certain queries. Future work should investigate options to, e. g., prevent users from executing such ordering queries.

Policy Reuseability As previously shown in Fig. 2, we decided to make `ReductionPolicy` a separate model to allow its reuse among vanishing dates with the same policy. We provide the helper function `make_policy` that transparently ensures policy reuse.

Model Identification with `VanishingDateMixin` An identification of all models that make use of `VanishingDateField` is required for two reasons: Firstly, to ensure a two-way cascading delete of `VanishingDateTime`, and secondly to create an initial `ReductionEvent`. To locate the relevant models, we decided to employ the common mix-in pattern, which uses inheritance to add functionality to a given class or model. Users of `VanishingDateField` have to add the `VanishingDateMixin` as a base class for their model. Thereby, we can automatically find all sub-classing models and register post-delete listeners to ensure a two-way deletion, as well as a post-safe listener to update reduction events.

6 Evaluation

In this section, we evaluate the practicality of our framework and its storage cost.

6.1 Practicality of Taiga Integration

To evaluate the practicality of our alternatives, we modified Taiga version 6.0.7 to use the timestamp replacements identified in Sect. 4.4 and detailed in the appendix. To test the correctness and impact of our modifications, we ran the modified Taiga and inspected the error output as well as the web front-end for potential negative effects. To check functional integrity, we created and modified various planning elements and compared the visible front-end behaviour before and after integrating our alternatives.

We found that all timestamps could be replaced as suggested, with few adjustments to the code base. As expected, rough date required the least effort of only replacing the field type. More effort was needed for the other alternatives: We used vanishing date and ordering date to each replace timestamps in three models. Since both replacements do not behave like a standard `DateTimeField`, all code that accessed or modified the field value had to be adjusted to either use the reference `VanishingDateTime`, or an appropriate context label instead. After these modifications, Taiga operated normally and we were able to create and modify elements as usual without functional impairments visible through UI or error log.

When it comes to presenting the replacements in the UI, rough date and vanishing date can mostly be treated like standard dates. However, to avoid confusion or wrong expectation of precision, the formatting of both should be adopted to reflect their precision. In contrast, ordering date can no longer be presented as a date in a meaningful way. But if the timestamp previously only fulfilled ordering purposes this should not be an issue. Otherwise, vanishing date might be a more fitting replacement.

6.2 Storage Cost

The following assesses storage cost compared to ordinary date and time (8 byte) using MariaDB as example [Ma19]. Rough date has the same memory footprint. Ordering date uses a 4 byte counter reducing cost by half. The cost of vanishing date is dominated by three UUIDs, which require 38 byte. Added to two 8 byte date, one foreign key and one event counter (each 4 byte), a total of 138 byte is required for vanishing date, which is 17.25 times the cost of an ordinary date and time. Additionally, usage-dependent storage cost is added for vanishing date and ordering date by their auxiliary models. The number `OrderingContext` used depends on the number of distinct context labels set by developers, and scales with the number of users if individual contexts were used to avoid unnecessary comparability. In MariaDB, each `OrderingContext` requires 44 bytes. Moreover, a variable amount of storage is required for `ReductionPolicy`, which depends on the number of defined reduction steps. To give an overall example, applying the replacements in Tab. 4 increases Taiga's average storage cost per timestamp by about 5 times (not weighted by instance frequency).

7 Conclusion

Excessive, unthought use of timestamps in software data models is a violation of the data minimisation principle and potentially harmful to user privacy. Our case study of the Taiga application not only supports the findings of prior work regarding excessive use, but also in terms of minimisation potential through the use of more-privacy preserving timestamp alternatives. We have presented a framework of alternatives for common timestamp functions and purposes. We demonstrated its practicality by implementing it as a Django app which we then used to replace timestamps in Taiga. Although demonstrated for Django, these alternatives only use standard concepts and can be implemented for other development frameworks. Our evaluation with Taiga revealed that code changes were necessary but limited to adopting changed initialisation and access methods. Additionally, the presentation of timestamp may need adjustment to convey decreased precision levels. Depending on the selection of alternatives, especially the frequency of vanishing date, storage cost might increase noticeably. Where this is nor acceptable, rough date and ordering date can be used with little to no additional storage cost, but without gradual reduction. Our integration test suggests that more privacy-preserving alternatives can be adopted with reasonably low effort. A user study to evaluate their usability with developers is left to future work.

Acknowledgements The work is supported by the German Federal Ministry of Education and Research (BMBF) as part of the project Employee Privacy in Development and Operations (EMPRI-DEVOPS) under grant 16KIS0922K.

References

[BF19] Burkert, C.; Federrath, H.: Towards Minimising Timestamp Usage In Application Software - A Case Study of the Mattermost Application. In: DPM/CBT@ESORICS. Vol. 11737. Lecture Notes in Computer Science, Springer, pp. 138–155, 2019.

[EM22] EMPRI-DEVOPS: django-privacydates, 2022, URL: https://github.com/EMPRI-DEVOPS/django-privacydates, visited on: 01/18/2022.

[LFH17] Lenhard, J.; Fritsch, L.; Herold, S.: A Literature Study on Privacy Patterns Research. In: 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, Vienna, Austria, August 30 - Sept. 1, 2017. IEEE Computer Society, pp. 194–201, 2017.

[Ma19] MariaDB: Data Type Storage Requirements, Apr. 2019, URL: https://mariadb.com/kb/en/data-type-storage-requirements/, visited on: 06/23/2021.

[PMB14] Paverd, A.; Martin, A.; Brown, I.: Modelling and automatically analysing privacy properties for honest-but-curious adversaries, tech. rep., 2014, URL: https://www.cs.ox.ac.uk/people/andrew.paverd/casper/casper-privacy-report.pdf.

[Py21] Python Software Foundation: datetime - Basic date and time types, May 2021, URL: https://docs.python.org/3/library/datetime.html, visited on: 05/14/2021.

[Ta21a] Taiga Agile: taiga-back, Mar. 2021, URL: https://github.com/taigaio/taiga-back/releases/tag/6.0.7, visited on: 03/11/2021.

[Ta21b] Taiga Agile: taiga-front, Mar. 2021, URL: https://github.com/taigaio/taiga-front/releases/tag/6.0.7, visited on: 03/11/2021.

[ZBY06] Zhang, J.; Borisov, N.; Yurcik, W.: Outsourcing Security Analysis with Anonymized Logs. In: Second International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2006, Baltimore, MD, USA, August 2, 2006 - September 1, 2006. IEEE, pp. 1–9, 2006.

A Taiga Timestamp Purposes and Replacements

Model / Timestamp	Presentation	Sorting	Comparison	Replacement
Attachment				
created_date		✔		OD
Epic				
created_date	✔			RD
HistoryChangeNotification				
updated_datetime			✔	RD
HistoryEntry				
created_at	✔	✔		VD+O
delete_comment_date	✔			VD
edit_comment_date	✔			VD
Issue				
created_date	✔			RD
modified_date	✔	✔		RD
Like				
created_date			✔	RD
Task				
created_date	✔			RD
TimeLine				
created	✔	✔	✔	VD+O
User				
date_joined	✔			RD
UserStory				
created_date	✔			RD
Watched				
created_date		✔		OD
WebNotification				
created	✔	✔		VD+O
read		✔		OD
WikiPage				
created_date	✔			RD
modified_date	✔			RD

Rough Date **(RD)** Ordering Date **(OD)** Vanishing Date **(VD)**
Vanishing Date with Ordering **(VD+O)**

Tab. 4: Identified purposes and suggested replacements for used timestamps in Taiga.