New Chrome Password Manager design refactoring

Vadym Doroshenko (dvadym@chromium.org)
Status: Implemented
Last updated: 20 March 2018
This document is public
go/new-cpm-design-refactoring

Notes

Some words that are used in the document
1. CPM - Chrome Password Manager
2. Form parsing - making sense of DOM data and extracting needed for CPM information - username/password etc.

What is bad now

CPM was initially written many years ago. Those times Internet was less dynamic and apparently it was not enough understanding about possible corner cases. As a result there were many implicit assumptions that are not true anymore. Many of them were loosened for the last 10 years. Often it was done by introducing additional “if” in not a best place or hacks. Usually such changes were done to make CPM work in some specific known cases, not fully thinking about general design. It makes some part of code extremely tricky and it’s still hard to get some stuff working.

Notes: for more information about the current CPM architecture can be found in slides,

Examples

1. Form parsing became extremely complicated and entangled, for example the main file for form parsing password_form_conversion_utils.cc has almost 1000 lines and its main function GetPasswordForm has more than 350 lines. It’s extremely hard to extend it and we had many bugs there.
2. In the current architecture CPM can’t use server-side signals for filling, for example CPM sometimes fills password in credit card fields (eg. bug). Also it’s not possible to use the results of server-side username detection classifier.
3. As input field identifiers nameForAutofill is used, which is name if it’s not empty otherwise id. In case when there are non unique or empty identifiers, CPM fails to fill.
4. The filling process is pretty brittle (slide and slide for more details), namely in the Renderer when the credentials are received from the browser, CPM tries to match exact the same form which was found, and there are multiple possible problems - non-unique identifiers (previous item in this list), JavaScript updated something etc.
5. The current filling and crowdsourcing is based on nameForAutofill, so it’s impossible in principle to use crowdsourcing in cases of multiple fields with the same nameForAutofill.
6. Information about a password form is spread in different places, which adds confusion and complexity. For example there are provisionally saving in browser and renderer processes.

The general ideas of this refactoring

1. This refactoring is about code that handling DOM data and events, i.e. about fill/save related code. Nothing outside of that code would be influenced (eg. PasswordStore, Password Manager UI, Credentials API etc).
2. The code will split in classes with explicit responsibilities, that should be easy testable. Whenever makes sense mock objects will be introduced.
3. Much logic will be moved from the renderer side to the browser side, that would
    a. Simplify the design (eg. no need to propagate server-side data to the renderer).

    b. Unify the implementation between iOS and other platforms, as result less maintenance cost and some features on iOS for free (server-side help for filling and saving).
    c. Allow to parse data with all available information - server-side, saved credentials.
4. New stable identifiers will be introduced (more details in section). That would allow to fix form/field matching problems.
5. The old assumptions and hacks will be removed.

Corner cases that must be considered

Notes:

1. *This list is for not forgetting take into consideration different not-trivial and non-obvious corner cases. Probably this refactoring wouldn't tackle code for processing some of these cases, but anyway it's good to remember them.*
2. *Please add new items.*
3. *Ideally a test for each corner-case should be written.*
4. *Handling parsing-related corner cases is detailed in the parsing design*

1. Some forms/credentials shouldn't be processed
    a. GAIA sync credentials should not be saved. Furthermore, GAIA sync credentials saved before sync was set up should not be filled on passwords.google.com reauthentication page.
    b. With URL about:blank, chrome:// etc. The URL can be checked by ChromePasswordManagerClient::CanShowBubbleOnURL (filling/saving).
    c. Blacklisted sites (saving).
2. The site URL might be changed in any moment (but origin, i.e. <scheme>:<host>:<port> should stay the same). Example accounts.google.com.
3. The form may be submitted with a target attribute is set to a different frame (by settings <form target="_top"/> for example), as the result in the current Blink implementation the submission is seen on the target frame. We need to able to match it with the observed form.
4. The forms may be arbitrarily scattered across different frames. PasswordManager should cope with different events (forms rendered, form found, navigation detected) simultaneously coming from different frames.
5. A site may mangle username/password after user typing.
6. A site may mark up the password field with autocomplete="new-password" or "current-password" but provide no text field marked with autocomplete="username".
7. Elements can be disabled: "A disabled input element is unusable and un-clickable. (…) Disabled <input> elements in a form will not be submitted."
8. Password elements can be marked as credit cards with autocomplete="cc-number" (and similar).
9. Visibility and autocomplete mark-up can clash
10. Readonly password fields might indicate filling being useless

Design

TODO: move picture from internal document

Notes:

1. PFM - PasswordFormManager.
2. All rectangles are classes.
3. Arrows represent data sending.
4. The picture on iOS is very similar, the difference is only in replacing PasswordAutofillAgent to PasswordController.
5. iOS code wouldn't require much effort, because I've changed it for last half year with keeping in mind the big picture.
6. We have to also write a fuzzer for FormParser (in addition to unit tests), to early catch vulnerabilities which might come from moving the parsing from renderer to the browser.

Class responsibilities:

1. PasswordAutofillAgent works with DOM, sends FormData that represents forms and fills credentials. It keeps stable identifiers of forms/fields. It sends data on any form related changes in DOM: a new form found, removed, user typed in input element (for saving fallbacks).
2. PasswordManager receives data from the renderer, creates one PFM per one form, propagates information from the renderer to the right PFM. It makes a decision when a submission was successful.
3. PasswordFormManager is a bookkeeper of all information that we have about one form (server-side, DOM, saved credentials etc) and makes some high-level decisions (to fill, to initiate parsing). Other its responsibilities should be moved to other classes (FormFiller, FormSaver, FormUploader). It's owned by PasswordManager.
4. FormParser (new class) takes FormData and server-side predictions and returns PasswordForm object as a parse result. It's owned by PFM.
5. FormFiller (new class) is responsible for sending filling data to the renderer (namely PasswordFormFillData object). It's owned by PFM.
6. FormSaver is responsible for saving/updating credentials to PasswordStore. In the initial version it will be cloned to UI, instead of cloning whole PFM as we do now.
7. FormUploader (new class) is responsible for sending uploads to the server. Basically it will just contain all current upload logic from PFM.

Form/field Identifiers

Stable identifiers for form/fields are required for several purposes:

1. The decision which fields should be filled is done in the browser process, then the credentials are sent to the renderer. We need to be sure that the right fields are filled.

   And it should be robust against DOM changes by JavaScript and ambiguity of id/name/action attributes.
2. In order to correctly calculate a signature and make a server-side upload, the submitted form should be matched to the observed form.
3. We will be able in future to use more robust form signatures for crowdsourcing (the current signatures depend heavily on id/name) (draft design doc).

There are several options for introducing stable identifiers:

1. Keep track of all forms/fields in PasswordAutofillAgent in stl::map<Form, id>, stl::map<Field, id> (or something like that) and assign to each form/field a unique id. We should be careful here, since there is a bug where people are complaining that Password Manager/Autofill keeps too long fields/form elements alive.
2. Set some webpage-visible attribute (say “autofill_identifier” or random attribute name) to Blink objects that represent form/fields.
3. Introduce some internal Blink attribute to form/fields elements, which can be set by PasswordManager (similar to SuggestedValue, that was introduced specifically for autofilling).
4. Use a memory address of underlying WebKit object as unique identifier.
5. Implement kind of WeakPtr for WebKit objects. WebPrivatePtr (it is used for keeping HTML* objects in Web* objects) has a template argument to be strong or weak pointer.

Notes: Independently of the way of introducing of stable identifiers, they might also be used in Autofill.
Separate design doc for this (*go/blink-form-stable-ids*).

PasswordFormManager redesigning

1. PFM (PasswordFormManager) has now too much responsibilities now and it’s very complicated (it’s about 1600 LOC). In the new design, it will contain all information about a specific password form and creates objects for doing actions (FormFiller, FormSaver, FormParser).
2. PFM is created by PasswordManager when a form is found in DOM for managing this specific form. Note, that the form may be synthetic, i.e. comprising from input elements which are not in any form.
3. PFM waits for server-side data some period of time (say 0.5 sec). If the server-side response is not received, it fills the form without it. The exact waiting period might be found by UMA. In general it’s possible to do refilling if server-side data received after, but it would change UX (how the user would fills, when CPM refills). So it could be a separate feature.
4. PFM receives user input on each keystroke in input elements from the managed form.
5. PFM always has up to date password form information. So we will be more free when to initiate saving.
    a. We would not need to send DOM data on DidStartProvisionalLoad, since this data is already in PFM.

    b. We would be able to easily experiment and add new submission detection signals.
6. PFM receives information that the managed form is submitted from PasswordManager.

Form Parsing

Form parsing is one of the most important part of CPM: converting raw DOM data to some meaningful data. The most important question is to find the username/password/new-password fields. A lot of our efforts were directed on improving it: crowdsourcing, client-side classifiers, a lot of hacks and tricks based on some specific use cases and observations.

Form parsing on iOS was implemented as a model for general form parsing (iOS design doc, iOS FormParser sources). iOS parsing is simpler, because a lot of signals are not available on iOS (no server-side data, no user input, no client-side classifiers). But the iOS implementation could be easily extended to adding new signals.

The separate design doc will be written for form parsing.

For short benefits that we will have

1. Form parsing will be easily testable in unit tests (iOS form parsing tests for examples).
2. We will make many important design decisions explicitly and will have tests for this. For example about signal priority: what has more priority autocomplete attributes or server-side predictions.
3. It will be extendable for adding new signals, say new ML classifiers.

PasswordManager class changes (WIP)

PasswordManager changes will only be related to handling PFM objects.

How PM works with PFMs now (technical details)

PFM could be in one of two possible states: pending (PFM is in |pending_login_managers_| field of PM) and provisionally saved ((PFM is in |provisional_save_manager_| field of PM). When the form is considered to be submitted, it’s moved from pending to provisionally saved and all other pending PFMs are removed. Here is the corresponding code.

One of the drawbacks of that approach is that PM incorrectly understood that the submission happened (but it was not), then PM doesn’t have an option to propose saving another form later, since all pending PFMs are gone.

How it will work

Basically the main thing which we need from the current scheme is to know which form is submitted. So it will be one field in PasswordManager (say |pfm_managers|), which will have all PFMs, one per each form. When PasswordManager thinks that some form is submitted, it sets the corresponding PFM in submitted state (just sets a boolean). Then later when PasswordManager thinks that the submission was successful, it uses PFM that was submitted for credentials saving/updating.

Testing and mocking

For simplicity of unit testing, mock objects for all objects for which it makes sense will be created.

In the current design the testing in many cases are difficult. Sometimes writing tests takes much more time than writing production code. In many cases browser tests are the only way to write tests. Unit tests are more preferable for testing most logic in this design doc because

1. It's possible to test more specific behaviour with unit tests, without involving entire Chrome infrastructure.
2. Browser tests are inherently much more flaky (multiple threads etc).
3. Browser tests are longer to run.

For instance, for the current form parsing logic browser tests are used password_form_conversion_utils_browsertest.cc: a lot of boilerplate, it's hard to understand and structure important behaviour. On other hand on iOS unit tests are used for form parsing logic testing(form_parser_unittest.cc). They have the minimum amount of boiler plate, each test has clear input and output (see no form test for example).

Metrics and Logging (WIP)

Code changes

1. **FormParser** a new class for form parsing will be introduced. The simplified version written for iOS can be seen here (iOS design doc), in the new version will be additional sources of data used - server-side, HTML client-side classifiers, user typed values. Output of FormParser is PasswordForm. It will be covered in a separate form parsing design doc.
2. password_form_conversion_utils - will be removed (its logic will be superseded by FormParser).

3. PasswordFormManager will be much simplified, and its responsibilities will be changed to keep all available info about the managed form - namely FormData, server-side responses, user typed values, page events etc. All save related logic will be removed from it.
4. FormSaver has all needed logic for saving - and it is moved to UI. As input it requires PasswordForm (result of parsing) and FormFetcher (for knowing saved credentials).
5. PasswordAutofillAgent
    a. All parsing related logic will be removed.
    b. The current tricky and error-prone logic for form matching will be removed and instead of it new logic based on stable form/field identifiers logic will be introduced.
    c. For DOM data extraction Autofill function ExtractFormData will be used.
6. All logic with provisionally saved forms in renderer and browser code will be removed (no need since stable identifiers).

Implementation

Steps

1. Introduce new IPCs, that correspond to current IPCs, with changing PasswordForm -> FormData.
2. Implement sending IPCs from //content's PasswordAutofillAgent (it's easy, because FormData is already a part of PasswordForm) (non iOS).
3. Implement sending forms from PasswordController (it's easy, FormData is already used for parsing) (iOS part).
4. Implement processing of new IPC in PasswordManager (reuse current code as much as possible).
5. Implement NewPasswordFormManager class (which ultimately will replace and become the PasswordFormManager).
6. Implement FormParser.
7. Implement filling with NewPasswordFormManager
    a. Extract and simplify finding best matches.
    b. Extract filling logic from PasswordFormManager and use it in NewPasswordFormManager.
8. Introduce stable form/fields identifiers in the renderer.
9. Implement saving with NewPasswordFormManager
    a. Pass a submitted form to NewPasswordFormManager.
    b. Implement Save/Update methods.
    c. Implement interfaces needed to pass NewPasswordFormManager to UI.
10. Switch from PasswordForm to FormData in PasswordGenerationAgent

    a. Currently, the only information from PasswordForm used there is FormData, so this should be easy.

11. Use UKM or/and server-side data from Canary and Beta for checking that no site is broken.

Estimation and work parallelising

It seems that work pretty parallelizable, the following tasks could be done in parallel:

- New renderer code on //content-based platforms (1 person/month).
- New //ios/web code on iOS (1 person/week).
- Form parsing, including incorporating server hints (1 person/month).
- Moving all saving/updating logic from PasswordFormManager to FormSaver (1 person/week).
- Implementing new PasswordFormManager (2 person/weeks).
- Implementing FormFiller (2-3 person/weeks).

In total 4-6 person/month. So it should be doable by 2-3 engineers in one quarter.

Which problems/bugs that solves the new design.

1. Filing with the server-side signal: PasswordFormManager would wait to the server side signal (but not more than some const time period) and fill only after this.
2. Form matching problem (empty and ambiguous identifiers, JavaScript changes DOM) would be solved by stable identifiers.
3. CPM in some cases constantly fills password fields (bug). The reason is usually that now we can't say for sure whether fields were filled earlier and keep filling them. Having stable identifiers would allow us to fill any specific field only once.

New directions that would be simplified by the new design

1. Implementing support of multi-page login flow: we will have all needed information in the browser process, so we can link it in some way between different pages.
2. Integrating client and server-side classifiers: it would be one place (FormParser) where parsing happens and where all data is available.
3. Experimenting with new few form submission detection signals. For example with skipping submission success detection by server-side signal.