A Secure Password Manager Governance Framework for Web User Authentication

Ali Cherry

Master of Science by Research

University of York

Computer Science

April 2024

Abstract

Existing password management frameworks fall short of providing adequate functionality and mitigation strategies against prominent attacks. Unfortunately, the architecture of these frameworks is not aligned with the distributed nature of web applications and is vulnerable to credential theft attacks by network-side, e.g. TLS Proxy in the Middle (TPitM), or front-end, e.g. cross-site scripting (XSS), eavesdropping adversaries. Browser-side frameworks, HTML Autofill and Credential Management API, are inherently vulnerable to XSS-credential theft. ByPass, a manager-to-server paradigm, is inherently vulnerable to TPitM-credential theft. Furthermore, all of the aforementioned frameworks employ an inaccurate app-to-credential mapping strategy, domain-based credential mapping, and might inadvertently divulge user's credentials to unintended (e.g. deceitful) web applications.

We propose Berytus, a novel browser-based governance framework that mediates between web applications and password managers to orchestrate secure and programmable authentication sessions. It is positioned between the web application and the password manager, operating natively in the browser, and providing an API for each party. Berytus harmonises multiple password manager usage by requiring available password managers to register with it. Present frameworks do not couple specialised security facilities with their approach, rather their credential transfer security depends on the application of standardised security measures in the web/browser landscape to mitigate against prominent attack vectors, e.g. Content Security Policy for XSS mitigation. Conversely, the Berytus architecture equips web applications with certified app-specific cryptographic keys to streamline an authenticated and accurate app-to-credential mapping strategy. Furthermore, Berytus mediates an authenticated key exchange between the web application and the password manager to achieve app-level end-to-end encryption of credentials, which as we show, can streamline a confidential credential transfer communication that is immune to credential theft attacks via phishing, XSS, malicious browser extension code injection and TPitM.

To assess the feasibility of Berytus, we extend Firefox to incorporate Berytus and develop Secret*, a Berytus-compatible password manager for programmable authentication and registration sessions. We make our code artefacts publicly available, provide a comprehensive security and functionality evaluation and discuss possible future directions.

Acknowledgements

Over the course of the research programme, my supervisors Dr Siamak Shahandashti and Dr Konstantinos Barmpis played an instrumental role in my research and personal journey. I give my utmost thanks to them as shepherds, providing me with needed guidance and scrutiny, but more important, for their friendship. Completing this research programme has brought immense satisfaction; it fulfilled my pursuit of exploring and contributing to my area of interest. Moreover, the City of York's natural and medieval scenes were a pleasure to the soul. I have worked in between gardens, pastures, rivers and bridges for which I am grateful. Colleagues, friends from York, thank you for the wonderful moments of joy and laughter. Special thanks to my abiding companions for their camaraderie in all circumstances. Above all, this work is a tribute to my Lord and my God, Jesus Christ; thank you.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for a degree or other qualification at this University or elsewhere. All sources are acknowledged as references. Part of this work is accepted and is due to appear at the proceedings of the 2024 International Conference on Information and Communications Security [36].

Table of Contents

List of Tables

List of Figures

1. Introduction

Decades after the introduction of passwords into computer systems, and after many proposals to replace password authentication [5], passwords remain the dominant authentication scheme despite its (inherent) poor security properties. On the web, insecure communication channels, database breaches and password reuse exacerbate the threat of account break-ins when password authentication is used. A cryptographic authentication alternative to passwords, e.g. digital signatures, can significantly mitigate against such attacks. However, shifting towards such alternative strategy requires a behavioural and technical cost. First, in the case of digital signature-based authentication, users have to digitally hold the necessary cryptographic material to compute signatures; e.g., the material can be stored on a flash drive. Second, the system's existing authentication subsystem has to be amended to replace password authentication with the desired authentication scheme. This implies that the replacement cost is incurred on each existing system or web application to replace passwords with a cryptographic alternative. Therefore, considering the size and complexity of the web infrastructure, with web applications having invested in passwords as the de facto authentication scheme, replacement in the short term appears improbable. Accordingly, to solve the password problem, the preliminary question is materialised as: what kind of scheme or strategy would increase password (credential) security while incurring minimal cost?

A practical solution is to use a software tool that composes a unique and strong password for each web application; (1) eliminating password reuse which defends against online database leaks. And that stores the composed passwords; (2) eliminating the cognitive burden of memorising passwords. These tools are known as *password managers* and they can operate as standalone applications or can be integrated into the browser where it acts as an *agent* which automatically populates input fields with the stored credentials. Most important, the password manager, being a client-side only solution, does not require any active cooperation from the web application. This results in a minimal technical cost for the web application while the user benefits from increased security and convenience. Evidently, the user has to install this tool on his machine. However, modern browsers have a built-in password manager which facilitates the integration without any further effort from the user. Software-based credential management through password managers enhances users' security, for passwords or other credentials. Therefore, due to its minimal cost and its practicality, it can be considered as one of the most promising strategies to tackle insecure password behaviour effectively. Consequentially, with software-based credential management, password managers act as agents on behalf of users, assisting and guiding them during authentication, ensuring the avoidance

of common pitfalls, e.g. password reuse, where possible, and alleviating associated burdens, e.g. digitally holding cryptographic material.

Unfortunately, while the premise of password managers is sound, in practice this strategy has its own shortcomings due to the nature of how it operates — for good reasons. Occasionally, as we will see later, password managers behave inaccurately, potentially causing usability and security issues. We consider three existing approaches for secret manager-assisted web user authentication: HTML Autofill [38], the Credential Management API [26] and ByPass [29]. Briefly, all three existing frameworks for facilitating password manager and web app communications have shortcomings. HTML Autofill piggybacks on insufficiently expressive markup language; ByPass is not compatible with established user mental models and takes away the control of user experience from web app developers; and Credential Management API is only available to built-in password managers and, similar to the aforementioned frameworks, is prone to inadvertently divulge user credentials to unintended web applications due to inaccurate credential suggestion approaches.

1.1. Motivation

The adoption of password managers is still primitive [17, 19, 24]. The low adoption rate of password managers implies that the average user might resort to insecure password behaviour — resulting in low account security. Previous studies identified factors hindering or motivating the adoption of password managers [17, 19, 24, 33, 25, 34]. Findings included psychological and sociological factors (e.g. relinquishing control), contextual factors (usefulness) and password manager behaviour & availability. In Oesch *et al.*'s study [34], participants avoided using generated passwords in fear of having trouble inputting them in environments where the password manager is not installed, e.g. a SmartTV. Therefore, password manager behaviour & availability is a crucial factor for successful adoption and effective use of password managers [34].

Password manager behaviour is influenced by the underlying technologies and standards present in the web user authentication space. Its availability is a byproduct of the framework in which it operates. The assumed framework's architectural decisions impact the feasibility of integration and determine where the password manager can be made available. In spite of the existing password manager frameworks being insufficient, we believe that an augmented and open framework, not only can fill the existing gaps, but also can boost password managers to provide highly valuable functionality; thereby empowering end-users and attracting non-users.

1.2. Research question and objectives

We refer to password managers as secret managers due to their capability of storing various types of credentials and not just passwords. Secret managers have been

implemented as web-only services and stand-alone applications, but unless otherwise specified, in this thesis we focus on in-browser secret managers. In light of the existing gaps, we clearly recognise the dissonance and incomplete integration (across the board) between web applications and secret managers when undertaking authentication. Therefore, we ask, considering the factors in this landscape:

RQ: Can we harmonise the interactions between web applications and secret managers to undertake programmable authentication sessions without degrading usability, while ensuring strong security measures and correct behaviour in a practical manner?

Based on this research question, we unravel the following research objectives:

- **Placing a mediator.** Secret managers and web applications are currently in dissonance. As a preliminary step, we aim to establish a mediator between the two, allowing them to communicate with each other through the mediator. The mediator should handle the co-existence of multiple secret managers, including built-in and embedded secret managers (i.e. extensions).

- **Defining a mutual agreement contract.** There are challenging instances that are faced by the web application and the secret manager, e.g., agreeing on an appropriate credential suggestion strategy or how credentials are transferred. Lack of agreement causes dissonance between the web app and the secret manager. We aim to streamline a purposely built agreement specification, taking into consideration the challenging instances, enabling both parties to work together in harmony.

1.3. Contribution

To answer the research question, we propose Berytus, a web governance framework that mediates between web applications and secret managers to orchestrate programmable registration and authentication sessions. It is positioned between the web application front-end and the secret manager client, operating natively in the browser. By establishing browser-based governance, Berytus ensures strong security policies (e.g. HTTPS only) and correct behaviour across various, potentially insecure implementations of secret managers [13, 28, 10]. Berytus achieves the best of all worlds: it provides specialised and extendable APIs, it is available to all browser-based secret managers, whether they are native or extensions, and preserves the established modus operandi for users and control over user experience by developers.

We instantiate our framework by extending the Mozilla Firefox browser and implement two APIs, a WebExtensions API for secret managers and a Web API for web applications. We develop Secret* (secret-star), a Berytus-compatible secret manager. We deploy an example web application at `https://github.com/ali`

`chry/berytus` where a programmable authentication session can be undertaken through a Berytus-compatible secret manager.

We provide comprehensive security and functionality evaluations and demonstrate that several crucial security protections and functionalities that cannot be or are not provided by existing solutions, can be provided by Berytus, including protection against credential theft attacks, as well as support for account structure design and for interactive challenge-based authentication schemes. The following usability enhancement and security services are the main outcomes of streamlining Berytus:

- **Unified secret management (platform-agnostic)** — Secret manager extensions can now directly register with the browser using the platform-agnostic web extensions API, addressing the usability challenge of multiple manager usage [34, 16] through the use of secret manager selection prompts. A web (platform-agnostic) framework paves the way towards standardisation across different environments, regardless of the running operating system.

- **Key-based credential mapping** — Berytus supports web application authentication against a certified public key. Consequently, web applications can assume ownership of cryptographic public keys, serving as application identifiers and, thus, replacing the troublesome domain-based credential mapping strategy [27, 32, 31, 6]. In particular, this solves two main issues. First, co-existence of distinct web applications under the same domain. Second, co-existence of the same web application under distinct domains. Both of these issues increase the risk of phishing which is now mitigated with key-based credential mapping.

- **App-level end-to-end encryption (App-level E2EE)** — Berytus mediates mutually authenticated key exchange mechanisms such as authenticated Diffie—Hellman, enabling app-level end-to-end encryption between web app backends and secret managers, using the frontend as the medium, for confidential credential transfer. App-level E2EE provides effective protection against credential theft via code injection attacks (e.g., through cross-site scripting [15, 13, 28] or through malicious browser extensions) and TLS proxy in the middle attacks.

The remainder of this thesis is structured as follows: Chapter 2 introduces the password problem, covering alternative authentication schemes, secret manager-assisted web user authentication frameworks and the security of secret managers. Chapter 3 presents our contribution, Berytus, and unravels its methodology, architecture, design and implementation. Chapter 4 carries out the evaluation of Berytus, tackling its security and functionality aspects. Chapter 5 offers a discussion on the validity of Berytus, limitations, recommendations for future work and concluding remarks.

2. Web authentication

2.1. Problem overview

A typical web user is required to maintain many passwords for her online accounts, a task that requires unreasonable cognitive burden. Secret managers are widely recommended by the experts to relieve users of such burden, and if designed well, bring extra security and usability benefits through the use of strong passwords and automatically completing login forms on behalf of the user, respectively. In summary, they assist users, often in an automated manner, during account authentication.

HTML Autofill [38] is the primary approach for secret manager-assisted user authentication on the web. Here, the web application declaratively signals, via HTML attributes, whether an input field can be automatically populated on behalf of the user, and if so, what type of information is expected, e.g. a password or an email address. The (in-browser) secret manager can then offer secrets on behalf of the user to automatically populate the field. In some circumstances, the secret manager has to rely on opinionated heuristics in absence of any explicit signals, causing them to stumble — occasionally. Furthermore, the HTML standard is neither sufficiently rich nor adequately expressive to be able to support nuanced and complex login mechanisms such as multi-step authentication (e.g., Secure Remote Password [1]). With the dawn of *single page applications*, logins are no longer necessarily tied to traditional HTML form submissions [40] but also harnessed through the JavaScript Fetch API [43]. This made *autofill-able* input field detection more difficult, requiring additional (fallible) heuristics.

The Credential Management API [26] can be seen as a solution in this regard; it does not necessitate field detection as it is the case in HTML Autofill. The API is a W3C Working Draft that provides a simple mechanism to store and retrieve user credentials (e.g., passwords). Using JavaScript, the web application can capture the username and password inputs, even when the input fields are not under a parent HTML element, and invoke the Credential Management API to store them. However, Credential Management API is only available to the so-called native user agent, i.e. the browser and its native compartments such as the native secret manager, as opposed to extensions. Hence, the API cannot be used to store and retrieve credentials into and from secret manager extensions.

In 2020, Stobert *et al.* proposed a remodelled password manager, ByPass, where it communicates directly with the web application's back-end through a bespoke API [29]. Such direct programmatic communication not only all but eliminates issues caused by misinterpretation, but means that ByPass is able to provide enhanced services such as account deletion and password renewal. Furthermore, since

passwords are directly communicated with the back-end, they are not susceptible to theft through front-end threats such as cross-site scripting and clipboard vulnerabilities. Despite all its benefits, ByPass requires both users and web app developers to buy into a major paradigm shift. Rather than assisting users in authenticating themselves when interacting with a web app, ByPass users need to launch their secret manager and request that it log them into their desired web app. This requires a change of the users' established mental models in interacting with web apps. Furthermore, as the login user interface is mandated to be that of ByPass, it takes away the control over the login user experience that web app developers relish today.

All three existing frameworks for facilitating secret manager and web app communications for assisted authentication have shortcomings. In this chapter, we will unpack the state of the art, discussing the existing frameworks in greater detail and reviewing password alternatives. Ultimately, by highlighting the gaps in this space and the persistence of the *password challenge*, we will demonstrate the need for an augmented, secure and practical framework which addresses the gaps.

2.2. Alternative authentication schemes

Bonneau *et al.*, motivated by the prevalence of passwords as the dominant authentication scheme, evaluated 35 other authentication schemes [5]. Despite proposal of various authentication schemes improving security, e.g., hardware tokens, passwords remain the go-to authentication scheme for newly crafted websites [5]. Clearly, the proposed alternative authentication schemes were not massively picked up to replace password authentication, despite some having superior security over passwords — a desirable measure reflecting the safety of digital accounts. Bonneau *et al.* consider *deployability* benefits of each proposed authentication scheme in their comparison [5]. Poor deployability might explain why some authentication schemes were not adopted.

"Server-Compatible", the forward compatibility of a text-based (password) authentication back-end system with alternative authentication schemes, is one of the deployability benefits that the majority of considered authentication schemes do not satisfy [5]. Hence, authentication schemes lacking Bonneau *et al.*'s contextualised server compatibility benefit incur additional effort or technical cost on web application developers to implement the necessary architectural changes into their existing (password-based) authentication system.

Secret managers provide all of the deployability benefits of passwords except for the *Browser-Compatible* benefit [5]; users are required to install their preferred secret manager on their own browsers. Nonetheless, secret manager-assisted password authentication is relatively easy to deploy and performs better in terms of security benefits when compared to user-entered password authentication (e.g., protection against phishing and password leaks) [5]. Note, here secret managers were evaluated assuming they employ HTML Autofill [5] which is a client-side only approach.

An approach other than HTML Autofill might require web application back-end changes, making it lack the Server-Compatible deployability benefit. On a final note, the authors do not consider websites' front-end compatibility, i.e. no client-side changes needed, as a deployability factor. Perhaps it is reasonable to assume that at a minimum, some sort of front-end modifications are needed to incorporate any authentication scheme.

In summary, we cultivate that the secret manager/HTML Autofill scheme has the highest deployability score and among one of the few that offer the Server-Compatible benefit [5]. Therefore, we highlight the value of the secret manager client-side scheme as an effective, practical strategy to increase users' password security on the web.

2.3. Multi-factor authentication

Password reuse and compromised passwords are major security threats. Due to the severity of such threats and the increasing risks of online database breaches and sophisticated password theft attacks, web applications have introduced multi-factor authentication. It adds another layer of security on top of passwords where the user is expected to provide, for example, a one-time code sent to his email address to proceed with the authentication. Hence, in such instances, an adversary would require access to the victim's email address in addition to his password.

It is salient that multi-factor authentication is a net security positive in the sense that it requires the adversary to attack additional domains which he might not be successful at. However, while multi-factor authentication is an additional security layer, it does not address the security and usability issues associated with passwords. Therefore, we reaffirm the need of a practical scheme that directly tackles insecure password behaviour, e.g., secret managers, instead of solely adding additional defensive layers on top of weak or reusable passwords.

2.4. Secret manager-assisted web user authentication

This section offers greater detail into the state of the art in existing technologies governing the interaction between secret managers and web apps for assisted web user authentication and registration. We discuss HTML Autofill, Credential Management API, and ByPass as major frameworks for secret manager-assisted web user authentication. Figure 2.1 outlines how these major frameworks are orchestrated from a high-level point of view and the necessary software modules (if any) to implement for a base (minimal) integration.

2.4.1. HTML Autofill

HTML Autofill is the process in which HTML input fields are filled by the browser (technically the *user agent*) with relevant data (e.g., personal information and se-

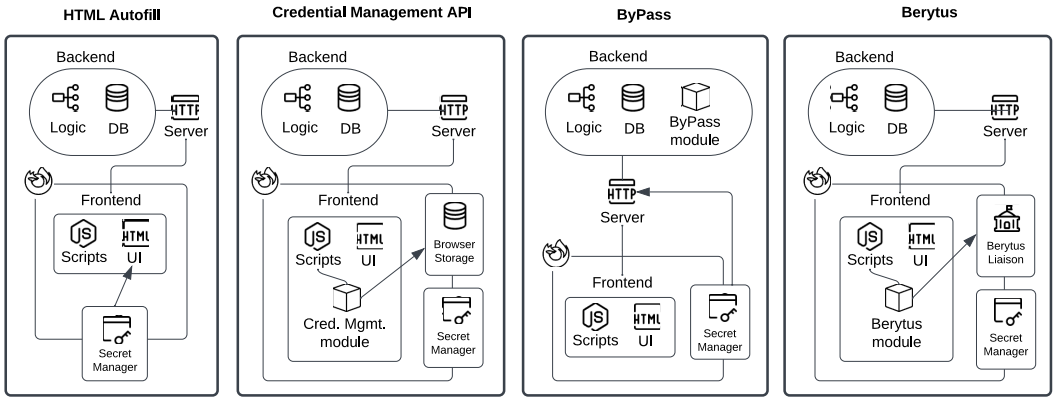

Figure 2.1.: Overview of HTML Autofill, Credential Management API, ByPass and Berytus architectures.

crets) on behalf of the user. Generally, the filled data would have been captured and saved by the user agent during an earlier web browsing session, or may be generated on the fly (e.g., a random password). Currently, built-in secret managers (i.e. native secret managers) and embedded secret managers (i.e. web extensions installed on the browser) are specialised agents to compose, store and fill passwords on behalf of their users through HTML Autofill. Figure 2.2 shows how a secret manager populates input fields with *autofill-able* data which could be a password value. By delegating the responsibility of managing passwords to secret managers implementing the HTML Autofill functionality, users benefit from additional security measures and enhanced usability. Strong passwords are generated by secret managers, instead of relying on the user's potentially weak password composition strategy, and only filled (exposed) on web pages if it has the same origin (domain name) of when it was saved. The former ensures secure password behaviour and relieves the user from the cognitive burden. The latter combats origin-based phishing attempts. Ultimately, end-users enjoy the added convenience of expedited login experiences without being concerned about security.

Web apps can support the filling process by integrating the HTML *autocomplete* attribute into relevant input fields [38]. This aspect of the HTML Standard might not be implemented for some login forms, leaving secret managers to the use of ad hoc heuristics for input field classification. Hence, at a minimum, no client-side web application code changes are needed for HTML Autofill. This results in great *deployability* [5] where this security and usability-enhancing functionality could be realized on (potentially) any website. However, the absence of the *autocomplete* attribute could lead to interaction problems between web apps and secret managers [31], causing inconvenience for users. For example, in the absence of the *autocomplete* attribute and other input "hints" (*id*, *name*, *type*) for the username field, most tested secret managers were not able to fill the field with the saved username value [31]. Therefore, while this paradigm is highly *deployable*, it is

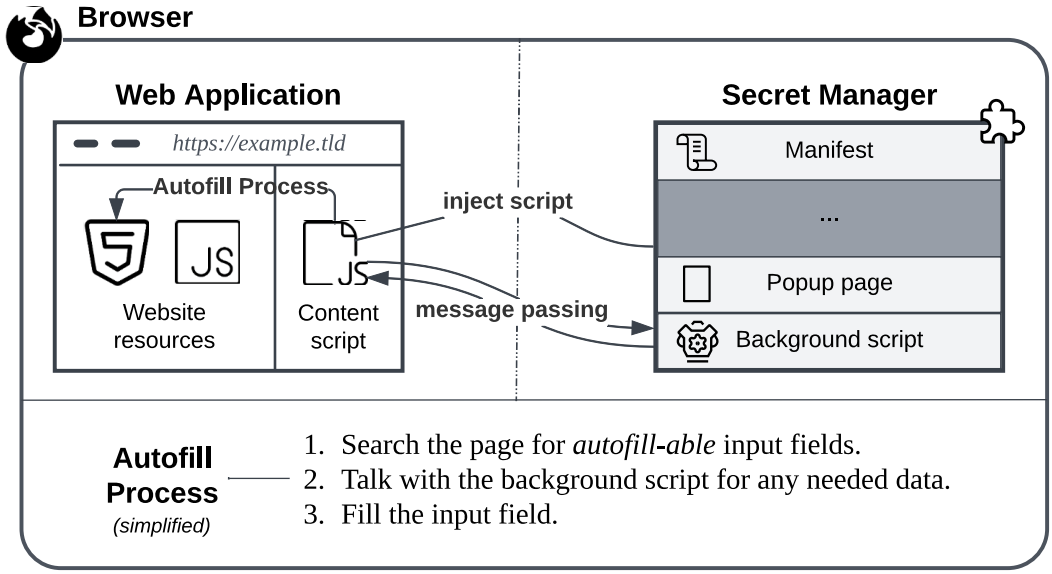

Figure 2.2.: Illustration of the HTML Autofill incorporation and application process executed on a web page by browser-based secret managers.

prone to behave incorrectly. On a final note, HTML Autofill is a *forceful* approach — web applications cannot officially disable its behaviour. The HTML Standard concurrently specifies a method for web apps to disable autofill (by setting the autocomplete attribute to `off`) and a suggestion for user agents to ignore such opting-out declaration at their own discretion [38].

Stajano *et al.*, motivated by the presence of heuristics causing secret managers to stumble, proposed Password-Manager Friendly (PMF), an additional set of HTML semantic labels for reliable autofill behaviour [14]. Unlike the HTML Standard's autofill specification, PMF aids secret managers in detecting different form types (login, registration, password reset and password change) and submission errors. If web applications incorporate those additional semantics into their forms, secret managers would no longer need to use fallible heuristics and, thus, secret managers would behave correctly when performing HTML Autofill. PMF is a positive step towards a more robust autofill behaviour where its semantics can be adopted into the HTML Standard in a future version. However, autofill, whether with the PMF's semantic labels or not, is not designed for interactive authentication challenges. An interactive authentication challenge, such as Secure Remote Password [1], is a multi-step authentication challenge with a bi-directional data flow. While a bi-directional data flow can be emulated using HTML by dynamically appending ad hoc HTML elements (e.g., with data embedded as an attribute) by both the web application and the secret manager in a sequential fashion, it is an anti-pattern; that is not what a markup language is designed for. A more appropriate means of communication between two software agents is through the introduction of a

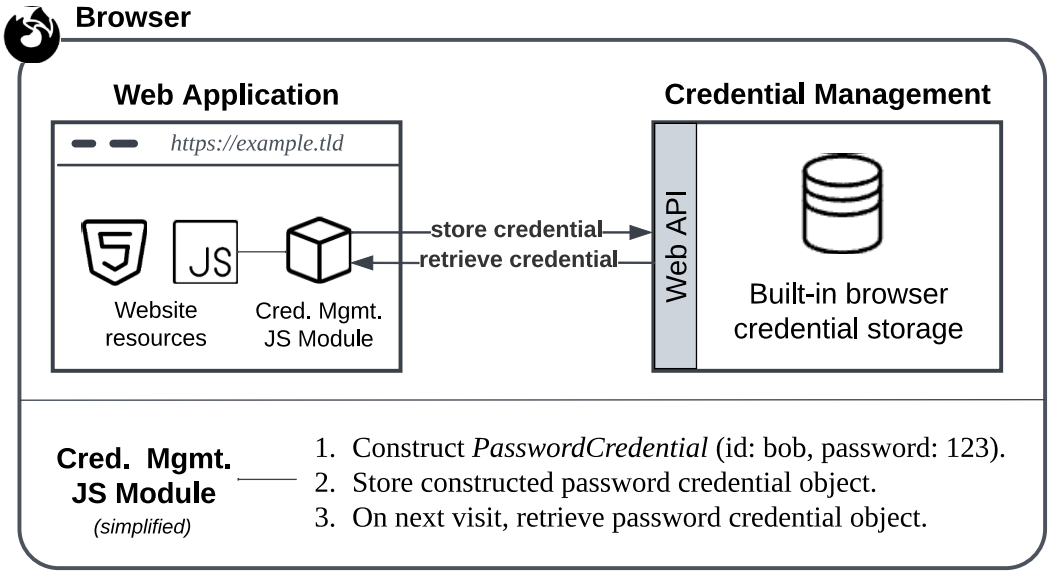

Figure 2.3.: Illustration of the Credential Management API password transfer (storage and retrieval) process between the web application and the built-in browser credential storage.

specialised API. Furthermore, both specifications (the HTML Standard and PMF) do not address the need for customisation, e.g., the definition of additional username or password fields in the credential structure.

To summarise, the HTML Autofill standard [38] is limited, secret managers have inconsistent filling behaviour [31, 34], and web apps might design arbitrary autofill-incompatible authentication protocols that require an interactive bi-directional data flow or credential structure customisation.

2.4.2. Credential Management API

As autofill was designed for user agents to aid users in sign-in forms, it became difficult to detect sign-in ceremonies leveraging the Fetch API [26, 43]. With the use of the JavaScript Fetch API, credential transfer over HTTP is no longer dispatched through traditional HTML form submission. Username and password fields are no longer necessarily siblings under an HTML form element (parent node), making it troublesome for HTML Autofill heuristics to detect username and password fields since they are now separated. As such, in this scenario, and depending on the streamlined heuristics, secret managers might fail to fill and save passwords. Lastly, user agents lacked support for federated sign-ins, and password change could be further supported by requiring web apps to notify user agents when credentials has been changed [26].

The Credential Management API was proposed to ensure correct credential management and to support users with federated sign-ins [26]. The API was published

in 2015 as a W3C Working Draft. Fundamentally, it offers a programming interface for web apps to store and retrieve credentials into and from the user agent. Using JavaScript, a web app can insert a *password credential*, constituted of a login handle (username) and the user's plaintext password, into the browser storage (see Figure 2.3). When the user visits the web app in the future, it can retrieve the password credential from the browser storage, programmatically, using JavaScript.

Credential Management API is purposely designed to be simple and extensible, it abstracts away from authentication and solely focuses on credentials [26]. Moreover, it supports credential change through the same storage mechanism.

Web Authentication (WebAuthn) [22] is an extension to the Credential Management API that enables digital signature-based authentication by introducing a new credential, *public key credential*. Currently, some secret manager extensions act as third-party public key credential providers by intercepting the WebAuthn API [42].

Credential Management API is implemented in modern browsers and only allows integration with the native secret manager and not pluggable secret managers (web extensions). It does not offer password generation, nor does it provide identity information. Moreover, just as it is the case with HTML Autofill, it does not offer credential customisation to address the needs of non-conventional authentication systems. For example, since credentials are independent, i.e. no two credentials can be combined or grouped under a single account record, and each must represent the sufficient authentication material of an account record, a web application requiring a composite login handle (e.g., account id and username) or a composite secret (e.g., two or more passwords) cannot achieve that through the API as the password credential expects a single login handle and a single secret. Note, a web app can circumvent this limitation by concatenating the identifiers and passwords to fit in one password credential. However, that would be considered as an anti-pattern since the API specification does not tackle such composition and expects identifiers and passwords to be individually atomic. Alternatively, an API that explicitly addresses such functional requirement is more appropriate.

While the Credential Management API is simple and extendable, it is insufficient to be a practical authentication and registration framework. It is not designed to cover the full domain of programmable authentication and registration sessions (e.g., identity information, account status and verification), it is not geared towards generic authentication subsystems where accounts could span several credentials and, at the moment, password storage is exclusively tied with the built-in browser secret manager.

2.4.3. ByPass

Motivated to address the usability issues of secret managers affecting adoption, Stobert *et al.* proposed a re-imagined password management model where the secret manager communicates directly with the web application's back-end whilst ensuring sound security measures [29]. As the secret manager directly interacts with the web app's back-end, front-end security vulnerabilities such as cross-site scripting (XSS)

are now mitigated. Note, since we could not find a public repository for the ByPass API, we did not provide an illustration for one of its account-related operations such as account login.

In ByPass, to log into a website, the user needs to open the secret manager's interface instead of navigating to the website. The user then searches for the website they wish to visit within the secret manager and the manager handles the account registration or authentication process depending on whether an account for the website is already stored or not. Fundamentally, all account authentication related operations such as password change or account deletion can also be carried out by the manager and the user only needs to interact with the secret manager's interface.

ByPass requires users to learn to interact with their browsers quite differently from what they are used to when navigating the web. The secret manager basically acts as an entry portal to all websites that support ByPass and the users need to either remember or find out via trial and error whether a website's back-end supports ByPass. From the developer's point of view, ByPass requires the implementation of new server-side components and also takes away from the developers the control they have over the design of login pages. Stobert *et al.* also recognise this practical challenge of requiring a "buy-in" from users and website developers to adopt the new, re-imagined password management model.

2.4.4. Domain-based credential mapping

When the user is faced with a login form, under HTML Autofill, the secret manager suggests a list of registered credentials to populate the username and password fields with. Secret managers often use the web page's domain name (origin) as an index key when suggesting credentials. I.e., the secret manager would search its database for credentials associated with a particular domain name and the found credentials are suggested to the user. This is called app-to-credential mapping, or simply, credential mapping.

Currently, HTML Autofill, ByPass and the Credential Management API rely on domain-based credential mapping; the domain name is used as an index key. Domain-based credential mapping protects against salient cases of phishing. For example, assume the user has a registered account under `pizzajoint.co.uk` (domain `A`). He stumbles upon a web application hosted at `pizzeria.it` (domain `B`) where it asks him for his domain `A` credential to receive a discount. In this case, the secret manager would not suggest filling his domain `A` credential since the domain names differ. Hence, assuming domain `B` is unaffiliated with domain `A`, the secret manager would have protected the user from this phishing attack. However, domain-based credential mapping can be inaccurate. Domain `B` could be in fact a legitimate affiliate of domain `A`. This causes a usability issue in secret manager-assisted authentication. The user has to circumvent this by manually instructing the secret manager to populate the credential of domain `A` on domain `B`.

In more technical detail, the same web application's authentication portal can be deployed to several, distinct domains. Moreover, domain names can be recycled,

e.g., due to domain name expiration, which might lead to accidental credential leakage by secret managers. Conversely, distinct authentication portals can be deployed to the same domain. For instance, consider a web forum hosting multiple, distinct web applications under the same domain name. Under the forum's domain, the secret manager would suggest all registered credentials to all hosted applications. Therefore, the secret manager would not be able to protect the user from phishing web applications hosted in the forum.

We identify the three aforementioned troublesome instances for domain-based credential mapping. First, unconnected (distinct) authentication portals can reside in the same domain. Second, the same authentication portal can reside in multiple domains. Third, an authentication portal residing under one domain can detach from it. These realisable instances clearly demonstrate the ineptitude of domain-based credential mapping of serving as an accurate app-to-credential mapping strategy for the web. While it protects against potential cases of phishing, it might hinder the usability of secret manager-assisted authentication and, worse, might regress credentials security (e.g., by leaking credentials as a result of domain recycling). The upside of domain-based credential mapping is that it does not require any supervision or involvement from web applications — the secret manager assumes that credentials registered under a specific domain are to be suggested (e.g., under HTML Autofill) for any web page in the domain. To summarise, we can see the practical benefit of domain-based credential mapping: a *deployable* credential mapping strategy that functions fairly well in practice. However, its indeterministic security and usability outcomes underline another significant gap in the secret manager-assisted authentication space. This raises a question on the possibility of streamlining a more accurate credential mapping strategy that is indifferent to the aforementioned instances. This is aligned with previous related work on the absence of an appropriate web standard for "Using Credentials in Multiple Environments" [31] (i.e. multiple domains).

2.4.5. Interaction problems

In Sections 2.4.1 and 2.4.4, we have referred to a few cases where secret managers, under HTML Autofill, fail to detect input fields or fail to accurately suggest credentials; or more generally, fail to function correctly. In previous work, Huaman *et al.* systematically identified 39 "interaction problems", i.e. challenging instances, between web applications and secret managers based on a collected set of user reviews and GitHub issues linked to 30 secret managers [31]. In a challenging instance, the user might be able to circumvent the problem through manual intervention to achieve the desired secret management behaviour. For example, when the suggested credentials are not relevant, the user can manually instruct the secret manager to use a credential associated with a different domain. Other instances might be fatal, user intervention does not lead to desired secret management behaviour, e.g., when the generated password is bigger than the input field's *max-length* [31]; forcing some users to compose their own passwords.

More important, secret managers behave differently from one another. A challenging instance for one secret manager might be fatal for another. This unsettling fact is problematic and has several complications. First, it introduces a new dimension when adopting a secret manager. Which secret manager would be the most effective in handling these edge cases? For some users, awareness of this fact might discourage them altogether from adopting a secret manager while others might have to spend some effort in being diligent when choosing a preferred secret manager. Second, each secret manager therefore implements HTML Autofill in an opinionated way, making it difficult to compare HTML Autofill with other approaches; which implementation of HTML Autofill are we comparing against? The polymorphism of (potentially insecure [13, 28, 10] and unusable [31]) HTML Autofill implementations clearly highlights the potential benefit of regulating secret management behaviour to guarantee secure and predictable credential transfer behaviour regardless of the secret manager implementation.

On a final note, the authors report problematic instances where web applications have used "obscure" JavaScript routines, e.g., manipulating user input using JavaScript. Therefore, since web apps have the autonomy to implement arbitrary front-end processing routines, regardless of its kind, a programmable approach to secret manager-assisted authentication (e.g., the Credential Management API) would be more interoperable than a declarative and forceful approach (e.g., HTML Autofill).

2.5. Relevant security risks

Cross-site scripting (XSS) and man-in-the-middle (MitM) attacks are typical web attacks that can be used to steal a user's credentials whether provided by the secret manager or manually inputted by the user. A less common attack vector, but has similar implications as XSS, is the use of a malicious browser extension to inject JavaScript code. Fundamentally, all of those attack vectors are capable of delivering a malicious HTML/JavaScript payload to steal the user's credentials. Apart from web attack vectors, clipboard vulnerabilities in the user's environment place an additional security risk when the credentials are copied to the clipboard; e.g., copied due to interaction problems (Section 2.4.5) or when using standalone secret managers.

Generally, XSS relies on vulnerable web applications that do not sanitise user input, causing potentially malicious payload to *traverse* across the website to target users. Generally, the MitM attack relies on vulnerable network nodes that are in between the website and the user which can be used to manipulate the website's HTML/JavaScript and inject potentially malicious payload. As it is widely known, TLS protects against HTTP MitM. However, TLS Proxies [11] can be easily deployed in an IT-managed environment, e.g., in corporate offices, enabling an adversary with access to the TLS Proxy to eavesdrop and manipulate the HTTPS response payload without users' awareness; we call this attack vector TLS Proxy

in the Middle (TPitM). Moreover, malicious browser extensions leverage the use of privileged browser facilities to inject malicious scripts into a web page's execution context. Lastly, for clipboard attacks, an adversary capable to run a software process on the user's machine (or with physical access to a locked machine [27]), leverages any clipboard vulnerabilities or loose security measures in the user's operating system to steal the copied credentials. For example, in instances where a secret manager cannot transfer the user's credentials to the web application, the user is forced to copy his credentials to the clipboard and paste them manually [7]; thereby exposing his credentials to adversaries with access to the clipboard. The following strategies can be used to mitigate against the aforementioned JavaScript code injection attack vectors (XSS and malicious browser extensions).

Content Security Policy. This is a standardised HTTP security policy, conveyed using HTTP headers or specified in the HTML `meta` tag. It is a preventive mitigation strategy that aims to defend against cross-site scripting and adjacent attack vectors. Through using Content Security Policy, web apps can specify restrictions on the sources from which the page code gets loaded. For instance, if the policy `default-src: 'self'` is in place, the execution of imported scripts will be restricted to those that are loaded from the same origin, and all network requests sent through the fetch directive [37] that are not destined to the website origin will be aborted. Using such tight security policies may be too restrictive in some cases where components of a web app may need to be imported from multiple origins which are not fixed in advance, e.g., web apps that use third-party services such as web analytics. Furthermore, Content Security Policy can only limit the execution of *imported* or *embedded* (inline) scripts in cross-site scripting attacks but does not prevent malicious browser extensions from injecting JavaScript code into the web page.

Credential Tokenisation. This is a mitigation strategy, initially proposed by Stock and Johns [15], where credentials are provided on the user interface (HTML document) as tokens and substituted (detokenised) with the genuine credentials in the dispatched HTTP request body using a search and replace algorithm. Tokenisation ensures that the credentials are not available in the clear to the front-end, and by extension any front-end eavesdropping adversaries, including malicious browser extensions[1]. In Stock and Johns' [15] credential tokenisation approach, the secret manager examines the HTTP request's destination origin and the password parameter name in the `application/x-www-form-urlencoded` (URL-encoded) HTML form. If the origin or parameter name does not match the ones stored in the secret manager (which have been captured during registration), detokenisation does not occur. Unfortunately, their approach is designed for HTML form submission and is not compatible with asynchronous JavaScript data submission (e.g., using Fetch API). Nonetheless, an alternative, theoretical approach for Credential Tokenisa-

[1]This implication is true in browsers where extensions cannot capture HTTP request payloads.

tion which supports HTML form submissions and the JavaScript Fetch API [43] is sufficient to combat front-end adversaries from stealing the credentials, including malicious browser extensions, assuming detokenisation exclusively occurs in HTTP requests destined to endpoints owned by the legitimate web application.

2.6. Inconsistent security of secret managers

Oesch and Ruoti [28] re-evaluated previously studied and vulnerable secret managers [10, 13, 15, 6]. Such vulnerable secret managers inadvertently expose credentials to adversaries leveraging cross-site scripting [13, 15], man-in-the-middle attacks [13, 15] or specially crafted HTML-only email phishing forms [6]. In these aforementioned attacks, the credentials are exposed following the autofill process. The impact and effectiveness of these attacks is exacerbated when autofill is performed automatically, without the user's consent in absence of proper security measures [28, 13, 15, 6]. When combined with automatic autofill, injected cross-origin iframes were abused to steal user's credentials under targeted websites [28, 8, 13]. The embedded, potentially invisible, iframe loaded external web content containing input fields of another domain (the targeted website, e.g., a bank). Consequently, naive secret managers have automatically autofilled the user's credentials into the iframe for which the adversary stealthily stole.

Fortunately, secret managers have improved and adopted a stricter policy when dealing with cross-origin iframes [28]. Generally, third-party browser-based secret managers (extensions) do not autofill cross-origin iframes, or at least require user consent in the case of native browser-based secret managers excluding Firefox [28]. Oesch and Ruoti [28] hunted for XSS-safe secret managers [15] (implementing the credential tokenisation strategy discussed in Section 2.5) and unfortunately, no secret manager implemented such defensive strategy. They justify the absence of such XSS-safe secret managers by highlighting the limitation imposed by browsers, the inability of extensions to manipulate the request body to replace the tokens with the genuine passwords[2] [28]. While it is a limitation in one sense, it is a security measure in another sense. It prevents malicious browser extensions from eavesdropping and manipulating HTTP request payloads. Nonetheless, native browser-based secret managers operating with higher privileges have not implemented it either; perhaps because of the common understanding that websites expect a valid password instead of a token to feed into client-side routines, e.g., for the Secure Remote Password protocol [35].

Overall, many of the reported vulnerabilities in previous works have been resolved by third-party browser-based secret managers [32]. However, XSS remains a potential threat along with shortage of security warnings from the secret manager on the key URL or domain name discrepancies between previously encountered form (from which the credentials were created) and the to-be-filled form (into which the

[2] When Stock and Johns' XSS mitigation strategy was proposed, it was possible for Mozilla Firefox extensions to replace tokens with the genuine credentials in the request body.

credentials are filled) [28] — the credential mapping strategy (see Section 2.4.4). For the latter, and from a security point of view, the user should be warned of the potential credential leakage as a result of loose credential mapping.

Having discussed the security of secret managers, their shortcomings due to weak security measures and the persistence of the XSS threat over password security, we make a noteworthy remark. While some secret managers have adopted a strong security policy for HTML Autofill, there is no guarantee that every secret manager would have strong security measures due to their autonomy over implementing HTML Autofill in the way they see fit. Hence, this clearly highlights a security gap which could be improved by introducing a browser routine that ensures strong security policies are met before the delegation of credential transfer to secret managers takes place.

2.7. Synthesis

Having covered the existing frameworks, we take a step back from the high-level outcomes of each framework and notice that each assumed a design pathway. Regardless of the means of communication, PMF and ByPass consider authentication and registration ceremonies in their design, distinguishing between the two ceremonies, while HTML Autofill and the Credential Management API abstract away from account-related ceremonies and focus on credential storage and transmission. This key architectural decision influences the functional outcomes of the streamlined approach. A specialised API enables secret managers to complete account-related operations on behalf of users (e.g., credential recovery). Conversely, a generalised API, one that abstracts from operations and focuses on credentials, might limit secret managers into credential creation without coupling the other steps in an operation (e.g., querying for identity information and handling account status and verification). ByPass is an instance of a specialised architecture capable of facilitating account-related operations such as account deletion. The downside of a specialised API is that it increases complexity since web application developers have to handle the operations and its stages. However, this added complexity is justifiable as a specialised architecture enables secret managers to offer valuable functionality to end-users, providing them with visibility over the type of operation being held and with assistance on completing the operation, including its inner steps.

Table 2.1 highlights the capability gaps in the existing frameworks for secret manager-assisted authentication. Each of the following capability has security or usability benefits.

- **Consistent behaviour.** The framework should not permit inconsistent or unpredictable behaviour to occur; e.g., due to not defining or enforcing a strict credential transfer specification, leaving secret managers to implement it the way they see fit, including the implementations of fallible heuristics (see Section 2.4.5).

Capability	HTML Autofill	Cred. Mgmt.	ByPass
Consistent behaviour	-	●	●
Integrates with web extensions	●	-	●
Consistent security measures	-	●	●
Safe from XSS-password theft	-	-	●
Safe from TPitM-password theft	-	-	-
Secure credential mapping	-	-	-
Supports stateful challenges	-	-	-
Supports credential customisation	-	-	-
Preserves web apps' Auth UI	●	●	-
Preserves users' mental model	●	●	-

● = provides capability; - = lacks capability

Table 2.1.: Comparison of provided (notable) capabilities for a secure and practical secret manager-assisted user authentication framework in proposed frameworks. Absence of a capability in a solution signals a significant gap.

- **Integrates with web extensions.** The framework should officially support third-party secret manager (web) extensions. A crucial feature that would make the framework *open* and not exclusive to first-party (built-in) secret managers (see Section 2.4.2).

- **Consistent security measures.** The framework should enforce a fixed set of security measures prior to the delegation of credential transfer to the secret manager (see Section 2.6).

- **Safe from XSS-password theft.** The framework should be immune to password theft attacks via cross-site scripting (see Section 2.5).

- **Safe from TPitM-password theft.** The framework should be immune to password theft attacks via TLS Proxy in the Middle (see Section 2.5).

- **Secure credential mapping.** The framework should enable a secure app-to-credential mapping strategy for which the secret manager, under the assumed credential mapping strategy, would not divulge the user's credentials to unintended parties (see Section 2.4.4).

- **Supports stateful challenges.** The framework should facilitate interactive authentication challenges capable of maintaining states across the lifetime of the challenge. The traditional password challenge is a stateless challenge whereas a more sophisticated and secure challenge such as the Secure Remote Password [1] protocol is a stateful challenge (see Section 2.4.1).

- **Supports credential customisation.** The framework should be interoperable with generic authentication subsystems that might design an arbitrary credential structure with composite login handles or require multiple credentials per account (see Section 2.4.2).
- **Preserves web apps' Auth UI.** The framework should not eliminate the web application's authentication user interface (see Section 2.4.3).
- **Preserves users' mental model.** The framework should not require the user to cultivate a divergent method to interact with the web application (see Section 2.4.3).

Overall, we argue that all of these capabilities should be offered to realise a practical and secure secret manager-assisted web user authentication framework. We can see from Table 2.1 that for some capabilities, it is offered by one framework but missing in another. For other capabilities, it is lacking from all frameworks. This calls for future work exploring an augmented framework that addresses these (security and usability) gaps while preserving the benefits of the present frameworks.

3. Proposed governance framework

To effectively mediate between web applications and secret managers, we propose Berytus, a browser-based governance framework for programmable account registration and authentication sessions through secret managers. Berytus's experimental design offers two integration pathways for web applications: (1) base integration requiring front-end changes only; (2) complete integration with enhanced security services requiring front-end and back-end changes. In this chapter, we will discuss the methodology, architecture, design and implementation of Berytus.

3.1. Methodology

Our methodology is constituted of three phases: conceptualisation, implementation and evaluation.

Conceptualisation. In essence, our method for conceptualising Berytus is based on the research goals listed in Section 1.2. We will now discuss the three pillars of our conceptualisation method: the positioning of Berytus, the target application scope of secret managers and the means of communication between the web application and secret manager.

1. To mediate the interactions between secret managers and web applications, we positioned Berytus to be *between* the secret manager and the web application. Moreover, we tailored the framework for client-side orchestration to preserve the fundamental notion and application of secret managers as *local* (i.e. client-side) credential management tools.

2. We only consider *browser-based* secret managers, not standalone applications or web-based secret managers. Both first-party (i.e. built-in) and third-party (i.e. extension) secret managers operating within the browser can register with the framework. Hence, placing Berytus in the browser is a natural choice, it is a client-side environment where web applications are loaded and where secret managers predominantly operate.

3. By recognising the need of a programming interface for both parties, the web application and the secret manager, we designed two separate *APIs*. Web developers can leverage the framework's web application-side API and the Web Components/HTML Custom Elements technology [39] to conceive a declarative approach, i.e. a custom markup language, for invoking the framework. Each API is exposed in the appropriate and corresponding execution context

as it is the case with other (standardised) browser APIs, i.e. in the web application's or secret manager's page scripts. Furthermore, the communication flow is a round trip sequence, from the web application to the secret manager, compromised of outbound secret management commands and inbound secret management responses. Hence, in Berytus, secret managers function as processing agents callable by the web application.

Implementation. Following the basic conceptualisation of Berytus as a mediator between web applications and secret managers offering two APIs, one for web apps and one for secret managers, we decided to extend the Mozilla Firefox browser to implement Berytus as a native solution. Extending a browser is necessary to provide a native (first-party) solution for the user, freeing him from the need to install the framework as a plugin. We chose Mozilla Firefox since it is a popular and open source browser. During the early phases of research, the primary goal was to connect web applications with secret managers. As such, we will describe below the method we took to establish a browser bridge between web applications and secret managers.

1. We began by introducing two entry points, one for web applications and one for secret manager extensions. We followed the instructions listed in Mozilla source code documentation to introduce a new API for browser extensions and a new API for web applications.

2. After creating the two points of entries, we needed a service that could connect the web application API implementation with a secret manager extension. Accordingly, we developed a component we call the *Liaison* as a centralised registry that keeps track of installed secret managers and is able to relay instructions to a targeted manager.

3. Following the introduction of the Liaison, both API implementations needed a proxy to reach the Liaison as they reside in child processes while the Liaison is in the parent process. As such, we developed the necessary Mozilla-specific inter-process communication components to allow both API implementations to reach the Liaison.

4. Last, we incorporated a new method in the secret manager extension API to allow secret manager extensions to register with the Liaison. On the other side, in the web application API, we did not incorporate a public method to communicate with a specific manager, rather we enabled web applications to communicate with a *user-selected* secret manager.

This established an appropriate browser bridge between the web application side and the secret manager extension side. From there onwards, we began the design and implementation of the web application API. On the flip side, the secret manager extension API reflected the *commands* defined in the Berytus web application API

as *requests*. To complete the implementation, we developed a sample secret manager extension, Secret*, that integrates with Berytus. Furthermore, we developed a standalone web application containing HTML and JavaScript code to invoke the Berytus web application API and simulate observable authentication and registration sessions, rendering the retrieved information from the secret manager. These implementations allow us to assess the feasibility of Berytus as a paradigm for programmable authentication and registration sessions between web applications and secret managers.

Evaluation. Since our conceptualised framework is designed to address the gaps discussed in Chapter 2, we assume an evaluation method that revolves around those gaps. We determine the soundness of our framework by assessing the gaps that it fills. Ideally, it should offer all of the capabilities listed in Table 2.1. It should be the best of all worlds, and not just another variation which improves in one area but deficient in another. Ultimately, we are assessing the effectiveness of Berytus as a secure and practical paradigm for secret manager-assisted authentication by conducting a security evaluation and a functionality evaluation.

Security evaluation. For the security aspects of Berytus, we evaluate whether Berytus would enable a communication between secret managers and web applications that is immune against prominent attacks. The following covers the steps we take to carry out the analysis:

1. We explore natural attack targets by adversaries on secret manager-assisted web user authentication frameworks. Dissecting the potential goals of such adversaries would shed a light on the crucial aspects that secret manager-assisted web user authentication frameworks should thrive to be secure in.

2. We identify the appropriate attack area of secret manager-assisted authentication frameworks and discuss how attack payloads can be crafted to carry the attack target. By identifying the attack area, we determine which secret manager-assisted authentication frameworks are implicated in the attacks.

3. We infer two attack modes, a passive, eavesdropping mode and an active, instigating mode that can be assumed in the attack payload.

4. We consider well-known and feasible attack vectors applicable in the identified attack area to deliver attack payloads.

5. Finally, we lay out a contextualised threat model and begin evaluating the effectiveness of each considered attack vector and report the possible mitigation strategies either provided by web standards, previous works or in the Berytus framework.

Our security goal for Berytus is to achieve immunity from credential theft attacks. The criteria are listed below as the following capabilities from Table 2.1.

- **"Safe from XSS-password theft":** Immunity from web page-based credential theft attacks.
- **"Safe from TPitM-password theft":** Immunity from network-based credential theft attacks.
- **"Secure credential mapping":** Immunity from phishing-based credential theft attacks.

In the first two capabilities, each of the attack vectors, XSS and TPitM, should not be sufficient on its own to steal the user's reusable credentials. Note, authentication schemes such as one-time password or digital signature-based authentication are safe against credential theft since the transmitted credential is only valid for a single session and cannot be used by the adversary at a later point in time. Nonetheless, we will evaluate Berytus on whether it is safe from credential theft attacks regardless of the authentication scheme including traditional password authentication.

Functionality evaluation. For the functionality aspects of Berytus, we analyse its functional compatibility with generic web applications. Web applications have complete autonomy over streamlining the user experience, user interface and the authentication flow. We argue that a *practical* framework should aim to achieve a high degree of functional compatibility with generic web applications. Achieving interoperability with such web applications implies that the framework is not exclusively applicable to orthodox (or a specific class of) web applications, making Berytus an open, non-restrictive paradigm, in terms of functional compatibility, for programmable authentication or registration sessions. Furthermore, we consider an important user requirement, support for third-party secret manager extensions. Based on our review of present frameworks, we identified noteworthy capabilities of a practical paradigm and summarised them in Table 2.1. As such, our functionality criteria for Berytus are to provide the following capabilities (see Section 2.7).

- **"Integrates with web extensions":** Users can install and register third-party secret managers.
- **"Supports stateful challenges":** Web apps can conduct Secure Remote Password [1] authentication or other stateful authentication challenges.
- **"Supports credential customisation":** Web apps can create credential structures with multiple account identifiers or secrets.
- **"Preserves web apps' Auth UI":** Web apps can design the front-end user experience and interface during secret manager-assisted authentication.
- **"Preserves users' mental model":** Users are not required to cultivate a new method of interaction with web applications.

3.2. Architectural overview

As an orchestrator, Berytus sits between the web application front-end and the secret manager client, operating natively in the browser. Berytus introduces two APIs, a Web API for web applications and a WebExtensions API for secret managers. Essentially, Berytus relays the instructions given by the web application via the Web API to the secret manager through the WebExtensions API. In this section, we will unpack the ingredient technologies, foundational components, secret management-related routines and facilitated services of Berytus.

3.2.1. Ingredient technologies

WebExtensions APIs and Web APIs. *WebExtensions* APIs are a collection of APIs that serve as the primary medium of interfacing with the browser when developing browser extensions. These APIs are available for browser extensions to invoke. They are often described using JSON Schema. Access to certain WebExtensions APIs are controlled via *permissions* requested by browser extensions that may be accepted or denied by the browser depending on the browser policies and user consent. For web page scripts, Web APIs are provided to achieve various goals, e.g., manipulating the contents of a web page or fetching resources over HTTP. Web APIs are often defined using *WebIDL* (web interface description language). Both of these web technologies are meant to be available across browsers. However, some browser vendors may choose not to implement specific WebExtensions/Web APIs.

3.2.2. Components

Let us now introduce the building blocks put forth by our paradigm. Based on some aspects in the landscape covering authentication on the web, we streamline specialised components (shown in Figure 3.1) for our approach.

We conceptualise account-related processes, e.g., authentication or registration, as *operations* where each operation is a series of one-or-more actions, resembling a form with multiple (action) steps. Furthermore, we introduce the concept of a *channel* to reflect an active logical link between the web application and the secret manager. The channel holds the two *actor* objects, one for the web application and one for the secret manager, containing identifying information of each party. Both parties would be able to mutually identify each other by exchanging actors. This is important, e.g., for high-value web applications such as banks to interact only with specific secret managers. And for secret managers to locate the corresponding account records (of the web application) in its database.

There are two actor specialisations. The first specialisation is the Origin Actor and is exclusive for web applications. It reflects the web page's Uniform Resource Identifier (URI). The second specialisation, Crypto Actor, can be used by both the web application and secret manager. It requires the backing of a (cryptographic) signing key. The primary difference between the crypto actor and the origin actor

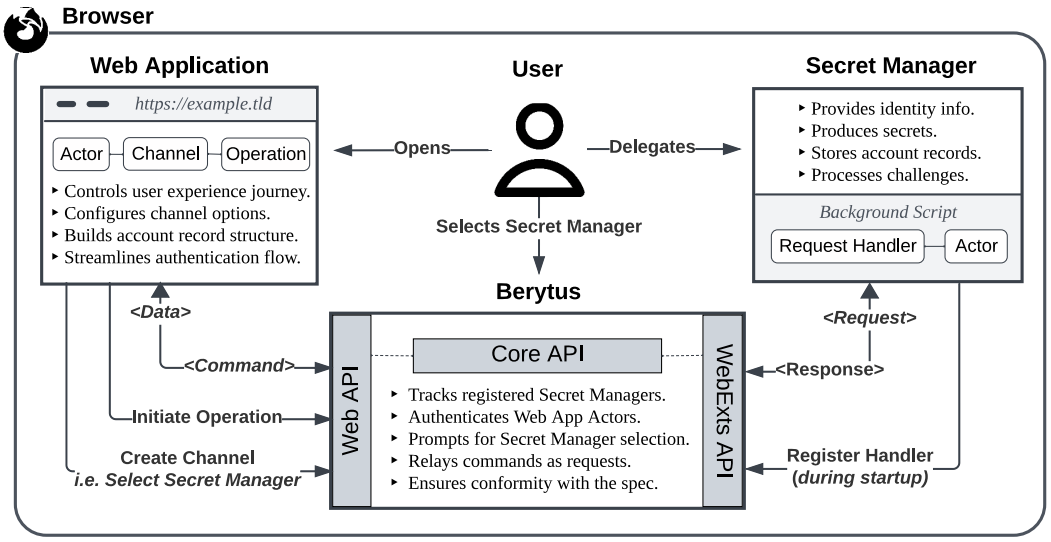

Figure 3.1.: Illustration of the Berytus communication model between the web application and the secret manager along with their components.

is the type of identifying material. Both the URI or public signing key can be used to identify a web application. However, a web application might be accessible through multiple URIs and not bound to a singular, constant URI. Hence, a web application hosted at distinct resource locations will produce distinct origin actors (one distinct actor for each distinct URI). Conversely, if a web application uses a crypto actor, it will construct uniform actors across all of its resource locations. Similarly, if a secret manager creates a crypto actor, it will construct uniform actors across various desktop or mobile computing environments.

Finally, we appoint secret managers to construct a *request handler* function, i.e. callback function, to process the web application's instructions programmatically. It is invoked to process requests such as constructing an account password field. Programmability is a pivotal functionality for secret managers; its absence would impose severe limitations on what secret managers can do.

3.2.3. Routines

In this section, we describe the routines that Berytus is responsible for from a high-level perspective.

Extension access control and managing Secret Manager registration

Berytus introduces a new WebExtensions permission, `berytus`, for controlling access to the Berytus WebExtensions API. This permission effectively induces a two-tier system for extensions: those with and without access to Berytus WebExtensions API, based on user consent. When an extension specifies the `berytus` permission

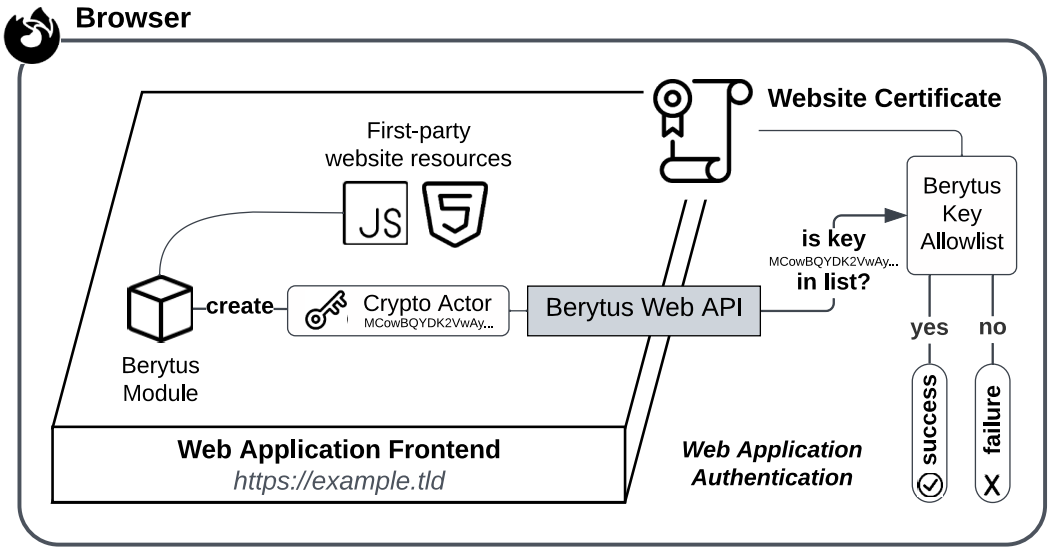

Figure 3.2.: Illustration of the Berytus web application authentication process.

entry in its manifest file, the user will be asked by the browser whether they would want the extension to act as her secret manager, and if the user consents, the extension will have access to the Berytus WebExtensions API.

The WebExtensions API provides two primary methods, one for registration and one for de-registration. The secret manager passes a request handler during registration and it gets stored in Berytus. It is invoked when Berytus Web API calls are dispatched by the web application. The Web API call is transformed into a request and fed to the request handler. The request encapsulates the web application's intent to perform a specific task, e.g. constructing an account password field. Berytus keeps track of registered and running secret managers along with their request handlers. At a future point in time, the secret manager can de-register and the stored request handler would be disposed.

Authenticating Web App Crypto Actors using digital certificates

To validate the authenticity of web apps' crypto actors, we propose a new X.509 v3 certificate extension: Berytus Signing Key Allowlist. The certificate extension specifies a list of one-or-more signing keys that can be used by the certificate's subject (web application). Each list entry contains the following properties (signing key, uris). The uris property indicates under which URIs the signing key can be used. In Figure 3.2, during a crypto actor instantiation in the web application's execution context, the actor's signing key would be validated against the allowlist. If the passed signing key was not specified in the allowlist, an exception will be raised and the actor object would not be instantiated. Overall, at a logical level, the web app crypto actor signing key should be *certified* by a trusted certificate

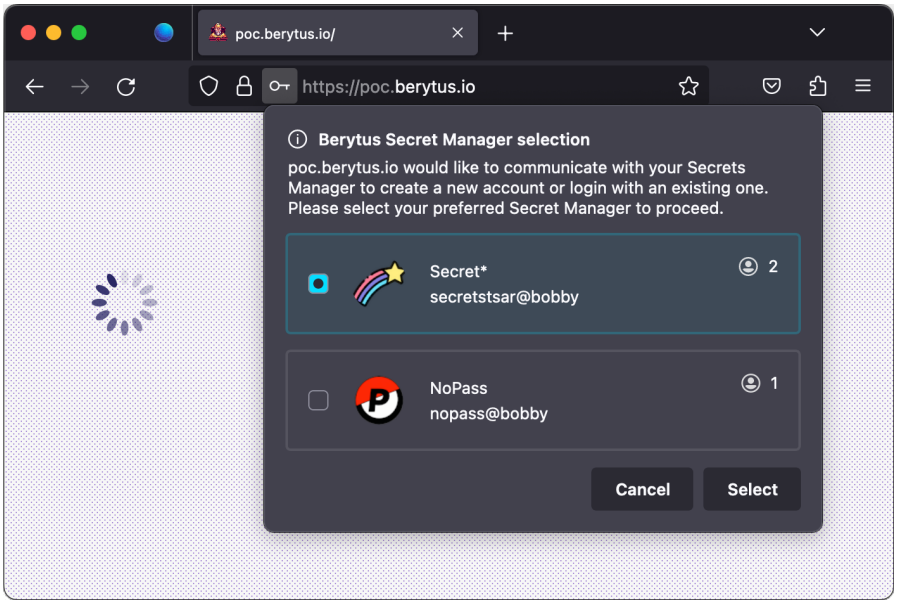

Figure 3.3.: An instance of the Berytus secret manager selection prompt. Each listed secret manager displays the number of registered accounts associated with the web application. Here, domain-based credential mapping was used.

authority.

In the secret manager's execution context, a crypto actor can be instantiated using an *uncertified* signing key; by default, the secret manager's crypto actor is unauthenticated. The status quo, with secret managers being client-side tools, does not facilitates secret manager certificates. Therefore, by default, the secret manager's instantiated crypto actor is not necessarily authentic. However, as we will see in later sections, Berytus facilitates mutually authenticated Diffie—Hellman key exchange which cryptographically authenticates both the secret manager's signing key and the web application's signing key without the need of digital certificates.

Securing channel creation and prompting for Secret Manager selection

Creating a channel is the first step towards secret manager-assisted user authentication. The web application initiates the channel creation process using the Berytus Web API. To open a channel, the web application provides its actor object and its preferred channel configuration settings (if any).

We enforce several security measures when it comes to dealing with channels. First, a channel can only be created from within HTTPS web pages and in top-level browsing contexts, i.e. not in embedded, potentially invisible, web pages (iframes). Second, browsing contexts are restricted to maintain at most one active channel at a time. Third, user approval is necessary to open a channel. This is achieved using

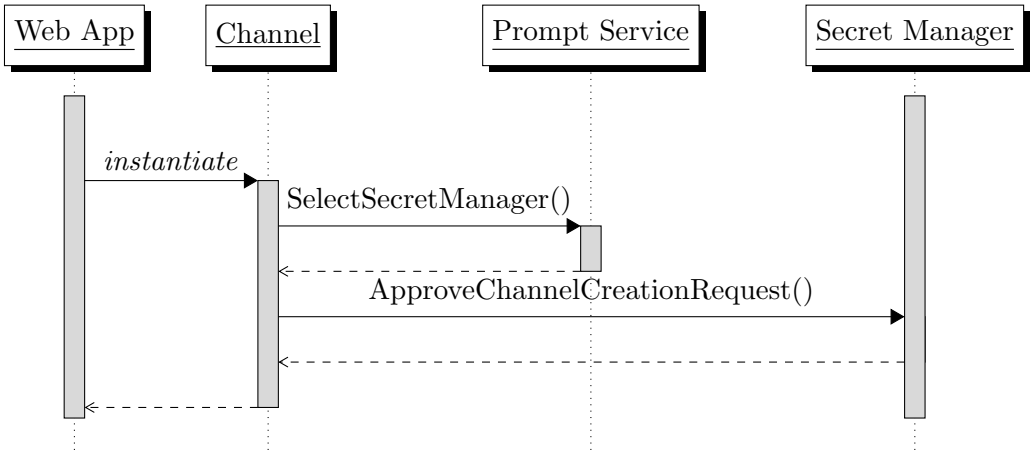

Figure 3.4.: (Simplified) sequence diagram showing the interactions between the web application, channel, a specialised prompt service, and the secret manager during the Berytus channel creation process.

native browser prompts, distinguishing them from other web application prompts. Fourth, a channel is time-bound and cannot be kept active indefinitely. Overall, these security measures are enforced consistently (see Table 2.1) regardless of the secret manager implementation. This gives users additional confidence in the security of the framework, without having to fully rely on the, varying and unregulated, secret manager's internal security policies.

When a channel creation instruction is executed, the user is prompted using the browser UI to select a secret manager from the list of registered secret managers (see Figure 3.3). The secret manager is provided with the web application's actor to determine the number of registered accounts. Apart from choosing a secret manager, the browser UI secret manager selection prompt ensures user consent is granted before opening the channel. After a secret manager is selected, the secret manager is requested to approve channel creation, and if approved, the channel is opened and returned to the web application (see Figure 3.4). Henceforth, one or more account-related operations can be initiated, e.g., registration followed by authentication.

Mediating an Authenticated Key Exchange

A Diffie–Hellman Key Exchange is a cryptographic method designed for a secure key exchange between two parties over a public, potentially insecure, communication medium. Berytus mediates an authenticated key exchange (specifically the elliptic curve-based X25519 [3] combined with authentication of exchanged keys by leveraging the Ed25519 [20] signing key of each party) using the client-side as a medium to communicate cryptographic material passed between the web application and the secret manager. Each party's cryptographic material is generated or

computed off the front-end. The process is as follows:

1. The web application transmits its session key via the front-end to Berytus.
2. Berytus drafts the session parameters, including the passed web app session key and the following properties (session id, HKDF parameters, AES key length).
3. The secret manager receives the session parameters draft from Berytus, includes its own session key into the parameters, signs the session parameters using its crypto actor's signing key and returns the session parameters and its signature to Berytus.
4. Berytus relays the secret manager-signed session parameters to the web application front-end.
5. The web application front-end transmits the secret manager-signed session parameters to a secure site, such as its back-end, to verify the secret manager signature of the session parameters, and if signature verification succeeds, the web app signs the session parameters using its crypto actor's signing key and transmits it to Berytus via the front-end.
6. Berytus relays the web app-signed session parameters to the secret manager. In turn, the secret manager verifies the web app signature of the session parameters, and if signature verification succeeds, the secret manager informs Berytus that the authenticated key exchange was successful.

Subsequently, both parties can compute the shared key using the mutually signed session parameters and leverage it for end-to-end encryption.

3.2.4. Facilities

As a result of the streamlined components and routines, Berytus facilitates the following services:

- **Unified secret management.** By requiring secret managers to register, the browser keeps track of running secret manager and the user benefits from the increased clarity and usability. Users are now able to *elect* a secret manager on a per-session basis. This is a highly sought-after functionality for users interacting with multiple secret managers.
- **Web app-to-credential mapping.** Secret managers can leverage the web app actor's identifying material to distinguish between web applications and perform web app-to-credential mapping. The origin actor is used for domain-based identification and the crypto actor is used for key-based identification. The web application is at liberty to pick from the two actor specialisations. However, by opting-in for a key-based app-to-credential mapping, it ensures

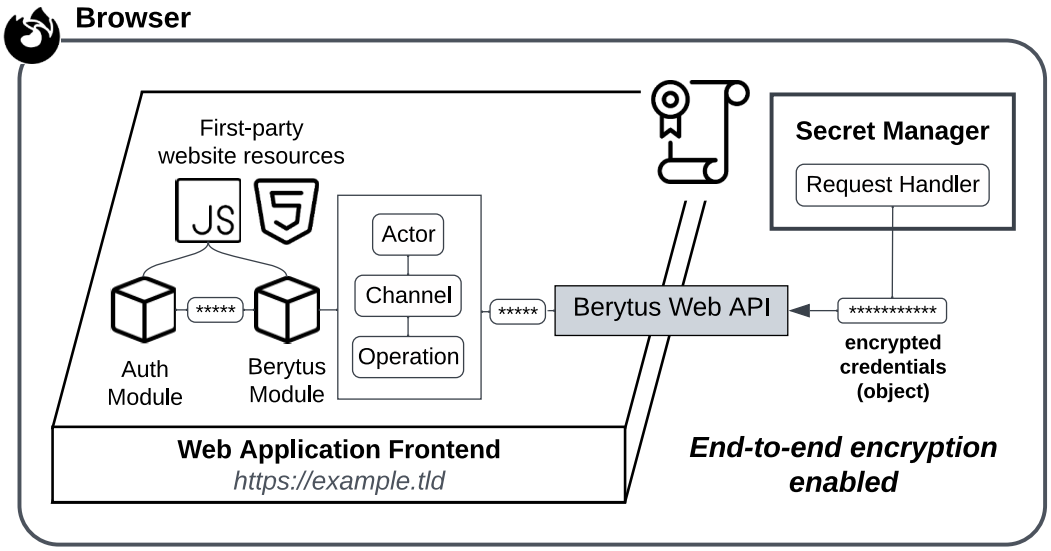

Figure 3.5.: Illustration of the Berytus app-level end-to-end encryption pathway. The secret manager encrypts sensitive information such as credentials before its transmission to the web app frontend. Eventually, the web app frontend transfers the encrypted data to its backend via HTTPS.

forward compatibility for uniform identification with applications outside the web/http realm, e.g., locally installed software, since the application would be no longer necessarily tied to a domain name or URI, but to a more expressive app-specific key.

- **App-level end-to-end encryption.** Berytus facilitates app-level end-to-end encryption between the secret manager and the web application following an authenticated Diffie—Hellman key exchange. Both parties can separately compute a shared symmetric key for app-level end-to-end encryption of secrets passed from the secret manager to the web application and vice versa (see Figure 3.5). Encryption is done, as we call, on the application-level, ensuring confidentiality over the secret manager-to-web app back-end (application) link; as opposed to network-level (TLS) encryption, ensuring confidentiality over the browser-to-web server (network) link. I.e., both parties maintain the shared key and use it for encryption/decryption in a secure site, isolated from the front-end and the web server. Traditionally, with TLS encryption, web servers own the encryption key and the final sink of encrypted data is at the web server process, thereafter the decrypted data is fed to the web app back-end; e.g., via the Common Gateway Interface. Conversely, with app-level end-to-end encryption, the web app back-end owns the encryption key and final sink of encrypted data is, therefore, at the web app back-end instance.

3.3. Design and implementation

Section 3.2 covered the Berytus architecture, the high-level structure, encompassing the foundational components, the communication model and the relevant processes. This section describes in greater details the designed internal components to achieve what Berytus is conceptualised for — harmonising the interactions between web applications and secret managers (through the use of specialised APIs) to undertake account-related operations. Moreover, what is achievable between a web application and a secret manager when undertaking an account-related operation falls down to the assumed *operational design*. Apart from the operations and internal components, we discuss the design pillars of Berytus.

3.3.1. Pillars

Design is an important aspect of any paradigm. It influences the degree of interoperability between systems and determines what is achievable. In Berytus, there are five major design pillars that drove the specification forward:

- **Operation-focused.** Berytus is designed to facilitate programmable authentication or registration sessions through secret managers. We generalise authentication and registration sessions as account-related operations. Hence, in our design, we aim to streamline account-related operations. The designed operation should cover the whole process in practice, and not just one part of it such as password composition and storage.

- **Incremental and modular.** To achieve a modular and incremental[1] approach, we dissect a large process, e.g., an operation, into smaller (atomic) tasks and define a method for each task. This allows web app developers to perform, in between tasks, *dynamic conditional branching* (based on the received data) and execute condition-specific processes.

- **Non-blocking execution.** The secret management tasks should be non-blocking to permit web applications to execute its own processes while the task has not finished yet. Similarly, for secret managers, a received secret management task should not disable them from invoking other functions such as prompting the user to unlock the vault.

- **Customisation and flexibility.** Developers often assume a fixed data organisation structure that might not be interoperable with universal APIs. We recognise this challenge and attempt to facilitate a venue for customisation of relevant structures. In general, when relevant, we aim to provide web app developers with flexibility to assume their own structure or desired paradigm.

[1] In an incremental API, the developer is capable of issuing several instructions (such as creating several password fields) at different points in time. Its counterpart is, what we call, a combined oneshot approach, where the data is bundled into one method call to accomplish a large task.

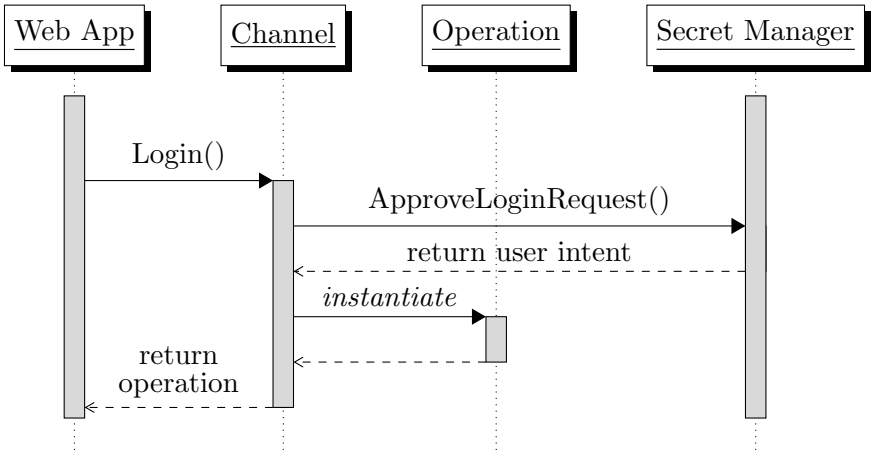

Figure 3.6.: (Simplified) sequence diagram showing the interactions between the web app, channel, operation and the secret manager during the Berytus login operation initiation process. The relayed user intent is either an authentication intent or registration intent.

- **Lock-In-Free.** Prior to the integration with Berytus, a web application has previously streamlined its preferred authentication scheme, e.g., password authentication. A successful integration should not constrain the web application to exclusively use the Berytus orchestration apart from its existing authentication flow (e.g., password authentication using an HTML form). Both users and non-users of Berytus should be able to authenticate themselves under the web application.

3.3.2. Operations

We streamlined two operations, account authentication and account creation. Both of these operations are generalised into a *login operation*. Additional account-related operations can be added in a future version of the Berytus API; it is only a matter of identifying and dissecting the new operation, in a systematic manner, to design a usable flow for both the web application and the secret manager. In this section, we will describe in detail the operational design of the login operation and the relevant processes.

Operation initiation and approval

Before diving into the login operation design, let us introduce the operation initiation and approval pattern. The channel instance is designed to be the entry point of operation initiation. I.e., the web application initiates an operation by interacting with the channel instance (calling one of its methods). Once the operation has been initiated, the secret manager's request handler is invoked to approve the operation creation. The secret manager should resolve this request or reject it — to state

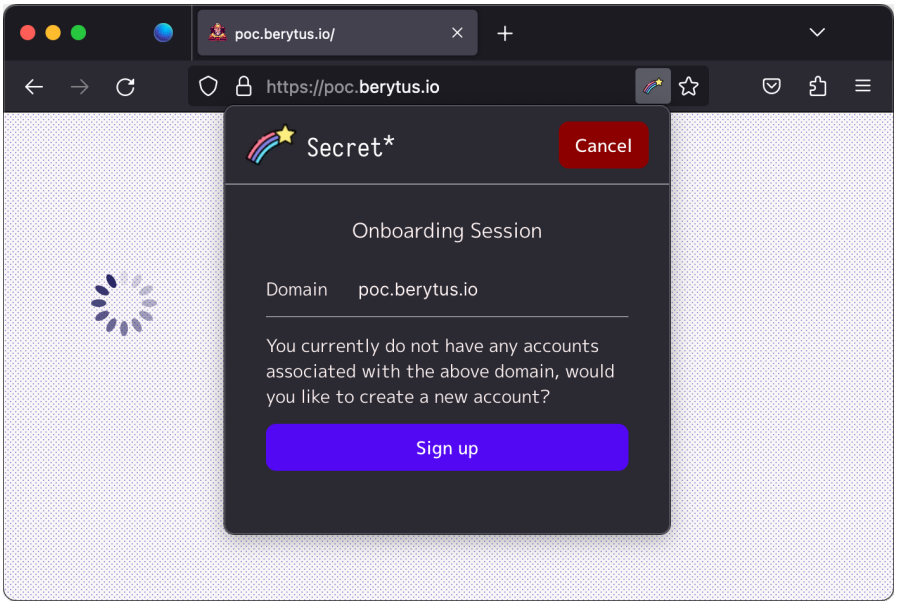

Figure 3.7.: An instance of the Secret* login operation approval prompt. The user does not have any registered accounts, therefore, the only option is to relay an account registration intent to the web app. Here, domain-based credential mapping was used.

the obvious, when the request is resolved, operation initiation succeeds, when it is rejected, it fails. Figure 3.6 illustrates this sequence of instructions for the login operation.

Login operation intent and credential suggestion

Generally, when the user is faced with a login page, he or she proceeds to authenticate his or her existing account or to register a new one. We assume a similar design for the login operation as a branched operation that depends on the user's intent. The user's intent is communicated by the secret manager when resolving the operation approval request, as shown in Figure 3.6. Figure 3.7 shows how the proof of concept secret manager, Secret*, cultivates the user's intent. Note, the secret manager is autonomous to perform any task, including prompting the user for her intent using its own user interface. Autonomy over processing requests also implies that the secret manager is able communicate the intent without the user's explicit confirmation (e.g., in cases where there are no registered accounts in the secret manager's database, it is logical to relay an intent to create a new account). Alternatively, in some instances, the web application may dictate the intent, thereby relieving the secret manager of acquiring the user's intent. This is useful in cases where the web application only allows authentication — therefore, the web app does not need to do conditional branching based on the relayed intent.

Field Type	Sample value at registration	Value producible by
Identity	identifier: `bob123`	App or Manager
Foreign Identity	identifier: `bob@example.org`	App or Manager
Password	password: `RyU8HxsJjk332Dg`	App or Manager
Secure Password (SRP [1])	salt: `0edb53..` verifier: `29c792..`	Manager only
Key	public key: `010101...`	Manager only
Private Key	private key: `010101...`	App or Manager

Table 3.1.: Overview of supported account fields in Berytus including sample field values and the responsible party (web application or secret manager) for producing the field value.

We will now give an overview of the authentication and registration processes under the login operation.

Account creation operation

Inspired by how web applications typically undertake registration, we dissect the registration process into fields and user attributes.

Credential as Fields. An account field resembles a standalone input field in a registration form. A field typically embodies a secret (e.g., a password) or identity (e.g., an email address) value. An account can have multiple fields, and thus a credential can be conceptualised as a set of fields of varying length. This enables unorthodox web applications to streamline account credentials having multiple identity or secret fields. This is achieved by abstracting away from credentials and solely focusing on fields. What constitutes a credential is logically defined by the web application and not Berytus; e.g., a web app might conceptualise a credential as (name, last name, password, date of birth). In turn, this design choice enables web applications to conceptualise a custom credential structure (see Table 2.1) and construct it using a field-level approach.

Field structure. An account field is data object, embodying 4 properties (id, type, options, value). The field id property is necessary to distinguish between the different fields in the account record. The field type property is a static identifier for the specialised field structure. Table 3.1 lists the different field types that Berytus supports. The field options property is used to describe field type-specific settings. For example, in the Password field structure, the field options include a password composition policy which can be specified by the web application.

Field value production. The following field properties (id, type, options) can be seen as metadata specified by the web application. The fourth property, the field

value, is the significant piece of information. The field value can be produced by either the secret manager or the web application. For example, the web application might produce a password field value or delegate the field value production to the secret manager. Producing a field value can be done by prompting the user for an appropriate value (e.g., an email address) or by generating a conforming value (e.g., a password or a key). Berytus does not interfere in the field value production process, however, it streamlines relevant field options to aid secret managers in producing valid values (e.g., a password composition policy). The decision to delegate field value production to the secret manager is made by the web application. However, as shown in Table 3.1, the web application cannot produce a field value for the Key field or Secure Password field for salient reasons, leaving field value production to the secret manager only. The Secure Password field corresponds to the Secure Remote Password [1] protocol where the password should not be divulged to the verifying party. For the Key field, it is created to generate a public key/private key pair, and the private key should remain isolated from the web app. Hence, in both cases, the field values should not be producible by the web application and left for the secret manager to produce.

Field constructors. For each field type, the API defines a specialised constructor to create the data object. The web app passes the field properties as arguments, with the value argument being optional. The web app should leave the field value unspecified if it wishes for, or must ask, the secret manager to produce one.

Registering fields. Once a field object has been constructed, the web application should transmit it to the secret manager to register it with the account. This is done through the `addFields` method of the account operation object which accepts a sequence of field objects in one instruction. The secret manager then processes the received fields and produces field values for fields with an unspecified value. The produced field values are then returned to the web application.

Rejecting and revising fields. Secret managers, similar to users, can produce field values that are theoretically conformable but cannot be usable by the web application. A prominent example is the username field value. A user, or secret manager, might produce a username value which adheres to the username format, however, it cannot be used as it is assumed by another person. For robustness, Berytus allows the web application to reject a produced field value by the secret manager and request a revision. When the secret manager receives a field rejection request, it parses the rejection reason and proposes an alternative value to the web application. This is an exhaustive process, as it is the case with registration forms; i.e. to keep on querying the *producer* until it or she composes a usable value.

Retrieving identity information. Apart from fields, web applications often request identity information such as name and address during registration. Berytus enables

retrieval of common user attributes (based on the OpenID Standard Claims [12, Section 5.1]). Additionally, Berytus facilitates storage of custom user attributes and, if predefined, their retrieval into and from the secret manager. Retrieval of predefined custom user attributes is mostly applicable between mutually acquainted web application and secret manager pairs. The user attribute structure has three metadata properties (id, mime, info). Hence, an acquainted web application and secret manager pair can agree on additional set of user attribute ids to exchange further identity information not specified in the Berytus API. Alternatively, custom user attribute definitions could be streamlined and put forth by the industry in a conventional manner.

Account versioning and categorisation. Web applications can set a version and a category on the account record. Account versioning aids web applications in tracking and migrating account records as their authentication system evolves. Although not streamlined, a specialised account-related operation could be designed and leveraged to update account records, e.g., to create new fields and migrate the record to the latest version. Account category is used as an arbitrary account type identifier, e.g., a user role identifier. The account category can be specified as an option in the account authentication operation to assist the user in selecting an account of the desired category (role).

Verification and account status. Once all of the necessary data has been gathered, the web application might choose to perform additional verification checks before creating the account in its back-end database. Verification checks could be related to the Foreign Identity fields specified in the account record, e.g., codes sent to the user's email for email address verification, or could be unrelated to Berytus, e.g., Face ID verification. To distinguish between back-end account creation pre- and post-verification, Berytus allows the setting of an *account status* in the registration operation. The registration operation does not tackle conventional account verification methods (e.g., email or phone verification). The default account status is `Pending` and it implies that the account is pending creation and not necessarily reserved or stored in the web application database. From the user's perspective, the account is created once the information has been transferred and she is prompted for additional verification checks. Therefore, it is important to establish an account status for account records not just for compatibility between varying web application back-end implementations, e.g., whether account record should be stored in its back-end prior verification or not, but also to avoid confusing the user.

Saving the account and transitioning to account authentication. To complete the registration operation, the web application instructs the secret manager to save the account record into its database. The operation commitment signature is optionally passed along the `save` command. The signature's message content embodies the account record data, signed by the web application signing key following a successful

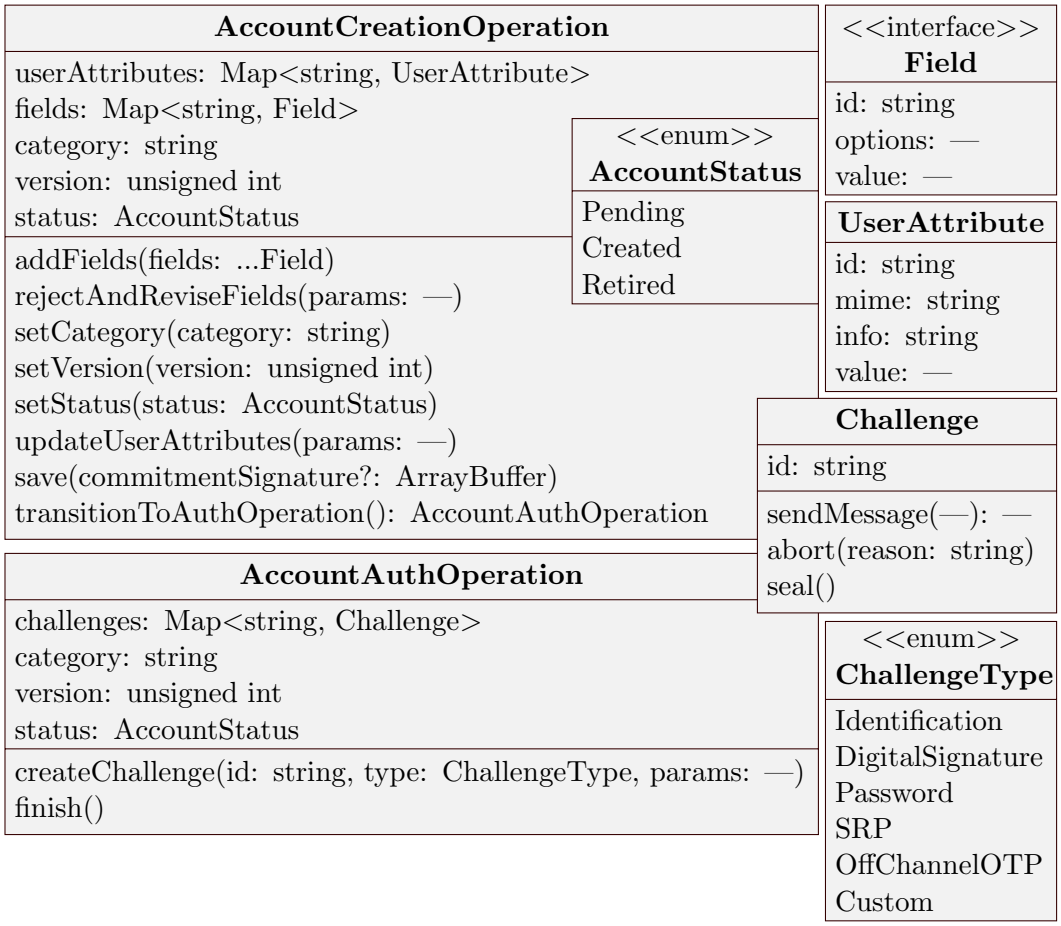

Figure 3.8.: Overview of the account creation operation and account authentication operation class diagrams in the Berytus Web API showing class attributes followed by class methods. For the sake of brevity, we omitted showing unnecessary details.

account record insertion onto its database. This ensures that saved account records in the secret manager are valid and not just created by any JavaScript code residing in the web application. After a successful `save` command execution, the closed operation can be transformed into an account authentication operation for the created account record.

Account authentication operation

The account authentication operation encompasses the steps related to user authentication on the web. At a minimum, a web application typically employs two challenges, an identification challenge and an authentication challenge. In the same spirit, we abstract this instance to streamline a *challenge-based* authen-

tication paradigm. The authentication operation permits the creation of multiple challenges. However, only one challenge can be active at a time.

Account selection. Clearly, to process an authentication operation, the secret manager must assume a registered account. Berytus does not interfere in this matter, albeit its act of non-interference is a design element as well. It is expected that the secret manager would prompt the user to select an account using its own UI facilities. In Secret*, account selection occurs in the login operation approval process and if the user selects an account, an intent to authenticate is relayed.

Approving the challenge. Once a web application initiates a challenge, Berytus contacts the secret manager to approve the challenge initiation request (see Appendix Figure B.1). Following its approval, the web application can begin sending challenge messages to the secret manager.

Challenge communication flow. The challenge is designed to facilitate passage of a sequence of messages between the web application and the secret manager. This model enables implementation of multi-step (or stateful) challenges such as the Secure Remote Password protocol [1]. Multi-step challenges are challenges where more than one message is sent back and forth between the web application and the secret manager. Once the secret manager receives a challenge message, it can respond to it, returning the appropriate data to proceed with the challenge. This challenge messaging pattern is illustrated in Appendix Figure B.2.

Supported challenges. Figure 3.8 lists the supported challenges in the `ChallengeType` enumeration class diagram. `Identification` is marked as challenge in the sense that the user, supported by the secret manager, must produce valid identification material, e.g., a valid (existing) email address. For authentication challenges, Berytus supports digital signature-based authentication (`DigitalSignature`), password authentication (`Password`), Secure Remote Password Protocol [1] authentication (`SRP`) and off-channel one-time password authentication (`OffChannelOTP`, e.g., email/phone one-time password). Furthermore, web applications can initiate custom authentication challenges by specifying the challenge type `Custom`, a challenge id matching the glob pattern `custom:*` and a messaging JSON Schema, used to validate the passed messages, in the challenge parameters. Hence, an acquainted web application and secret manager pair can organise custom authentication challenges at run time.

Closing or aborting the challenge. When all the messages have been sent and if a satisfactory point has been reached by the web application, it can close (seal) the challenge to implicitly imply challenge success. If a satisfactory point was not reached or cannot be reached during the challenge, the web application can abort the challenge by providing an abortion reason code. Appendix Figure B.2 also shows

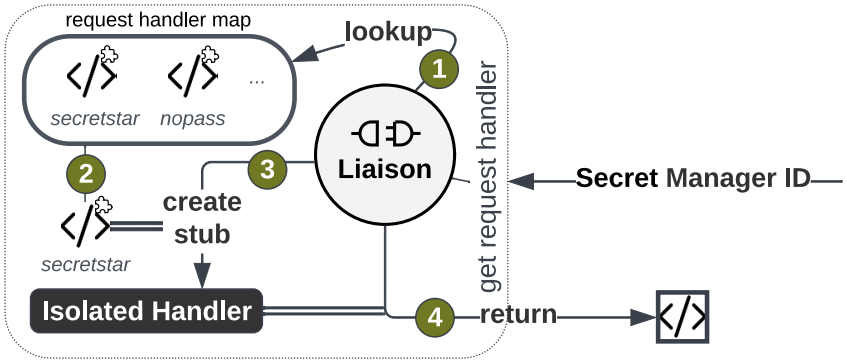

Figure 3.9.: Illustration of the request handler isolation process in the Liaison's get request handler function. The underlying request handler is tucked away in a newly created (not-shared) Isolated Handler instance.

challenge abortion or closure. In some controversial instances, web applications mask whether a user exists or not during the authentication operation. In Berytus, this can be achieved by closing the `Identification` challenge, implicitly implying success, and aborting the following authentication challenge, signalling an incorrect credential error.

After covering the account creation operation and account authentication operation, we make note of a salient design element, granularity. Processes are dissected into atomic tasks. This bestows valuable, low-level access to the inner workings of the framework. Wrapper libraries can be introduced to alleviate web app developers from writing common boilerplate code.

3.3.3. Implementation

To demonstrate the feasibility of our framework, we streamlined and implemented a set of internal browser components for the Mozilla Firefox browser. Additionally, we developed a sample Berytus-compatible secret manager. All of the project artefacts are available at `https://github.com/alichry/berytus`. Build instructions are provided as well as binaries to quickly experiment with the Berytus framework.

Internal components

The following list of internal components were designed to fit into the Mozilla (v116.0a1) architecture. The internal components are grouped together to form the core module. In essence, the core module is responsible for the routines specified in Section 3.2.3 and other functional requirements. The core components are implemented in TypeScript/JavaScript. The Web API is implemented in C++ and partially acts as a wrapper, forwarding instructions to the secret manager by interacting with the core components. This design can be adapted into other browser architectures.

- **Liaison.** The central registry residing in the parent process that holds a list of all registered request handlers. This is a singleton instance. Privileged code wishing to interact with a specific secret manager would need to query the Liaison to retrieve its request handler.

  As shown in Figure 3.9, the queried request handler is wrapped by a unique (not shared) safe handler stub that protects against manipulation of the underlying request handler. The stub holds a reference to the underlying request handler (or a resolver which resolves with the request handler) in a private member, thereby isolating the underlying request handler. The stub implements all the methods defined in the request handler interface. In each implemented method, the underlying request handler is accessed through the private member and the corresponding method is called — i.e. the stub is nothing but a proxy isolating the underlying request handler. This is to protect against malicious JavaScript adversaries attempting to manipulate the exposed request handler. When malicious JavaScript code re-assigns the exposed, public, request handler method of the safe handler stub to point to a malicious function, only that unique safe handler stub instance will be affected. I.e., since the safe handler stub is not shared and the underlying request handler is encapsulated, the underlying request handler is not affected, nor is all of the other retrieved safe handler stubs. In the absence of this, JavaScript, privacy encapsulation, malicious in-browser JavaScript code can override the presumed request handler of the targeted secret manager extension with a newly crafted one to impersonate the target secret manager and potentially receive sensitive information. Note, we assume that the malicious in-browser JavaScript code cannot disable the privacy encapsulation security property of the native, running JavaScript engine.

- **ExtensionAPI.** This module embodies the core logic of the WebExtensions API implementation, enabling an extension to register its request handler with Berytus and process instructions programmatically. When an extension interacts with the Berytus WebExtensions API namespace, two instances of ExtensionAPI are created, one in the parent process and one in the child process. Once the extension's background script registers its request handler, the child instance maintains a reference to its handler and communicate with its parent counterpart to register with the Liaison. As the request handler is a function, it is not serializable and cannot be message-passed through IPC. Therefore, the parent ExtensionAPI instance will construct a handler of its own, passing it to Liaison and registering it on behalf of the extension, and will turn method calls into events that are passed to its child counterpart. The child, upon receiving an event from its parent counterpart will invoke the underlying request handler of the extension. This process is illustrated in Figure 3.10. Last, the ExtensionAPI exposes two additional methods (apart from registration and de-registration) for the secret manager extension, one to resolve requests and one to reject them. Currently, when the secret manager

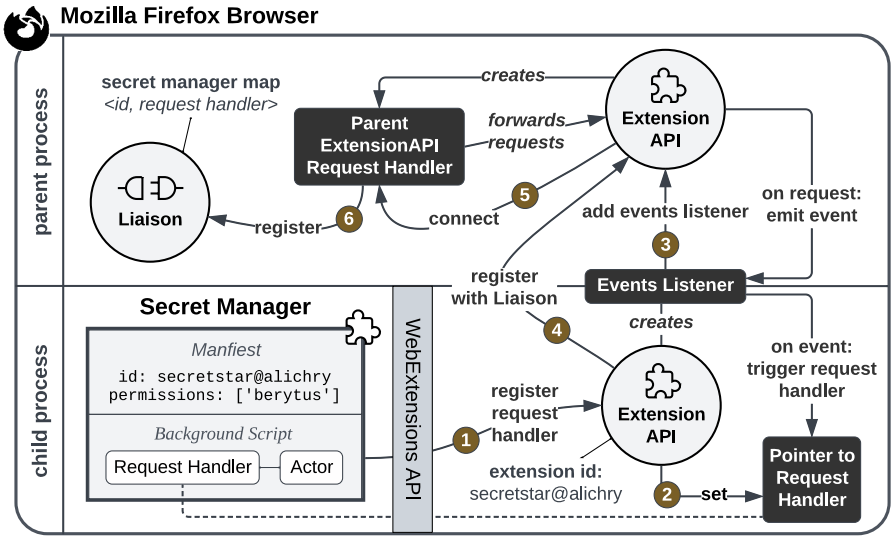

Figure 3.10.: Overview of component interactions during the secret manager's request handler registration process.

receives a request, it has two options of processing the request. First, the request handler returns a promise that either resolves or rejects it. Second, the secret manager extension calls, from within its execution context, the resolve/reject request method to process the request. This is used in instances where the request handler, residing in the background script, would need to open the popup page to process the request. The popup page would leverage the available resolve/reject request method in its execution context to process the pending request.

- **Agent.** Since the Liaison resides in the parent process, child processes cannot directly interact with the Liaison nor the secret manager's request handler. The Agent facilitates access to the effective request handler from the child process through the use of IPC under the hood. The Agent configures inter-process communication with its parent counterpart, allowing for proxied interaction with parent components from within the child process. The purpose of the Agent is twofold. First, the Agent is responsible for relaying request handler invocations to the targeted extension, by consulting with the Liaison, regardless of whether the Agent is instantiated in the parent or child process. Second, the Agent can aggregate several invocations on multiple extensions' request handlers into a single response; e.g., merging all responses into a list (see step 5 in Figure 3.11). Ultimately, code residing in the child process can leverage the Agent to invoke the Secret Manager's request handler. Note, if the Agent was instantiated in the parent process, message passing is not needed; the behaviour and interface is consistent re-

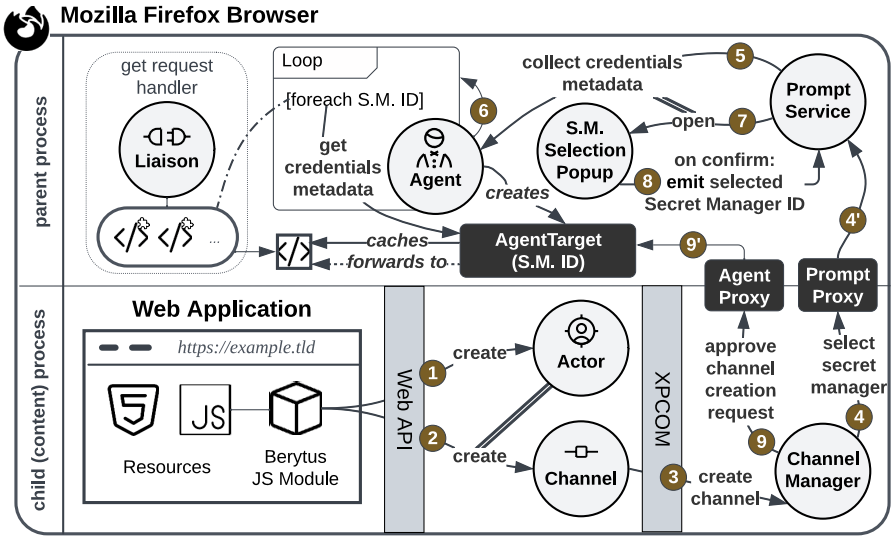

Figure 3.11.: Overview of component interactions during the channel creation process.

gardless of where the instantiated Agent resides.

- **ChannelManager.** This module creates and tracks channels in the child (content) process. The ChannelManager ensures that at most one channel is active under a browsing context/window. Furthermore, it checks whether the browsing context for which the channel is being created is a top-level browsing context, i.e. it rejects channel creation for iframes. Currently, in Mozilla/Firefox, multiple unrelated browsing contexts, e.g., browser tabs, can co-exist within the same child process. To provide additional security, ChannelManager does not expose a mean to fetch channel objects, created channels are returned to the original caller and saved into a `WeakMap`. Hence, created channels by the ChannelManager are isolated from malicious actors within the same child process.

Secret manager implementation

Since we are introducing a novel WebExtensions API for secret managers, developing a secret manager that integrates with it is necessary to demonstrate feasibility. We developed Secret* (Secret-*star*), a secret manager that integrates with the Berytus WebExtensions API.

Secret* has three compartments: (1) The background script, instantiating a request handler and registering it with Berytus to process secret management instructions; (2) The user interface facility, hosting the extension pop up page and external windows to prompt the user to approve operations, provide necessary input and configure secret management settings; (3) The storage facility, to store and re-

trieve data when processing requests. We use Dexie.js as a wrapper for IndexedDB; a client-side (browser) storage facility. We use React/TypeScript as the UI framework for Secret*. The background script is also implemented in TypeScript.

In Secret*, the request handler saves the incoming request data (which includes the request parameters, e.g., document URI, session ID, public keys, etc.) into the database and opens the user interface page, either through the popup page or an external browser window. Choosing between the popup page or an external window is a build-time configuration setting. Either way, the request gets fulfilled from within the UI page; it is possible since the WebExtensions APIs (including Berytus) are also available in the extension pages. Secret* has a configurable setting "seamless onboarding". When enabled, Secret* would not require explicit confirmation of every request but will process them automatically when possible. At a minimum, operation approval and transfer of identity information always require user confirmation. For the remaining requests, unless user input is necessary, they would be processed automatically without requiring explicit user confirmation.

To actualise a rounded implementation, Secret* has additional secret management pages. Secret management pages include an account list page, an account record page and setting configuration page (where identity information is inputted). Lastly, in the prototype, Secret* serves a sample web application that is not intended to be part of Secret* but runs an end-to-end authentication or registration simulation using the Berytus Web API.

3.3.4. A minimal working example

The following listings embodies a minimal working example to conduct a login operation. In Listing 3.1, the web application creates the channel instance and starts the login operation. This would prompt the user to select a secret manager (Figure 3.3) and to approve the login operation, relaying her login intent (Figure 3.7).

```
/*! Domain-based credential mapping */
const originActor = new BerytusAnonymousWebAppActor();
/*! Alternatively, for key-based mapping, e.g.: */
const cryptoActor = new BerytusCryptoWebAppActor(
    'MCowBQYDK2VwAyEAamy324oIpAIek6KAwHuhIvbpLUq4x6FB33eyZkEeN9w='
);

const channel = await BerytusChannel.create({
    webApp: originActor
});
const operation = await channel.login({
    requiredUserAttributes: {
        name: true,
        picture: false,
        gender: true,
        birthdate: true,
        address: true,
    },
});
```

```
20
21  if (operation.intent === 'Authenticate') {
22    /*! handle authentication */
23  } else { /*! => operation.intent === 'Register' */
24    /*! handle registration */
25  }
```

Listing 3.1: A working example of creating a channel and initiating a login operation.

Next, the web application should branch out depending on the user's login intent.

Account creation. Assuming the intent is to register, Listing 3.2 shows how a web application can create and register a username field, an email field, a secure (remote) password [1] field and a password field. See Appendix Listing A.1 for creation of all supported fields.

```
1  const fields = await operation.addFields(
2      new BerytusIdentityField("username", {
3          private: false,
4          humanReadable: true,
5          maxLength: 24,
6      }),
7      new BerytusForeignIdentityField(
8          "email",
9          {
10             private: true,
11             kind: "EmailAddress",
12         },
13         "bobby@example.org", /*! Web app-produced field value */
14     ),
15     new BerytusSecurePasswordField("srp", {
16         identityFieldId: "username",
17     }),
18     new BerytusPasswordField("password", {
19         passwordRules: "minlength: 16;", /*! Apple's password rules format */
20     }),
21 );
22 fields.username.value; //! => (e.g.) "bobby"
23 fields.email.value; //! => "bobby@example.org"
24 fields.srp.value; //! => (e.g.) { salt: "0edb53...", verifier: "29c792..." }
```

Listing 3.2: A working example of creating and registering account fields.

Similar to a registration form, the web application should validate provided values, see Appendix Listing A.2 for a sample exhaustive validation of the username field. Finally, following field value validation, the web application should close the operation by instructing the secret manager to save the account (Listing 3.3). Optionally, before closing the operation, the web application can set an account version or category. Once the account creation operation has been closed, with the account saved in the secret manager, the web application can request a transition to an account authentication operation for the newly registered account.

```
await operation.setVersion(1);
await operation.setCategory("Employee");
await operation.setStatus("Created");
await operation.save();

const authOperation = await operation.transitionToAuthOperation();
```

Listing 3.3: A working example of setting account metadata, saving the record and transitioning into an authentication operation.

Account authentication. Assuming the user's intent is to authenticate, or if the web application requested a transition from an account creation operation to an account authentication operation, Listing 3.4 shows how a web application can start and process an identification challenge and a password challenge.

```
/**! @var accountExists & login - from the webapp codebase. E.g.: */
const accountExists = () => false; const login = () => true;

const idCh = await operation.createChallenge("id", "Identification");
const { payload: { username, email } } = await idCh.sendMessage(
  {
    name: 'GetIdentityFields',
    payload: ['username', 'email'] /*! identity field ids */
  }
);
username; //! => "bobby"
email; //! => "bobby@example.org"

if (accountExists(username)) {
    await idCh.abort("Identification:IdentityDoesNotExists");
    throw new Error("User failed to pass identification challenge");
}

await idCh.seal();

const passCh = await operation.createChallenge("pass", "Password");
const { payload: { password } } = await passCh.sendMessage(
  {
    name: "GetPasswordFields",
    payload: ["password"] /*! password field ids */
  }
);
password; //! => eq45asp0d...

if (! login(username, password)) {
    await passCh.abort("Password:IncorrectPassword");
    throw new Error("User failed to pass password challenge");
}
await passCh.seal();
```

Listing 3.4: A working example of conducting identification and password challenges.

Finally, as shown below in Listing 3.5, the account authentication operation can be closed to communicate authentication success to the secret manager.

```
1  await operation.finish();
```

Listing 3.5: Closure of the account authentication operation.

This concludes the minimal working example for account creation and authentication operations. For the sake of brevity, we have not shown examples of conducting a Secure Remote Password authentication challenge or one-time password challenge, however, the pattern is identical. Note, this challenge messaging instruction pattern is unnecessarily verbose. An improvement for this API draft could be the introduction of specialised challenge interfaces embodying the various send message parameters as explicit methods.

4. Evaluation

In this chapter, we will evaluate Berytus's security aspects, covering the offered benefits and available mitigation strategies, and its functionality aspects, covering functional compatibility and integration effort.

4.1. Security evaluation

To evaluate the security aspects of Berytus, we will uncover its security benefits and perform a security analysis of prominent attacks in the secret manager-assisted web user authentication space against it. When relevant, we will compare Berytus with HTML Autofill, Credential Management API and ByPass.

4.1.1. Security benefits

As an orchestrator, Berytus is able to provide security services to web apps and secret managers, including authentication of web apps and mediation of authenticated key exchanges. In this section, we discuss the security benefits provided by Berytus.

Protection from web application impersonation. Safeguarding against imposters is an important security measure in many systems. In secret manager-assisted authentication, the secret manager must not transfer the user's credentials to unintended parties, including deceitful or impersonating web applications. Berytus's web app authentication routine (Figure 3.2) assures the secret manager that the web app party it is communicating with is legitimate. The condition for web application authentication to occur is that the web application opts into using a crypto actor backed by an app-specific signing key. This enables a secure web app-to-credential mapping strategy using cryptographic keys as indices; serving as a superior alternative to the tried-and-busted domain-based credential mapping. With *secure credential mapping*, the user does not have to manually copy his credentials to the clipboard (decreasing the risk of credential theft via clipboard attacks discussed in Section 2.5) and the secret manager would not suggest credentials for irrelevant or deceitful web applications (mitigating against phishing attacks including ones in a forum context or due to domain name reuse discussed in Section 2.4.4); all due to an accurate credential mapping strategy. By shifting from domain names to authenticated app-specific keys, secure credential mapping not only increases security, but also maintains usability, addressing the previously discussed interaction problems (Section 2.4.5) related to domain-based credential mapping (e.g., multiple

domain usage by a single web application, ensuring the corresponding credentials are suggested regardless of where the web application is hosted). There are two security levels for secure credential mapping:

Secure credential mapping level 1. The first line of defence is certificate-based authentication of the web app signing key. To assume a crypto actor signing key, the web application must have a certificate that includes the Berytus Signing Key Allowlist extension (see authenticating web app actors in Section 3.2.3). Therefore, if a web application tries to assume a signing key that is not defined in the Berytus Signing Key Allowlist extension of the certificate, web app authentication fails. In essence, the Certificate Authority is the responsible body for validating certificate issuing requests which includes proof of possession validation of Berytus Signing Keys. Thus, in this level, web app authentication relies on the security of the public key infrastructure model.

Secure credential mapping level 2. The second line of defence is cryptographic authentication of the signing key. Signing key authentication can also occur during an authenticated Diffie—Hellman Key Exchange between the web application and the secret manager. In the key exchange process (see Section 3.2.4), the web app session parameters signature serves as the present proof that the web application owns the assumed signing key. Hence, the secret manager can verify the web app-signature of the session parameters, asserting whether the web application owns the assumed signing key. If signature verification fails, the secret manager would not transmit the credentials. This could be an attack detection mechanism; the secret manager was assured that it is communicating with the legitimate web app using certificate-based authentication (level 1), however further verification using cryptographic authentication of the signing key indicates that the web app does not own the assumed signing key (level 2).

Protection from eavesdroppers. Another crucial security measure is mitigating against eavesdroppers. The secret manager might indeed be communicating with the legitimate party, however, an adversary in the middle could intercept the communication link and steal sensitive information such as credentials. The web app client-side (front-end) context, the medium for web application—secret manager communication, is a hazardous area in terms of security. Attack vectors such as cross-site scripting (XSS) enable an adversary to inject scripts into the client-side context. Therefore, it is not realistic to consider the client-side context as inherently secure. Due to the high security risk of using the client-side as a medium, Berytus facilitates app-level end-to-end encryption between the web application and the secret manager following an authenticated Diffie—Hellman Key Exchange, specifically X25519 [3, 9]. In Berytus's app-level end-to-end encryption, the final sink of ciphertext is at the web application back-end. The ciphertext is decrypted off the web application—secret manager communication context, in the web application

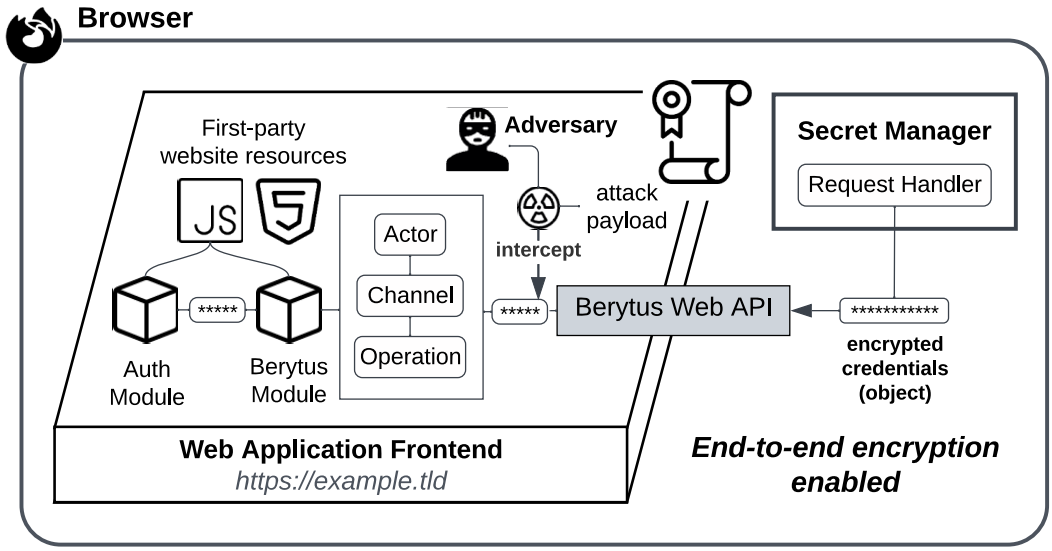

Figure 4.1.: Illustration of a front-end adversary eavesdropping over the web application—secret manager channel using a purposely crafted attack payload. The adversary cannot steal the credentials since it was encrypted off the front-end by the secret manager.

back-end. Hence, any adversaries eavesdropping over the transmitted credentials whether on the web app client-side or on the network-side would not be able to steal credentials. This is illustrated in Figure 4.1.

Safe password-entry interface. Ruoti *et al.*, motivated by the persistence of passwords following years of scholarship on alternative authentication schemes, suggest a formula to strengthen password authentication: "strong password protocols and safe password entry" [18]. In a strong password protocol such as the Secure Remote Password (SRP) [1] protocol, the user's password is not communicated in the clear during user authentication. Instead of the clear password, a zero-knowledge proof is transmitted, permitting the web application to verify that the user has knowledge of the password without her conveying it. To compute the zero-knowledge proof, the user has to enter the password on the web app client-side. Therefore, while the password is not transmitted to the web app server-side in the clear, it is still accessible on the web page before the proof gets computed. This is problematic since such strong password protocols on their own are insufficient to combat XSS-credential theft attacks; the password is temporarily accessible in the clear on the web app client-side. Ruoti *et al.* recommend using a safe password entry interface, isolated from the web page, such as the browser interface to enter the password and compute the proof [18]. Berytus supports a strong password protocol such as SRP and facilitates a safe password entry interface isolated from the web page, the secret manager's interface. This combination mitigates against

Criterion	Cred. Mgmt. API	HTML Autofill¶	ByPass	Berytus
HTTPS only	●	●	●	●
Top-level document only	●	●	/	●
Domain-based credential mapping	●	●	●	●
Web App Authentication	-	-	-	●
Key-based credential mapping	-	-	-	●
Authenticated Key Exchange	-	-	-	●

¶: we assume the most theoretically secure secret manager under HTML Autofill.
● = supported; - = not supported; / = not applicable;

Table 4.1.: Comparison of supported security policies and services.

XSS-credential phishing without the need to use end-to-end encryption.

4.1.2. Attack targets

We identified the following attack targets against users utilising Berytus or other solutions in the secret manager-assisted authentication space:

- **Credential theft.** The attacker steals the credentials of an account belonging to the targeted web application.
- **Identity information theft.** The attacker accesses the user's personal information stored in the secret manager.
- **User authentication hijacking.** The attacker is not necessarily interested in the user's credentials, rather he captures valid (potentially encrypted) authentication request payloads to authenticate himself as the user under the targeted web application (impersonation of the user to the web app).

4.1.3. Attack area

Berytus being a browser-based framework, the attack area is naturally the client-side. The attack payload, to carry on a target (see Section 4.1.2), could reside in a web application front-end, in a secret manager extension, in a malicious browser extension or in the privileged execution context of the browser. This is also the case for the other browser-side solutions, HTML Autofill and the Credential Management API. ByPass is safe from attacks in this area as it is designed for server-side integration without the involvement of the web app front-end. However, in a later section, we will highlight a security shortcoming of ByPass which Berytus solves.

4.1.4. Attack payload

In Berytus and the Credential Management API, communication with the secret manager (storage) is performed through Web APIs. Any means of instruction capable of interfacing with Web APIs is plausible for crafting an attack payload against Berytus or the Credential Management API, e.g., JavaScript. For HTML Autofill, credential or identity information is delivered to HTML input fields. Hence, any means of harvesting the transmitted data in input fields is feasible to craft a payload. JavaScript can be leveraged to create HTML autofill-able input fields using the DOM Web API and to steal the transmitted autofill data. Ultimately, an attack payload crafted using JavaScript is sufficient to attack secret managers implementing HTML Autofill or the Berytus API or to attack the browser credential storage when the Credential Management API is available. Thus, we will assume JavaScript code to be, effectively, the driving code of the attacks.

4.1.5. Attack modes

The attack payload can be designed to assume two attack modes: active mode and passive mode.

In active mode, the payload invokes the API (Berytus API or Credential Management API) to initiate credential transmission. The attacker directly invokes the corresponding API and has liberty over specifying the desired arguments to pass. In HTML Autofill, active mode could be conceptualized as the render of additional input fields for which the secret manager would autofill. Active mode might cause a user interface dialog to open which could alarm the user if he is not expecting it. Conversely, passive mode is a stealthy approach for credential theft.

In passive mode, the attacker patiently awaits for the credential transmission process to begin and intercepts it. Techniques such as JavaScript "monkey patching", i.e. augmenting the semantics of APIs at run-time, may be employed to intercept the data passed between a secret manager and a web app through Web APIs [23, 21]. Monkey patch-interception is transparent and done on the fly; the first-party scripts of a web application would not be able to notice it. In HTML Autofill, passive mode can be assumed by patiently awaiting for an existing input field to be filled with the password. Similar to monkey patch-interception, neither the web application nor the user would be able to detect adversaries (under typical instances) utilising the DOM API to access the autofilled input fields.

4.1.6. Attack vectors

After crafting the attack payload, the question then becomes, how to transmit this payload? I.e., what would be the potential attack vectors for this payload? We consider the following attack vectors and group them into three categories.

- **JavaScript Code Injection.** This group includes attack vectors capable of injecting malicious JavaScript code into legitimate websites. Cross-site

scripting (XSS), TLS Proxy in the Middle and malicious browser extension code injection (ExtInj) were covered in Section 2.5. Note, TPitM can be leveraged to inject code into a loaded web page (TPInj). These attack vectors are relevant for all browser-side solutions and individually sufficient to deliver the attack payload into legitimate web applications.

- **Certificate Spoofing.** This group encapsulates attack vectors capable of assuming an illegitimate certificate to impersonate another web application. We consider one attack vector under Certificate Spoofing which is only applicable for Berytus web app authentication instances, App key Impersonation (AppImp). In our proposed X.509 extension, a Berytus signing key allow list (see web app actor authentication routine in Section 3.2.3) can be defined to facilitate key-based credential mapping. An attacker can spoof a certificate containing the same Berytus public signing key as the one of a different web application. For example, this certificate could be signed by a corrupt or negligent certificate authority.

- **Composite Vector.** This group represents instances where multiple attack vectors are leveraged at once. We combine two independent attack vectors, Certificate Spoofing and DNS Poisoning, to form a composite attack vector, website impersonation (SiteImp). In website impersonation, the attacker spoofs a certificate to impersonate another website's origin (domain name) and poisons a DNS resolver to redirect users visiting a legitimate web application to his malicious web application. Note, both website impersonation and web app impersonation attack vectors are used for credential phishing. However, website impersonation is far more difficult to achieve as the attacker has to be able to poison a DNS resolver in addition to spoofing a certificate.

4.1.7. Attack instances

In this section, we specify the threat model and the attack contexts we consider, cover the most prominent attacks within the threat model, and in each case discuss corresponding mitigation strategies. Table 4.2 shows a summary of the contextualised attack vectors alongside effective mitigation strategies.

Threat Model. There are three broad types of entities in the web authentication and secret management ecosystem: web apps, the browser (limited to its native code, encompassing Berytus as a component), and browser extensions (including secret managers). We assume that the browser's native code (and its privileged execution context), including Berytus, is trusted. We also assume that secret managers are trusted entities, but other browser extensions could be malicious. Web apps could be legitimate or malicious. We do not assume an extra-vigilant user, so the user may visit and not distinguish malicious websites and may install malicious browser extensions. However, note that if the user consents to an extension being given the secret management permission, the extension is considered to be a secret

	Attack Vector	Attack Context			
		Legitimate Web App		Malicious Web App	
		Website Auth	Web App Auth	Website Auth	Web App Auth
JavaScript Code Injection	XSS	CSP/CT	CSP/CT/E2EE		
	TPInj	×	E2EE		
	ExtInj	CT	CT/E2EE		
Certificate Spoofing	AppImp				E2EE
Composite Vector	SiteImp			×	

× = cannot be mitigated; / = or; = inapplicable;
CSP = Content Security Policy; CT = Credential Tokenisation;
E2EE = App-level end-to-end encryption;

Table 4.2.: Attack mitigation strategy matrix for credential theft attacks. See Section 4.1.6 for a description of the attack vectors and Section 4.1.7 for the contexts.

manager and hence trusted. Moreover, we assume that the user's secret manager will only transfer credentials under the same credential mapping strategy of when it was saved; e.g., if the credential was registered under domain-based credential mapping, it will only be transferred when domain-based credential mapping is used. We assume this to be true for the two levels of secure credential mapping as well (see Section 4.1.1).

Attack Contexts. We define the following contexts, shown in Table 4.2, distinguishing between legitimate web applications and malicious web applications, and between website authentication and web application authentication.

- *Legitimate Web App / Attacker's Malicious Web App.* This factor influences the types of attack vectors that the attacker can leverage. The user either visits a legitimate web application or, somehow, ends up on the attacker's malicious web application. Some attack vectors are only applicable under a legitimate web application or under a malicious web application only. JavaScript Code Injection is used to execute malicious code on websites which the attacker does not control. Hence, JavaScript Code Injection attack vectors are irrelevant on web applications which the attacker controls. Similarly, in website impersonation and web app impersonation attack vectors, the attacker spoofs a certificate exclusively under her malicious web app, impersonating another website's origin or another web application's signing key, respectively.

- *Website authentication / Web application authentication.* An important factor that determines the possible mitigation strategies against considered attack vectors and the secret manager's app-to-credential mapping strategy. Website authentication is certificate-based authentication of the website's origin (domain); i.e. HTTPS. The Credential Management API, HTML Autofill and ByPass all require website authentication, and thus employ a domain-based app-to-credential mapping strategy. Web application authentication, a step following website authentication, is certificate-based authentication of the assumed web app key, e.g., Berytus's web app crypto actor authentication. Essentially, web app authentication is exclusive to Berytus as it is the only framework that considers web applications and app-specific keys as opposed to websites. Hence, exclusively under Berytus's web app authentication instances, key-based app-to-credential mapping strategy is employed. Moreover, Berytus's app-level end-to-end encryption can only be used as a mitigation strategy under web app authentication. Note, in the Berytus's experimental design, web app authentication is a choice left for web app developers. A web app is free to opt in for (Berytus) web app authentication in addition to the default (HTTPS) website authentication.

Credential and identity information theft attacks

In the attack instances below, the attacker's target is to steal the user's credential(s) or his identity information. For the sake of brevity, we will assume it is credential theft as the pathway is similar. Both attack modes, active or passive, can be used.

XSS: Cross-site scripting. In this attack, an adversary injects malicious JavaScript code (embodying the attack payload) into a benign web app's client-side execution context using cross-site scripting [15, 13, 28]. A tight Content Security Policy, which limits the origin from where web app resources can be loaded and whether inline scripts are permitted, is an effective mitigation strategy against XSS attacks. Credential Tokenisation, in which credentials are replaced with tokens in the front-end, has been proposed as a mitigation strategy against XSS-credential theft [15]. However, as discussed in Section 2.5, the previously proposed credential tokenisation technique is designed for HTML form submission [15] and is not compatible with JavaScript workflows. Berytus provides the possibility of setting up app-level end-to-end encryption which provides an orthogonal mitigation strategy against such attacks and is compatible with JavaScript workflows. In some contexts, Content Security Policy may be deemed too restrictive, or even infeasible to implement in contexts such as a single page application hosting forum, hence having an alternative mitigation strategy is useful.

TPInj: TLS Proxy code Injection. In this attack, a malicious TLS Proxy in the Middle tampers with the web app's delivered (HTML or JavaScript) code to include malicious code (attack payload) before it gets to the browser. Execution

of this attack payload cannot be mitigated using preventive measures, such as a restrictive Content Security Policy, specified in HTTP headers or body, as they can be tampered by the TLS proxy. App-level end-to-end encryption facilitated by Berytus can serve as a mitigation strategy here assuming a trusted registration of the web application key. While a TLS proxy can effectively bypass the protection provided by TLS encryption, initiating app-level end-to-end encryption will only be succeed if the web app is able to prove knowledge of a secret key (during authenticated key exchange) corresponding to a public key associated with the app under Berytus at the time of registration. Therefore, establishing app-level end-to-end encryption through Berytus effectively provides a separate independent secure medium for credential transfer between the web app and the secret manager which cannot be monitored by a TLS proxy.

ExtInj: Extension code Injection. In this attack, a malicious browser extension installed on the browser injects malicious JavaScript code as the attack payload into a benign web app's client-side execution context. Content Security Policy is not designed to prevent code injection by browser extensions. Both Credential Tokenisation and Berytus's facilitated app-level end-to-end encryption can render extension code injection ineffective in stealing users' credentials. However, to reiterate, Stock and Johns' [15] credential tokenisation approach is limited as it is tied to HTML form submission. Moreover, in practice, as we have mentioned in Section 2.6, previous work could not find any secret manager implementing it, possibly due to limitations imposed by browsers on extensions [28]. Conversely, Berytus supports secret manager extensions to establish app-level end-to-end encryption of users' credentials, making app-level end-to-end encryption the sole mitigation strategy effective against extension code injection that is both practical and realisable by secret manager extensions.

AppImp: Certificate Spoofing - Berytus App key Impersonation. In this Berytus-exclusive attack, a malicious web app spoofs a certificate to impersonate a legitimate web app's signing key used for key-based credential mapping. Although the attacker has managed to fraudulently pass certificate-based authentication of signing keys (secure credential mapping level 1, see Section 4.1.1), Berytus's app-level end-to-end encryption serves an alternative avenue to cryptographically authenticate signing keys following an authenticated key exchange (secure credential mapping level 2). Thus, a malicious web app cannot impersonate a legitimate web app's signing key by spoofing the certificate if app-level end-to-end encryption was leveraged.

SiteImp: Certificate Spoofing and DNS Poisoning - Website Impersonation. In this attack, the adversary spoofs a certificate to impersonate another website's domain (origin) used for domain-based credential mapping and poisons a DNS resolver to redirect users to his malicious web application. Moreover, since the

attacker controls the malicious web application, she is able to execute the attack payload to steal the user's credentials under the impersonated website. Unfortunately, credentials registered exclusively under website authentication (domain-based credential mapping) are thus at risk of phishing via site impersonation. The impersonating website can ask the secret manager for a credential associated with the impersonated domain name. Conversely, credentials registered under web app authentication and app-level end-to-end encryption (secure credential mapping level 2), as described in the case of AppImp, are not at risk of credential theft.

User authentication hijacking attacks

We now look at another attack target, user authentication hijacking. The attacker is not interested in stealing the reusable credentials, but rather in any digital material to impersonate the user in front of the web app, e.g., an authentication token. HTTP cookies are designed in a way that client-side scripts are not able to access them. Therefore, injected scripts (through JavaScript Code Injection attack vectors) are not capable of stealing HTTP cookies stored in the user's browser. However, injected scripts can capture the client-side authentication request payload and forward it to the attacker's server. The attacker's server with access to a valid authentication request payload can submit it to the legitimate web application's server and obtain the authentication cookie. This sophisticated (real-time and time-limited) attack works against any authentication scheme passing through the web app front-end, including Secure Remote Password [1] and digital signature-based authentication. To mitigate against this, we ought to conceal the valid authentication request payload and authentication token from the adversary. Clearly, preventing cross-site scripting attacks, e.g., using CSP, disables the adversary from retrieving the authentication payload or cookie. However, malicious browser extensions can freely inject code. Hence, a mitigation strategy concealing authoritative payloads from client-side adversaries is needed. End-to-end encryption on itself is insufficient, as the encrypted form of a credential contained within the authentication payload is sufficient to impersonate the user.

Credential Tokenisation can conceal authentication payloads from client-side adversaries. However, detokenisation should only occur on network requests destined to the legitimate web application endpoints. Apart from Stock and Johns' [15] credential tokenisation strategy, we describe a Berytus-specific credential tokenisation strategy where detokenisation occurs on a signed list of HTTP endpoints. During the key exchange, the web application could put forth a list of detokenisation endpoints in the session parameters, and since the session parameters message is signed by the web app's signing key, we can be assured that those endpoints are not owned by the attacker. This credential tokenisation strategy is superior to that of Stock and Johns. It is not limited to (URL-encoded) HTML form submission but it would also be compatible with JavaScript workflows, e.g., leveraging the Fetch API [43]. It enables web applications to explicitly communicate detokenisation endpoints to the secret manager on a per-session basis instead of relying on a fixed set of filters, e.g.,

Stock and Johns' [15] password parameter name. This Berytus-specific credential tokenisation strategy is sufficient to mitigate against client-side user authentication hijacking attacks and is listed as future work in the discussion chapter.

4.1.8. Security comparison

The unravelled attack instances convey the danger of the considered attack vectors and the effectiveness of the relevant mitigation strategies. Noticeably, Berytus has superior security against credential phishing attacks. Berytus's key-based credential mapping coupled with app-level end-to-end encryption puts an end to credential phishing attempts. Account credentials registered and authenticated under this combination are always safe from phishing, however, credentials registered and authenticated under domain-based credential mapping are not. Hence, we declare that HTML Autofill, the Credential Management API and ByPass are insufficient on themselves to combat credential phishing attempts while with Berytus, and its full fleet of security services in action, credential phishing attempts are eliminated. ByPass is not phishing-proof as it does not mitigate against TLS MitM Proxies capable of intercepting credentials. Moreover, Berytus protects against accidental credential leakage by using a secure credential mapping strategy while the Credential Management API, HTML Autofill or ByPass might leak credentials as a result of domain reuse or co-existence of separate web applications under the same domain.

In essence, Berytus's facilitated services highlight their importance is mitigating against JavaScript Code Injection credential theft attacks. However, app-level end-to-end encryption is not meant to be a replacement for CSP, it ensures confidentiality of the user's credentials. Hence, in instances where CSP is too restrictive or ineffective, e.g., against malicious browser extensions, app-level end-to-end encryption provides complete protection against credential theft. For user authentication hijacking attacks, we have identified a Berytus-specific tokenisation strategy as a countermeasure. However, injected scripts can always interfere in the user's browser, manipulate a web page and silently send server requests, e.g., to transfer funds from the victim's account to the attacker. This calls for future work to investigate this problem; i.e. can a web app assert that the operation is invoked by the user, or at least to what degree, and not by an adversary impersonating the user?

More important, for the full security fleet to be in service, there should be a manager that stores web application keys. We designated the secret manager for this role. The secret manager stores web application (signing) public keys alongside user's credentials, and the key, being unique, is used as an index when suggesting credentials. This is a solution that enables secure credential mapping and app-level end-to-end encryption that does not fully rely on trusted third-parties (certificate authorities) and mitigates against spoofed certificates as in the case of AppImp. Hence, we emphasise the potential of secret managers to maintain web application keys, playing this role in establishing authenticated key exchanges and app-level end-to-end encryption. On a final note, this section corroborates the usefulness of

streamlining such security services into browser-side frameworks for which compatible paradigms such as the Credential Management API can adopt this security model.

4.2. Functionality evaluation

In Section 2.2, we highlighted the significance of deployability factors in security-enhancing authentication schemes. In our proposal, we also have to consider the deployability aspect of Berytus. We evaluate the feasibility of deployment of Berytus based on the following categories.

Functional compatibility. In secret manager-assisted authentication frameworks, the user, the web application and the secret manager are all involved in the orchestration. Hence, it is essential to fulfil the requirements of each party. Analysing the functional compatibility between the web application and the secret manager is the primary step to assert the *possibility of integration* with Berytus.

Integration effort. Integrating with Berytus entails code tailoring. Therefore, examining the integration effort aids in evaluating the *cost of integration*. One Berytus-integrated secret manager is sufficient to provide secret management and authentication facilities to any Berytus-integrated web application. Thus, with the existence of millions of service-offering web applications, the integration burden mostly falls on web applications to proceed with it. As such, we are mostly interested in examining the web application integration effort with Berytus.

4.2.1. Functional compatibility

Berytus is designed for compatibility with generic web applications, i.e. web apps with varying functional requirements. While it is difficult to offer maximum compatibility with generic authentication systems, Berytus strives to achieve increased compatibility by offering a high degree of modularity and flexibility. This design pillar enables some notable capabilities for secret managers and web apps that are discussed below and in part have been identified following our review of previous frameworks (see Table 2.1). In each capability, we provide its functional label and its corresponding label from Table 2.1. A comparison between existing frameworks and Berytus with respect to these capabilities is presented in Table 4.3.

- **Support for Multi-Step and Multi-Factor Authentication (Supports stateful challenges).** Existing frameworks only support "one-shot" scenarios, where credentials should be transmitted in one go; i.e. one instruction. Berytus models provision of multiple authentication credentials as independent successive *challenges*, where challenges are computed in a multi-step fashion as in multiple programming instructions, e.g., in the case of Secure

Remote Password [1]. And through multiple completions of challenges, Berytus is able to support multi-factor authentication. Web applications implementing multi-step or multi-factor authentication flows can be integrated with Berytus. For other solutions, none offer such capability, but it can be added in a future version release of solutions backed by an API.

- **Developer Control over User Experience (Preserves web apps' Auth UI).** The autonomy of web applications entails that they maintain control over the user experience and user interface for its users. In the authentication context, control over user experience implies that, inter alia, the web application decides when and how the operation starts and unfolds, respectively. In HTML Autofill, the web application does not have control over when the credential transfer operation starts and how it is unfolded. However, in HTML Autofill, developers have control over the UI, and hence they only have partial control over the login UX. Both the Credential Management API and Berytus provide this capability. In Berytus, the web application initiates the operation and reacts freely in modifying the user interface during the operation. In contrast, ByPass eradicates the web app client-side, diminishing web applications' control over the rendered user interface during authentication.

- **Maintaining the Conventional Usage Pattern (Preserves users' mental model).** A typical user expects to visit a web page on his browser to log in to the corresponding web app. Berytus, HTML Autofill and the Credential Management API align with this mental model, whereas more radical frameworks like ByPass expect users to conduct all their account-related tasks with web apps through the secret manager's interface.

- **Support for Secret Manager Extensions (Integrates with web extensions).** Berytus is designed to work with third-party secret managers that are installed as extensions. This in turn supports choice for end users and encourages an open ecosystem. Similarly, secret manager extensions can perform HTML Autofill or invoke the ByPass server API. However, only the Credential Management API officially works with the native, built-in browser secret manager. To circumvent the lack of third-party secret manager support in the Credential Management API, secret manager extensions can redefine the Credential Management API to "hook" into it and process the credential storage/retrieval instructions on behalf of the built-in secret manager. It becomes problematic when multiple secret manager extensions *monkey patch* the Credential Management API; an official integration pathway with secret manager extensions is better than secret manager extensions freely hooking into the API. Berytus supports multiple secret manager usage along with an official integration pathway for secret manager extensions. Hence, Berytus ensures interoperability between web applications and third-party secret managers.

Functional Capability	HTML Autofill	Cred Mgmt API	ByPass	Berytus
Multi-Step & Multi-Factor Auth	✕	○	○	●
Developer Control over UX	◐	●	✕	●
Conventional Usage Pattern	●	●	✕	●
Secret Manager Extensions	●	○	●	●
Flexible Account Design	○	○	○	●
Custom Authentication	✕	○	○	●
Secret Manager Allowlisting	✕	○	○	●

✕: not possible, ○ : not currently supported,
◐ : partially supported, ● : supported

Table 4.3.: Comparison of provided functional capabilities.

- **Flexibility over Account Design (Supports credential customisation).** Berytus allows web apps to define their account structure as any combination of the supported fields. Furthermore, Berytus has built-in support for record versioning, which allows graceful handling and migration between multiple versions of account record structures, and categorisation, which allows the possibility of multiple role-based accounts for the same user on the same web app. Solutions other than Berytus have not advised, but can in a future version release, a functional pathway to handle account record migration throughout the development lifetime of an authentication system.

- **Support for Custom Authentication.** Berytus facilitates custom authentication challenges to be initiated following approval from the selected secret manager. For custom challenges, Berytus accepts a JSON Schema specifying the messaging data format to validate the challenge messages sent back and forth between the web app and secret manager. Therefore, generic web applications can implement unorthodox authentication schemes, including authentication schemes that are not well-known or privy between an acquainted web application and secret manager pair. Existing solutions do not provide such support.

- **Support for Secret Manager Allowlisting.** Web apps with strict security policies may wish to restrict the list of secret managers they trust with user credentials. Berytus provides the facility for web apps to specify a public key allowlist of approved secret managers. Such a facility is not provided by any of the existing frameworks.

Each functional capability influences the degree of interoperability and either

fulfils a web application functional requirement or not. For instance, if a web application employs Secure Remote Password [1] authentication, it would not be interoperable with solutions not supporting the multi-step authentication capability. Additionally, if a web application needs control overs the user experience, it would not opt in for solutions which impede them from it.

This section identified noteworthy functional capabilities, serving as a blueprint for secret manager-assisted authentication frameworks interested in functional compatibility with generic web applications. Berytus showed superior functional compatibility as it was designed in that spirit. These capabilities can be adopted in a sibling, browser-side API, architecture such as the Credential Management API. On a final note, the goal of satisfying the functional requirements of web applications was to streamline a solution that is not infeasible in practice. A solution such as Berytus has satisfied the generic functional requirements we have identified in this space. Therefore, in Berytus, integration is not only possible with orthodox web applications but also with slightly more generic web applications.

4.2.2. Integration effort

The functional compatibility evaluation section paved the way in answering the *possibility of integration* question. In this section, we look at the *cost of integration*, or simply, the integration effort. In essence, the integration effort varies based on the undertaken design decisions. We begin by examining the integration surface to determine which domains are affected by the Berytus integration. Following the identification of implicated domains, we briefly conduct a localised assessment of the implications of the Berytus security services on the affected domains. Finally, we compare the potential implications when integrating with Berytus, taking into account its different integration decisions, against the Credential Management API, ByPass and HTML Autofill.

Integration surface

We have identified the following architectural domains that might be affected by the integration:

Web app client-side. Since Berytus is a Web API, the web application front-end (client-side) is naturally part of the integration surface. Given an existing web app front-end, the creation of a new module is necessary; no logical changes to the existing modules are necessary. The new module should contain instructions to create the Berytus channel and start operations. Generally, the new module should be instantiated following a button click (e.g., “sign in with Berytus”), sending HTTP requests to the web app back-end when necessary (e.g., transmitting credentials to proceed with authentication).

Web app server-side. As a Web (browser) API, Berytus is not interested in the web application's back-end (server-side) implementation or configuration. A web application performing password-based or digital signature-based authentication does not have modify its back-end authentication logic to achieve a successful integration. However, some integration decisions, e.g., enabling app-level end-to-end encryption, might necessitate back-end architectural changes. Therefore, at a minimum, the server-side is detached from the integration surface unless some integration decisions are opted in for.

Public key infrastructure. The Berytus architectural specification suggests the use of digital certificates, when opted in for, to authenticate web applications. Therefore, similar to the web app server-side, public key infrastructure is not involved in the integration unless the web application opts in for web app authentication.

Implications on the integration surface

After covering the integration surface, we now assess the potential implications of the integration decisions on the identified domains. In each implication, shown as a bullet point, we provide a Berytus-specific label followed by a generic label to align it with implications of other secret manager-assisted authentication frameworks. There are three integration decisions a web application has the choice of implementing, described below.

1. Base. This is the fundamental decision for a web application to integrate with Berytus. At a minimum, the web app client-side has to incorporate JavaScript code to invoke the Berytus Web API.

2. Web app authentication. Opting in for web app authentication requires the possession of a Signing Key for which its public key counterpart is specified in the Berytus Signing Key Allowlist certificate extension (see web app actor authentication in Section 3.2.3). The mere possession of a Signing Key, however, does not necessarily have to be part of the Web App server-side. It does not have to be stored in a database; offline storage or a key management service is an option. Apart from key possession, the Certificate Authority (public key infrastructure) and the web app client-side are involved in web app authentication.

Web app client-side. The implications of web app authentication for the web app client-side are trivial; only the specification of the base64 string of the SPKI DER-encoded Signing Key (e.g., see ECC SPKI [4]) as part of the channel creation options is necessary, this could be hard coded.

Public key infrastructure. There are three implications of web app authentication for public key infrastructure.

- **Introduction of Signing Key Allowlist: Addition of a certificate extension.** To achieve Web App Authentication, specification of a Berytus Signing Key Allowlist in the web application's certificate is necessary. Fortunately, the X.509 v3 standard introduced certificate extensions to allow the addition of custom fields within the certificate. Therefore, it is possible to create certificates containing the Berytus Signing Key Allowlist as an extension without the need of an X.509 version update. The only requirement is to agree on and register an Object Identifier (OID) [41] for the Berytus certificate extension. To produce a working example, we assumed an arbitrary OID `1.2.3.4.22.11.23` for the Berytus signing key allowlist extension, as shown in Listing 4.1.

- **Creation of a CSR with the Berytus extension: Modification of CSR creation process.** The Certificate Signing Request [2] (CSR) is the primary mean to request certificates from certificate authorities. The CSR contains information on the certificate's subject as well as any certificate extensions. To create a certificate with the Berytus extension, the CSR must include the Berytus extension. Thus, system administrators should extend their CSR creation process to include the extension before sending it to the Certificate Authority.

- **Signing Key Proof of Possession: Addition of a C.A. PoP routine.** For each specified signing key in the Berytus Signing Key Allowlist certificate extension, the Certificate Authority (C.A.) should verify for its Proof of Possession (PoP). The CSR [2] is sufficient to assert the Proof of Possession of the certificate's corresponding key (subject key). However, the signing key, being distinct from the subject key, has to be checked for Proof of Possession as well. Hence, Signing Keys Proof of Possession validation is an additional step that Berytus-compatible certificate authorities have to undertake. The Certificate Authority is free to streamline the Signing Key Proof of Possession process, however, digital signatures are expected to be the core method of verification. For instance, in our proof of concept, to embed the Berytus extension into the CSR [2], each specified signing key must sign the certificate's subject key; see Appendix Listing A.3 for a minimal working example of CSR generation with the Berytus extension. Therefore, by requiring each specified signing key in the Berytus extension to sign the certificate's subject key, the Certificate Authority can assert the Signing Key Proof of Possession status. On a final note, it is unrealistic to expect all certificate authorities to perform the Berytus Signing Key PoP routine. Fortunately, at a minimum, one certificate authority implementing this routine is sufficient for certificates with the Berytus extension to begin rolling out.

```
X509v3 extensions:
    X509v3 Basic Constraints:
        CA:FALSE
    X509v3 Key Usage:
        Digital Signature, Non Repudiation...
    X509v3 Subject Alternative Name:
        DNS:example.tld, DNS:www.example.tld
    1.2.3.4.22.11.23:
        0.....key:MCowBQYDK2VwAyE...,sksig:upTIvgc...
    X509v3 Subject Key Identifier:
        AB:6E:E8:98:4C:FD:28:B4:7A:7E:13:D7:CB:F2:...
```

Listing 4.1: This listing shows the extensions defined in a created certificate using the shell script in listing A.3. The assumed Berytus Signing Key Allowlist OID [41] is `1.2.3.4.22.11.23`.

3. App-level end-to-end encryption. Web app authentication is a precursor for app-level end-to-end encryption. Therefore, the implications of web app authentication are also applicable when opting in for app-level end-to-end encryption. In app-level end-to-end encryption, both the web app client-side and the web app server-side are involved; encrypted information traverses the web app client-side context before arriving to the web app server-side.

Web app server-side. The implications of app-level end-to-end encryption for web app server-side are described below. Note, the cryptographic routines, e.g., key derivation or message signing, involved in app-level end-to-end encryption are offloaded to cryptographic libraries, hence the web app server-side only has to invoke them and not implement them.

- **Storage of signing and session keys: Provision of a new storage unit.** As part of the authenticated key exchange, the web application is responsible for storing the signing key, generated session keys and session parameters. Hence, a new storage unit is needed to maintain cryptographic material.
- **Invocation of authenticated key exchange routines: Addition of HTTP request handlers.** The authenticated key exchange process (see mediating an authenticated key exchange in Section 3.2.3) requires the web app server-side to generate and store a session key, sign the session parameters and compute the shared key. Consequently, from a high-level perspective, the web app server-side has to implement additional HTTP request handlers to invoke these functions. The web app client-side would transmit the cryptographic material to the server-side by dispatching HTTP requests.
- **Invocation of data encryption/decryption routines: Addition of HTTP request handlers.** Upon receiving ciphertext, the web app server-side should invoke ciphertext decryption routines where appropriate; e.g., in

the authentication module to decrypt the ciphertext credentials before proceeding with the authentication. Similarly, the web app server-side should encrypt credentials transmitted from the server-side to its client-side; e.g., when creating a Private Key field (see Table 3.1). We also group this implication under "Addition of HTTP request handlers" since these invocations can be placed in a request handler that transparently decrypts or encrypts data passing through it before reaching other request handlers; i.e. a request handler middleware.

Web app client-side. There are two implications of app-level end-to-end encryption for web app client-side:

- **Transmission of authenticated key exchange messages: Dispatch of new HTTP requests.** Throughout the authenticated key exchange process (see mediating an authenticated key exchange in Section 3.2.3), the client-side is used as the medium to pass key exchange material, i.e. *messages*, between the web application and the secret manager. As such, the web app client-side is responsible for transmitting the material received from the secret manager to a secure site, the server-side. This is done by dispatching new HTTP requests containing key exchange material from the client-side to the server-side.

- **Submission of ciphertext credentials: Dispatch of new HTTP requests.** Following a successful authenticated key exchange, the secret manager would provide credentials as ciphertext. Hence, the web app client-side, having previously defined an authentication request input format compatible with cleartext credentials, must now define a new request input format for ciphertext credentials. The new ciphertext request format is not necessarily divergent from the cleartext request format. Generally, it is expected that the ciphertext credential would have a different encoding than plaintext credentials, and that ciphertext HTTP request inputs might include metadata in the headers; e.g., a session-specific shared key identifier used as an index key to lookup the stored shared key for ciphertext decryption. Both request input formats can share the same authentication endpoint. When sharing the same endpoint, the ciphertext requests would be first handled by the web app server-side request handler responsible for the invocation of ciphertext decryption routines; thereafter, the request is passed to the authentication request handler which verifies cleartext credentials. Conversely, the cleartext requests would be exclusively handled by the web app server-side authentication request handler. This cleartext/ciphertext request endpoint sharing pattern could be applied not just to credentials but any information encrypted by the secret manager. Overall, the web app client-side dispatches two distinguishable sets of HTTP requests, one for plaintext payloads, and one for ciphertext payloads, possibly sharing the same endpoint, as a result of enabling app-level end-to-end encryption.

Incentives and effort amelioration

The Berytus integration effort can be regarded as an investment into the web application's security. Therefore, web app developers might find the Berytus security services appealing and proceed with the integration despite the necessary effort and architectural changes. More important, to alleviate the integration effort in the web app server-side, we propose a middleware model, discussed below, which might encourage web app developers further. This benefit is similar to that of providing a software library to support developers in adopting a proposed paradigm.

The middleware model is an excellent solution to achieve a complete integration (web app authentication and app-level end-to-end encryption), it does not require any changes to the authentication logic defined in the authentication module. Figure 4.2 illustrates the deployment of such a middleware. It handles ciphertext decryption before feeding the request body to the authentication module, and thus the authentication logic remains unchanged. In addition to data encryption/decryption, it can process authenticated key exchanges, relieving the web app developer from implementing the necessary request handlers for the key exchange. It is a flexible, backward-compatible solution to achieve app-level end-to-end encryption.

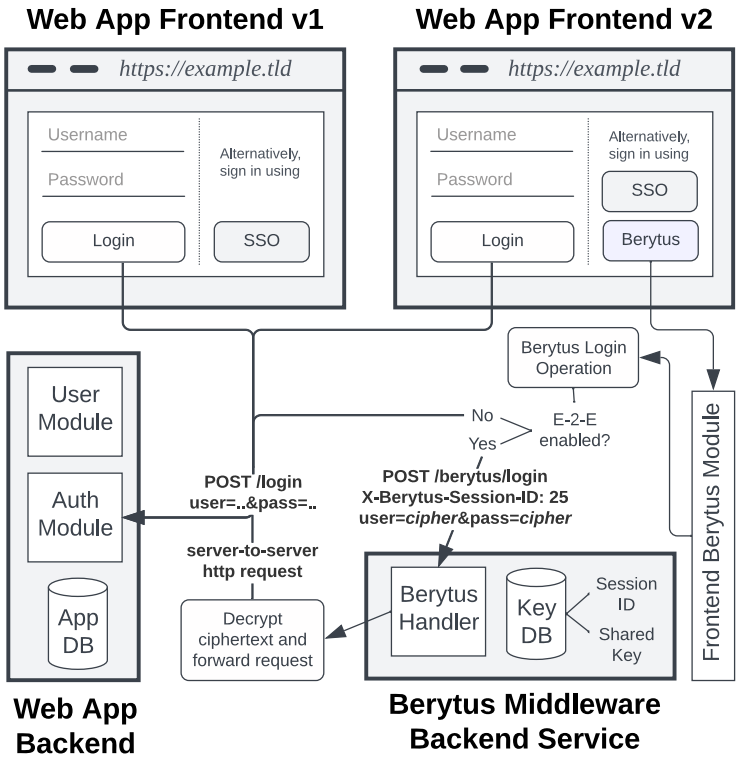

Figure 4.2.: Illustration of an external middleware service model to achieve end-to-end encryption without requiring web app backend changes.

Implication	Cred. Mgmt. API	HTML Autofill	ByPass	Berytus Base	Berytus App Auth	Berytus E2E	Berytus E2E-midlw.
Public key infrastructure:				●	●	●	
Addition of a certificate extension				●	●	●	
Modification of CSR creation process				●	●	●	
Addition of a C.A. PoP routine				●	●	●	
Web app server-side:			●		●		
Provision of a new storage unit					●		
Addition of HTTP request handlers			●		●		
Web app client-side:	●	●		●	●	●	●
Addition of HTML semantics		●					
Addition of JavaScript code	●			●	●	●	●
Dispatch of new HTTP requests						●	●

Table 4.4.: Comparison of implicated software domains when integrating with the Credential Management API, HTML Autofill, ByPass or Berytus (Base; Web App Authentication; End-to-end encryption; End-to-end encryption using a middleware).

Although the middleware was deployed as an external service in Figure 4.2, it can be implemented within the web application back-end as a standalone module that exposes request handlers for the key exchange and would modify incoming HTTP requests on the fly, decrypting ciphertext when appropriate, before the input arrives to the authentication module. There are advantages and disadvantages depending on where the middleware is deployed. When deployed as an external service, the network latency is increased and the authentication module would have incoming HTTP requests with a source IP address of the external service instead of the user's IP address. In a different architecture such as event-driven microservices, services are loosely coupled and instead of sending a server-to-server HTTP request, the authentication request is turned as an event containing the original user's IP address and pushed to an event stream which other services, such as an authentication service, can pick up. Ultimately, introducing the middleware as an external service, event-driven or not, offers increased flexibility; backward compatibility is guaranteed since the existing web application back-end code base remains unchanged.

In this section, we have dissected the implicated software domains when integrating with Berytus at various levels. Table 4.4 summarises them along with a high-level comparison between the existing solutions. Note, we have not considered the feasibility of a middleware model for ByPass since we could not find a

public repository for its API. Moreover, we have not evaluated the effort in terms of required developer hours; however, this examination serves as a starting point for future developer studies. Nonetheless, the use of a middleware to achieve web app authentication and app-level end-to-end encryption is promising, requiring only web app client-side changes, and might persuade web app developers to integrate with Berytus due to the common understanding that, in general, client-side changes could be easier to incorporate than server-side changes. More important, since only one certificate authority is sufficient to issue certificates with the Berytus extension, Berytus as a security solution is not infeasible and certainly possible.

5. Discussion and conclusions

In this chapter, we will validate Berytus per the evaluation method described in the methodology (Section 3.1), discuss the limitations, recommendations for future work and the amplified potential of secret managers.

5.1. Validation

In the security evaluation section, we dissected the prominent attack vectors on credential theft, referring to previous mitigation strategies such as Credential Tokenisation [15] and standardised ones such as the Content Security Policy. The analysis showed that Berytus is superior to other secret manager-assisted authentication frameworks in combating credential theft attacks. Berytus provides the two capabilities "Safe from XSS-password theft" and "Safe from TPitM-password theft" listed in Table 2.1. Furthermore, Berytus's accurate and authenticated web app-to-credential mapping combats credential phishing attacks by deceitful web applications through the use of web app authentication and app-level end-to-end encryption; this is listed as "Secure credential mapping" in Table 2.1. Lastly, Berytus's security policies are consistent irrespective of the secret manager implementation. Overall, the security evaluation shows that Berytus improves the security status quo in the secret manager-assisted authentication space.

In the functionality evaluation section, we focused on functional compatibility and integration effort, answering the important question, the feasibility of deployment by tackling two factors: functional compatibility and integration effort. Berytus satisfies the identified functional requirements, offering sufficient functional compatibility for orthodox and (slightly-) unorthodox web applications. Furthermore, we have examined the integration effort, showing how it is logically distributed among the implicated domains and highlighted the high-level steps required for a base or complete integration. The integration effort assessment indicated that at least one certificate authority is needed to buy into the Berytus certificate extension (web app authentication) for certified web app-specific keys to start rolling out. Moreover, the assessment demonstrated that by leveraging our middleware model (Figure 4.2), a complete integration bestowing valuable security benefits can be realised without web app server-side changes. Hence, we assert that Berytus is indeed feasible to deploy and, thus, a practical solution.

Our evaluation method (Section 3.1) emphasised on improving the status quo by fulfilling all of the capabilities listed in Table 2.1 and Berytus does indeed fulfil them. Therefore, the Berytus experiment is a success, passing the evaluation criteria. Overall, Berytus is a promising secret manager-assisted web user authentication

framework, it is feasible to deploy and secure against prominent, credential theft attacks. It addressed the existing gaps identified in Table 2.1, proposed an alternative, app-specific credential mapping strategy through secure web application—secret manager identification and streamlined an app-level end-to-end encryption flow to protect against web page-based and network-based eavesdropping adversaries.

As far as we know, Berytus is the first secret manager-assisted web user authentication framework offering browser-based unified secret management, authenticated secret manager—web application identification and app-level end-to-end encryption. Previous secret manager-assisted authentication frameworks did not consider coupling security services in their approaches, rather they depended on the existing web infrastructure security. ByPass is a great and significant proposal, its shift to server-side communication shows how security is improved due to web app client-side attack vectors being eliminated from the attack surface. Berytus sticks with the client-side context and introduces crucial security services to mitigate against client-side attack vectors which other browser-side frameworks such as HTML Autofill, PMF or the Credential Management API do not. Ultimately, Berytus is an experiment, providing beneficial security services and a unified secret management interface, both of which improves the status quo and can be adopted into sibling architectures such as the Credential Management API.

5.2. Limitations

We identified two limitations in our work, discussed below.

Signing key revocation. During the lifetime of the Berytus Signing Key, revocation might be necessary, e.g., if the cryptographic private key was compromised. We have not conceptualised a pathway to facilitate secure key revocation. In Berytus, the Signing Key is used as an application identifier. Hence, there should be two additional facilities, one to revoke the key, and one to migrate to the new key as the latest application identifier. Revoking a Signing Key and switching to another should be streamlined under scrutiny, otherwise a loose approach would cause the user's credentials to be unusable without manual intervention. Future work could propose an alternative Berytus extension and investigate whether standardised protocols such as the Online Certificate Status Protocol (OSCP) could be leveraged in determining the revocation status of Berytus keys.

Multiple browser usage. Berytus offers unified secret management for secret managers installed within the browser, not taking into consideration secret managers installed on other browsers or as standalone operating system applications. An improved, alternative design of Berytus supporting secret managers in different environments could be worth exploring.

5.3. Further work

Apart from the limitations, we suggest a few recommendations for future work:

Conducting usability and developer studies. To further investigate the integration effort, we suggest conducting developer studies examining their perception of Berytus and estimating the required number of developer hours for the three integration pathways of Berytus; (1) base, without the use of any security service; (2) use of web application authentication; (3) use of web application authentication, authenticated key exchange and app-level end-to-end encryption. Furthermore, a usability study could shed light on any shortcomings of Berytus and help us understand the users' perceptions and mental models.

Requiring full usage of Berytus's security services. While Berytus's experimental design offers three integration levels where the security increases gradually along with the integration effort, we now, however, recommend that the full fleet of security services be employed at all times. Assuming the middleware solution significantly decreases the web app integration effort, the web app integration effort of the first (base) pathway, i.e. without web app authentication or end-to-end encryption, would be similar to that of the third pathway, web app authentication and end-to-end encryption enabled. This assumption could be verified by comparing the required number of developer hours (following a developer study) of the first pathway with the third pathway, and if similar, we suggest requiring full usage of Berytus's security services instead of providing a pick-and-choose security paradigm.

Streamlining a Berytus-specific Credential Tokenisation strategy. During the early design phases of Berytus, we considered implementing credential tokenisation, however, we opted in for end-to-end encryption as it protects sensitive data from being eavesdropped past the client-side, e.g., network adversaries in the middle. In the security evaluation, user authentication hijacking attacks showed how adversaries do not necessarily need to acquire knowledge of credentials but any of its valid shapes to succeed in user authentication and impersonate the user. For web page-based (web app client-side) adversaries, e.g., using JavaScript Code Injection, credential tokenisation is an effective mitigation strategy as the valid shape of the credentials would not be accessible. However, detokenisation should occur in HTTP requests destined to endpoints (i.e. resources) owned by the web application. Hence, we suggest future work to streamline an appropriate Berytus-specific strategy where, e.g., the session parameters during the key exchange could be used to specify the valid HTTP endpoints where credentials should be detokenised.

5.4. Amplified potential of secret managers

Berytus enables secret managers to process account-related operations on behalf of users. The streamlined operations include tasks not present in the status quo such as the Secure Remote Password [1] protocol and one-time password email/phone verification[1]. Secret managers could evolve into processing passwordless, one-time password authentication by hosting an email server alongside its credentials vault. Fundamentally, secret managers become robust agents for users, managing their credentials, including email- or phone-based ones, and processing account-related operations on their behalf. For end users, Berytus bestows a Single-Sign-On-like experience but with increased privacy; a local (client-side) paradigm does not necessitate users to trust third-party or centralised systems with their personal information or credentials. We identified two innovative functionality of secret managers that could be implemented:

- **Secure client-side encryption (oracle).** Modern web applications are now leveraging the client-side context to execute cryptographic functions such as data encryption. Since the execution is on the client-side, the user is assured that the secret asymmetric or symmetric key does not leave her device and all the cryptographic functions are executed locally. However, there is no *visible or technical* guarantee. Security-aware users might examine the source code or analyse the network traffic to verify that their secrets are not transmitted anywhere else. This is similar to strong password protocols such as SRP, if an adversary "sidesteps" the protocol [18], he could steal the password (secret). Just as Berytus enables secret managers to facilitate a safe password entry interface, it could enable them to provide a secure programming interface for data encryption/decryption where the (a)symmetric secret key does not leave the secret manager. Providing a cryptographic, e.g., decryption, oracle would assure users that their secrets do not get exposed to the web application.

- **Generic resource credential storage.** Some web applications communicate cryptographic secrets such as SSH private keys or cryptocurrency wallet keys to end-users. Such secrets are not necessarily coupled with an account but given to the user. Normally, the secret material is downloaded or copied to the clipboard. Secret managers could offer a more organised and secure storage for such secrets. Users would not have to define an ad hoc organisational method (e.g., creating a file system tree structure) or manually download and move the secrets to an appropriate location. This is mostly beneficial for users (e.g., consultants or developers) managing a wide range of resource credentials.

[1] WebOTP [30], an extension of Credential Management API, is designed for assisted OTP verification, however, it does not provide the user agent (or secret manager) with sufficient data such as the phone number or email address of which the OTP was sent to.

Fundamentally, at a very high level, web applications become identifiable systems where account credentials (e.g., passwords), resource credentials (e.g., SSH keys) and personal information are shared. Providing such innovative functionality would radically transform the role and selling point of secret managers. Consequently, users would reconsider the overall benefit of using secret managers and could drive its adoption forward. In such an ideal world, the final *security* burden would be to develop secure secret manager implementations, e.g., secure credential storage, as the communication between the web application and the secret manager is secure under Berytus.

5.5. Concluding remarks

In this thesis, we presented Berytus, a novel governance framework that harmonises web application—secret manager interactions to orchestrate programmable account authentication and registration operations. This answers our research question in Section 1.2. It does not degrade usability, it has consistent, strong security measures and its design tackles practical challenges to ensure correct behaviour. We have demonstrated its feasibility, both theoretically and in practice, providing an Open Source implementation of Berytus in Mozilla Firefox and a Berytus-compatible secret manager, Secret*. We have fulfilled the research goals, we placed a mediator between web applications and secret managers, allowing both parties to communicate, and defined a mutual orchestration agreement.

The Berytus approach is not only novel, but also fills present gaps in other secret manager-assisted web user authentication frameworks to tailor a practical solution that fulfils significant functional and security requirements. None of the existing solutions attempted to couple security services such as web application authentication using app-specific keys or application-level end-to-end encryption. These security services are needed because of the insecure nature of the client-side context. Instead of focusing on preventative measures against malicious code injection (e.g., CSP), we streamlined the communication flow to be immune from credential theft attacks by injected code. Using application-level end-to-end encryption, credentials travel in encrypted form, ensuring its confidentiality from its departure from the secret manager to its arrival to the web application back-end. We have shown cases where malicious browser extensions can inject scripts regardless of the Content Security Policy, the primary mitigation strategy against cross-site scripting. Hence, this emphasises the value of application-level end-to-end encryption in achieving immunity against credential theft attacks, whether against cross-site scripting, malicious browser extensions or TLS Proxies in the Middle.

We have also considered the functional requirements of generic web applications, providing flexibility over account (credential) design, developer control over the user experience and support for multi-step & multi-factor authentication. Berytus does not force web applications to implement a new authentication scheme, rather it streamlines, as part of its API, prominent authentication schemes such

as password, secure remote password, digital signatures, and one-time passwords. The integration constitutes of connecting the secret manager as the input provider, instead of the user, to the web application's existing authentication module. Hence, the web application's authentication logic need not be changed, and can concurrently accommodate both users and non-users of Berytus. We have proposed a middleware model as a practical and flexible solution to actualise the full security fleet of Berytus. Consequently, the integration effort is significantly reduced and web application developers are only required to perform client-side changes only.

Berytus serves as a novel foundational paradigm for secure and practical secret manager-assisted web user authentication. We have identified two limitations which can be addressed in future work. We suggest future work to focus on streamlining an appropriate Berytus signing key revocation process and on exploring relevant pathways to support secret managers in different computing environments. Moreover, we encourage researchers to conduct developer studies and usability studies to quantify the integration effort on the developer side and to explore users' perceptions of Berytus, respectively. The security analysis covering prominent attack targets and vectors emphasised the danger of user authentication or operation hijacking attacks apart from credential theft attacks. We identified a mitigation strategy against user authentication hijacking using a Berytus-specific credential tokenisation strategy and recommend future work to investigate it further along with exploring relevant security measures to achieve immunity from operation hijacking.

Berytus bridged the gap between vision and practice to reveal the unexplored potential of secret managers. Secret managers could evolve to become robust agents processing account-related operations and managing account and resource credentials. Users would be equipped with competent agents, helping them manage their digital security whilst bestowing convenience. In conclusion, the experimental secret manager governance framework, Berytus, materialised the potential of evolved secret managers by facilitating a secure and practical foundation for secret manager—web application communication to undertake account-related operations, and answered the research question on the possibility of achieving harmony between the two parties without degrading usability or security. This is a significant improvement in the secret manager-assisted web user authentication space, positively impacting users' and web applications' security, and Berytus's design decisions and security services can be incorporated into compatible sibling architectures such as the Credential Management API.

References

[1] T. D. Wu, "The Secure Remote Password Protocol," in *Network and Distributed System Security Symposium*, 1998.

[2] M. Nystrom and B. Kaliski, *PKCS #10: Certification Request Syntax Specification Version 1.7*, Nov. 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2986#section-3.

[3] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *Public Key Cryptography - PKC 2006*, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228, ISBN: 978-3-540-33852-9.

[4] S. Turner, D. Brown, K. Yiu, R. Housley, and T. Polk, *Elliptic Curve Cryptography Subject Public Key Information*, Jan. 2009. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5480 (visited on 2024-01-08).

[5] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *IEEE Symp. Security and Privacy*, Edition: IEEE Symp. Security and Privacy, IEEE, May 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-quest-to-replace-passwords-a-framework-for-comparative-evaluation-of-web-authentication-schemes/.

[6] M. Blanchou and P. Youn, *Password Managers: Exposing Passwords Everywhere*, 2013. [Online]. Available: https://raw.githubusercontent.com/iSECPartners/publications/master/whitepapers/password_managers.pdf.

[7] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, You, Get Off of My Clipboard," in *Financial Cryptography and Data Security*, A.-R. Sadeghi, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–161, ISBN: 978-3-642-39884-1.

[8] R. Gonzalez, E. Y. Chen, and C. Jackson, "Automated password extraction attack on modern password managers," *arXiv preprint arXiv:1309.1416*, 2013. [Online]. Available: https://doi.org/10.48550/arXiv.1309.1416.

[9] D. J. Bernstein, *[Cfrg] 25519 naming*, Aug. 2014. [Online]. Available: https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU/ (visited on 2024-02-07).

[10] Z. Li, W. He, D. Akhawe, and D. Song, "The Emperor's New Password Manager: Security Analysis of Web-Based Password Managers," in *23rd USENIX Security Symposium (USENIX Security 14)*, ser. SEC'14, event-place: San Diego, CA, USA: USENIX Association, 2014, pp. 465–479, ISBN: 978-1-931971-15-7. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/li_zhiwei.

[11] M. O'Neill, S. Ruoti, K. E. Seamons, and D. Zappala, "TLS Proxies: Friend or Foe?" *CoRR*, vol. abs/1407.7146, 2014, arXiv: 1407.7146. [Online]. Available: http://arxiv.org/abs/1407.7146.

[12] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, *OpenID Connect Core 1.0 incorporating errata set 1*, Aug. 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html#StandardClaims (visited on 2024-01-16).

[13] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson, "Password Managers: Attacks and Defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 449–464, ISBN: 978-1-931971-15-7. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/silver.

[14] F. Stajano, M. Spencer, and G. Jenkinson, "Password-Manager Friendly (PMF): Semantic Annotations to Improve the Effectiveness of Password Managers," in *Lecture Notes in Computer Science*, vol. 9393, Dec. 2014, p. 61, ISBN: 978-3-319-24191-3. DOI: 10.1007/978-3-319-24192-0_4.

[15] B. Stock and M. Johns, "Protecting users against XSS-based password manager abuse," ACM, 2014. DOI: 10.1145/2590296.2590336. [Online]. Available: https://dx.doi.org/10.1145/2590296.2590336.

[16] E. Stobert and R. Biddle, "Expert Password Management," in *Passwords*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:40395405.

[17] N. Alkaldi and K. Renaud, "Why Do People Adopt, or Reject, Smartphone Password Managers?" In *The 1st European Workshop on Usable Security*, EuroUSEC 2016, Jan. 2016. DOI: 10.14722/eurousec.2016.23011.

[18] S. Ruoti, J. Andersen, and K. Seamons, "Strengthening Password-based Authentication," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/soups2016/workshop-program/way2016/presentation/ruoti_password.

[19] S. Aurigemma, T. Mattson, and L. Leonard, "So Much Promise, So Little Use: What is Stopping Home End-Users from Using Password Manager Applications?" In *Proceedings of the 50th Hawaii International Conference on System Sciences*, Hawaii International Conference on System Sciences, Jan. 2017. DOI: 10.24251/HICSS.2017.490.

[20] S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, Issue: 8032 Num Pages: 60 Series: Request for Comments Published: RFC 8032, Jan. 2017. DOI: 10.17487/RFC8032. [Online]. Available: https://www.rfc-editor.org/info/rfc8032.

[21] B. Pfretzschner and L. ben Othmane, "Identification of Dependency-based Attacks on Node.js," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17, event-place: Reggio Calabria, Italy, New York, NY, USA: Association for Computing Machinery, 2017, ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3120928. [Online]. Available: https://doi.org/10.1145/3098954.3120928.

[22] D. Balfanz *et al.*, *Web Authentication: An API for accessing Public Key Credentials Level 1*, Jan. 2019. [Online]. Available: https://www.w3.org/TR/2019/PR-webauthn-20190117/ (visited on 2024-01-27).

[23] *Monkey Patching: An Analysis of Code Poisoning JavaScript.* Oct. 2019. [Online]. Available: https://jscrambler.com/blog/an-analysis-of-code-poisoning-monkey-patching-javascript.

[24] S. Pearman, S. A. Zhang, L. Bauer, N. Christin, and L. F. Cranor, "Why people (don't) use password managers effectively," in *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 319–338, ISBN: 978-1-939133-05-2. [Online]. Available: https://www.usenix.org/conference/soups2019/presentation/pearman.

[25] S. Seiler-Hwang, P. Arias-Cabarcos, A. Marín, F. Almenares, D. Díaz-Sánchez, and C. Becker, ""I Don't See Why I Would Ever Want to Use It": Analyzing the Usability of Popular Smartphone Password Managers," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, event-place: London, United Kingdom, New York, NY, USA: Association for Computing Machinery, 2019, pp. 1937–1953, ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354192. [Online]. Available: https://doi.org/10.1145/3319535.3354192.

[26] M. West, *Credential Management Level 1*, Jan. 2019. [Online]. Available: https://www.w3.org/TR/2019/WD-credential-management-1-20190117/ (visited on 2024-01-04).

[27] M. Carr and S. F. Shahandashti, "Revisiting Security Vulnerabilities in Commercial Password Managers," in *ICT Systems Security and Privacy Protection*, M. Hölbl, K. Rannenberg, and T. Welzer, Eds., Cham: Springer International Publishing, 2020, pp. 265–279, ISBN: 978-3-030-58201-2.

[28] S. Oesch and S. Ruoti, "That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 2165–2182, ISBN: 978-1-939133-17-5.

[Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/oesch.

[29] E. Stobert, T. Safaie, H. Molyneaux, M. Mannan, and A. Youssef, "ByPass: Reconsidering the Usability of Password Managers," in *Security and Privacy in Communication Networks*, N. Park, K. Sun, S. Foresti, K. Butler, and N. Saxena, Eds., Cham: Springer International Publishing, 2020, pp. 446–466, ISBN: 978-3-030-63086-7.

[30] S. Goto, *WebOTP API*, Apr. 2021. [Online]. Available: https://wicg.github.io/web-otp/ (visited on 2024-03-31).

[31] N. Huaman, S. Amft, M. Oltrogge, Y. Acar, and S. Fahl, "They Would do Better if They Worked Together: The Case of Interaction Problems Between Password Managers and Websites," IEEE, 2021. DOI: 10.1109/sp40001.2021.00094. [Online]. Available: https://dx.doi.org/10.1109/sp40001.2021.00094.

[32] S. Oesch, A. Gautam, and S. Ruoti, "The Emperor's New Autofill Framework:A Security Analysis of Autofill on IOS and Android," in *Annual Computer Security Applications Conference*, ser. ACSAC '21, event-place: Virtual Event, USA, New York, NY, USA: Association for Computing Machinery, 2021, pp. 996–1010, ISBN: 978-1-4503-8579-4. DOI: 10.1145/3485832.3485884. [Online]. Available: https://doi.org/10.1145/3485832.3485884.

[33] H. Ray, F. Wolf, R. Kuber, and A. J. Aviv, "Why Older Adults (Don't) Use Password Managers," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 73–90, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ray.

[34] S. Oesch, S. Ruoti, J. Simmons, and A. Gautam, ""It Basically Started Using Me:" An Observational Study of Password Manager Usage," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22, event-place: New Orleans, LA, USA, New York, NY, USA: Association for Computing Machinery, 2022, ISBN: 978-1-4503-9157-3. DOI: 10.1145/3491102.3517534. [Online]. Available: https://doi.org/10.1145/3491102.3517534.

[35] *SRP: What Is It?* Nov. 2022. [Online]. Available: http://srp.stanford.edu/whatisit.html.

[36] A. Cherry, K. Barmpis, and S. F. Shahandashti, "The Emperor is Now Clothed: A Secure Governance Framework for Web User Authentication through Password Managers," in *Information and Communications Security: 26th International Conference, ICICS 2024, Mytilene, Lesvos, Greece, August 26–28, 2024, Proceedings*, 2024.

[37] *Fetch directive*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/fetch_directive (visited on 2024-01-20).

[38] *HTML Standard: Autofill*. [Online]. Available: https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#autofill (visited on 2024-01-24).

[39] *HTML Standard: Custom elements*. [Online]. Available: https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements (visited on 2024-02-16).

[40] *HTML Standard: Form submission*. [Online]. Available: https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#form-submission-2 (visited on 2024-01-04).

[41] *OID repository*. [Online]. Available: http://oid-info.com/.

[42] *Terms: Third-Party Passkey Provider*. [Online]. Available: https://passkeys.dev/docs/reference/terms/#third-party-passkey-provider.

[43] *Using the Fetch API - Web APIs — MDN*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch (visited on 2024-01-04).

A. Code samples

```
const fields = await operation.addFields(
    new BerytusIdentityField("username", {
        private: false,
        humanReadable: true,
        maxLength: 24,
    }),
    new BerytusForeignIdentityField("email", {
        private: true,
        kind: "EmailAddress",
    }),
    new BerytusForeignIdentityField("phone", {
        private: false,
        kind: "PhoneNumber",
    }),
    new BerytusPasswordField("password", {
        passwordRules: "minlength: 6;", /*! Apple's password rules format */
    }),
    new BerytusSecurePasswordField("srp", {
        identityFieldId: "username",
    }),
    new BerytusKeyField("key", {
        alg: -42 /*! COSE Algorithm ID: "RSAES-OAEP w/ SHA-256" */,
    }),
    new BerytusSharedKeyField("sharedKey", { alg: -42 }),
    /*! Field registration with web app-produced field values: */
    new BerytusIdentityField(
        "accountId",
        { private: false, humanReadable: false, maxLength: 26 },
        "123456",
    ),
    new BerytusForeignIdentityField(
        "phone2",
        { private: false, kind: "PhoneNumber" },
        "+123456789",
    ),
    new BerytusPasswordField("pass2", {}, "helloPass"),
    new BerytusSharedKeyField(
        "sharedKey2",
        { alg: -42 },
        new BerytusSharedKeyFieldValue(new Uint8Array([1, 2, 3])),
    ),
);
```

Listing A.1: Example of account field creation and registration.

```
/**! @var usernameExists - from the webapp codebase; e.g., for a demo: */
const usernameExists = (username) => confirm(`${username} exists?`);

await (async () => {
    while (usernameExists(operation.fields.get("username").value)) {
        /*! The provided username is registered, reject it and ask
        for a new revision. Once rejectAndReviseFields() resolves,
    `   `operation.fields.get("username").value` reflects the new
        field value */
        await operation.rejectAndReviseFields({
            field: "username" /* field id or field object */,
            reason: "Identity:IdentityAlreadyExists",
            /*! Specify `newValue` to propose a revised value,
            otherwise the secret manager will produce one */
            //newValue: "usernameThatDoesNotExists"
        });
    }
})();
/*! Here, `operation.fields.get("username").value` is a valid username */
```

Listing A.2: Example showing an exhaustive validation of the username field value. Note, it is a good software engineering practice to implement a bailout case in exhaustive loops, e.g., using a maximum number of attempts limiter.

```
#!/bin/sh
set -e
# create configuration file, specifying
# the berytus extension
cat <<'EOF' > openssl.cnf
[req]
req_extensions = v3_exts

[v3_exts]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names
1.2.3.4.22.11.23=ASN1:SEQUENCE:berytus_extension_format

[alt_names]
DNS.1 = example.tld
DNS.2 = www.example.tld

[berytus_extension_format]
allowlist=UTF8STRING:$ENV::ALLOWLIST
EOF

# generate a berytus signing key
openssl genpkey \
    -algorithm ed25519 \
    -out example.berytus.privkey.pem
openssl pkey \
    -in example.berytus.privkey.pem \
```

```
    -pubout -out example.berytus.pubkey.pem

# generate the certificate subject key
openssl genrsa -out example.tls.privkey.pem 4096
openssl rsa -in example.tls.privkey.pem -pubout \
    -out example.tls.pubkey.pem
openssl pkey -pubin -inform pem \
    -in example.tls.pubkey.pem \
    -pubout -outform der \
    > example.tls.pubkey.der

list=""
for pubkey in *.berytus.pubkey.pem
do
    privkey="$(basename "$pubkey" .pubkey.pem).privkey.pem"
    spki_b64="$(openssl pkey -pubin -inform pem -in "$pubkey" \
                -pubout -outform der | base64)"
    # Signing the subject key DER digest using
    # the berytus key
    subject_sig="$(openssl pkeyutl -sign -rawin -inkey "$privkey" \
                   -in example.tls.pubkey.der | base64)"
    entry="key:${spki_b64},sksig:${subject_sig}"
    if [ -z "$list" ]; then
        list="$entry"
        continue
    fi
    list="$list,$entry"
done

# create the certificate signing request
ALLOWLIST="$list" openssl req \
    -new -key example.tls.privkey.pem \
    -config openssl.cnf \
    -subj '/C=UK/CN=Example/' \
    -out  example.tls.csr

# This where the CA would validate the CSR,
# checking each specified berytus key and its
# subject key signature.

# create the self-signed certificate
openssl x509 -req -in ./example.tls.csr \
    -out example.tls.crt \
    -key example.tls.privkey.pem \
    -sha256 -days 365 \
    -copy_extensions=copyall
```

Listing A.3: A working example of creating an Ed25519 key and including it in the X.509 Berytus Signing Key Allowlist certificate extension.

B. Sequence diagrams

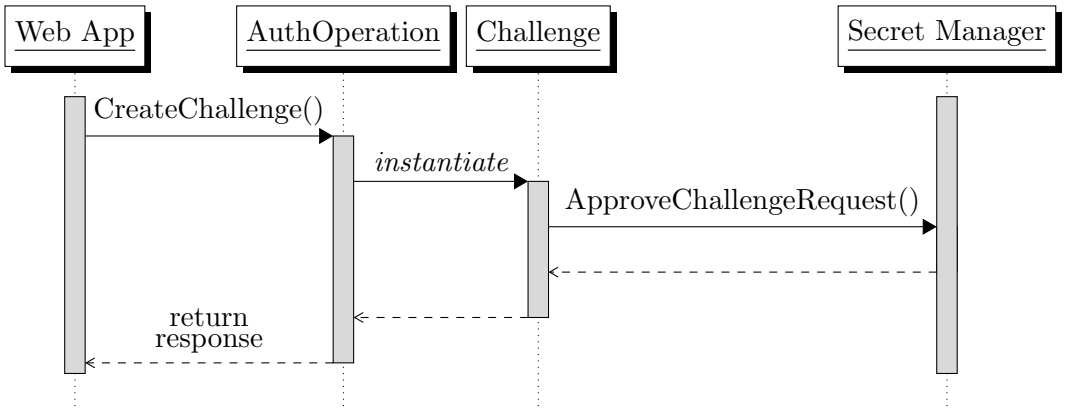

Figure B.1.: (Simplified) sequence diagram of the Berytus account authentication operation's challenge initiation process.

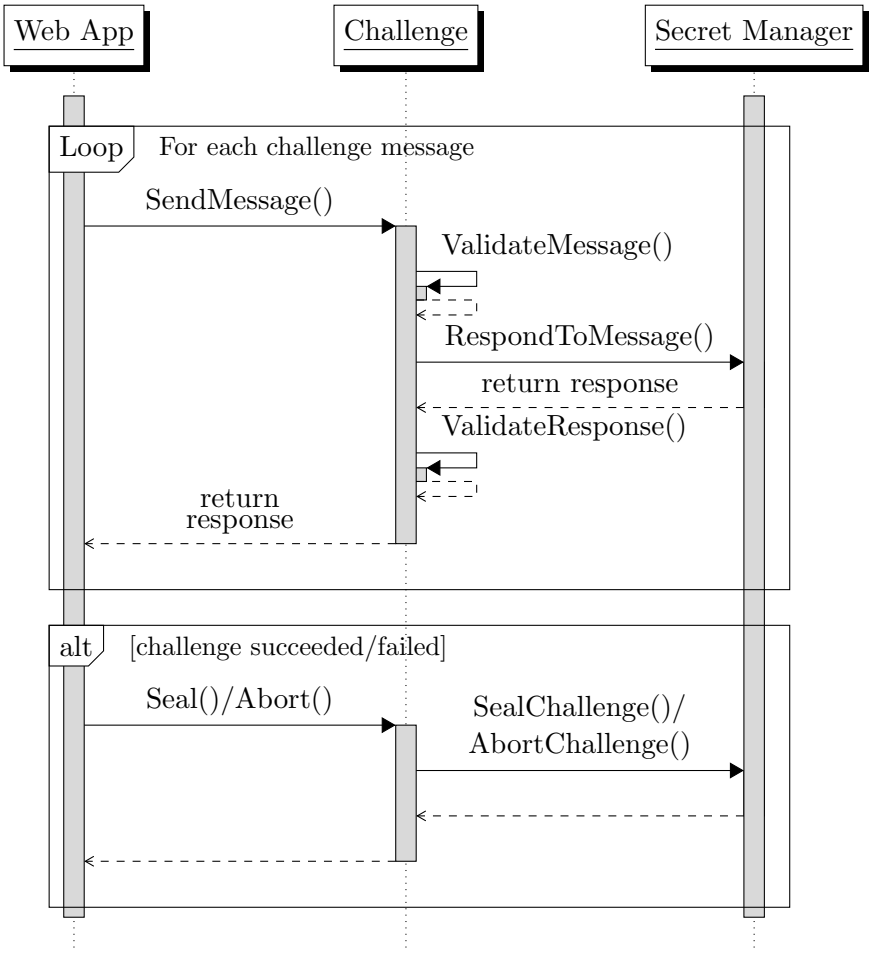

Figure B.2.: (Simplified) sequence diagram of the Berytus account authentication operation's challenge messaging pattern.