The Emperor is Now Clothed: A Secure Governance Framework for Web User Authentication through Password Managers*

Ali Cherry, Konstantinos Barmpis, and Siamak F. Shahandashti

University of York, York, United Kingdom
{ali.cherry,konstantinos.barmpis,siamak.shahandashti}@york.ac.uk

Abstract. Existing approaches to facilitate the interaction between password managers and web applications fall short of providing adequate functionality and mitigation strategies against prominent attacks. HTML Autofill is not sufficiently expressive, Credential Management API does not support browser extension password managers, and other proposed solutions do not conform to established user mental models. In this paper, we propose Berytus, a browser-based governance framework that mediates the interaction between password managers and web applications. Two APIs are designed to support Berytus acting as an orchestrator between password managers and web applications. An implementation of the framework in Firefox is developed that fully supports registration and authentication processes. As an orchestrator, Berytus is able to authenticate web applications and facilitate authenticated key exchange between web applications and password managers, which as we show, can provide effective mitigation strategies against phishing, cross-site scripting, inline code injection (e.g., by a malicious browser extension), and TLS proxy in the middle attacks, whereas existing mitigation strategies such as Content Security Policy and credential tokenisation are only partially effective. The framework design also provides desirable functional properties such as support for multi-step, multi-factor, and custom authentication schemes. We provide a comprehensive security and functionality evaluation and discuss possible future directions.

Keywords: Password Manager · HTML Autofill · User Authentication

1 Introduction

A typical web user is required to maintain many passwords for their online accounts, a task that requires unreasonable cognitive burden. Password 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 streamlining the authentication process, respectively.

* This is the ePrint of a paper to appear at the proceedings of ICICS 2024.
This work is licensed under CC BY-NC-SA 4.0.

HTML Autofill [25] is the most widely-used framework for password managers to assist browser users during authentication. Here, the password manager interprets the web page, based on HTML elements and attributes, to determine which input fields can be automatically populated on behalf of the user, and if so, what type of information is expected; e.g., a password or a credit card number. An (in-browser) password manager can then offer appropriate stored user secrets accordingly to automatically populate the fields. Unfortunately, HTML Autofill is prone to behaving incorrectly since it is essentially a heuristic approach based on educated guesses to interpret the front-end markup language [19]. To make matters worse, with the growing prevalence of *single-page applications* (SAPs), HTML Autofill is proving ineffective in correctly populating credentials. This is because SAPs often leverage the JavaScript Fetch API [10] instead of HTML form submission [26], and username and password fields are no longer necessarily coupled under a form element, forcing password managers to implement additional best-effort heuristics. Such issues highlight a need for specialised solutions to provide programmatic management of passwords and other user secrets.

W3C's Credential Management API [17] is an existing solution in this regard. The API provides a simple mechanism for web applications to store and retrieve user credentials in browser storage. It enables programmatic access to user credentials. However, Credential Management API is only available to so-called *native* user agents, i.e., in-built browser password managers, and cannot be used to store and retrieve credentials into and from password manager *extensions*.

In 2020, Stobert et al. proposed a remodelled password manager, ByPass, that communicates directly with the web application's back-end through a bespoke API [20]. Similar to Credential Management API, ByPass also eliminates issues caused by misinterpretation by introducing a programming interface. Furthermore, ByPass is not susceptible to credential theft by front-end threats such as cross-site scripting, and is able to provide enhanced services such as account deletion and password renewal. Despite all its benefits, ByPass radically transforms the user experience. The user is expected to launch the password manager and select a website to initiate the login process, instead of navigating to the website through the browser as is conventional. Consequently, Bypass not only takes away the control that web app developers relish today over the login user experience, it also requires users to develop and employ a new mental model.

Password managers are tasked with handling user credentials that are inherently sensitive. Hence, any password management governing framework must be evaluated based not only on its functionality features but also on whether it provides security services that help protect against credential theft attacks. Two prominent categories of such attacks are code injection and man-in-the-middle (MitM) attacks. Credentials can be stolen if client-side scripts, e.g., JavaScript code, is successfully injected on the client side by malicious entities external to the browser or by browser extensions. The former is the well-known *cross-site scripting* (XSS) attack, and we denote the latter by the term *extension code-injection* (ECI). On the other hand, while general MitM attacks are instigated by external network entities, a more subtle version may occur as a result of

faulty or compromised TLS proxies [14]. We call this a *TLS-proxy-in-the-middle* (TPitM) attack. While *Content Security Policy* (CSP) [24] can mitigate against XSS attacks, it is not effective against ECI attacks, and although TLS can defeat general MitM attacks, it does not provide any protection against TPitM attacks. HTML Autofill or Credential Management API do not provide any security services that can help mitigate against ECI or TPitM attacks. ByPass's architecture makes it intrinsically secure against XSS and ECI attacks as the client-side is "bypassed", however there is no mitigation against TPitM attacks.

Our Contributions. In this paper, we propose *Berytus*, a web governance framework that mediates between web applications and password managers to orchestrate programmable registration and authentication sessions. Crucially, Berytus is positioned between the web application front-end and the password manager client, operating natively in the browser. This architectural choice enables Berytus to provide programmatic password management services to both native and extension password managers while preserving developer user experience control and user mental models. We design two APIs for web applications and password managers to communicate with Berytus, respectively. Vital security services such as web application authentication and authenticated key exchange between web applications and password managers are built into the Berytus APIs, which allow application-level *end-to-end encryption* (E2EE) to be set up between the password manager and the web application back-end. This provides an effective security mitigation mechanism against XSS, ECI, and TPitM attacks.

Apart from the main functional and security services discussed above, Berytus's design offers extra benefits. By authenticating web applications, Berytus is able to facilitate and enable password managers to rely on more accurate *web application based credential mapping*, which aligns with the distributed nature of web applications and avoids the issues with domain-based mapping [5,12,7,3]. Furthermore, Berytus resolves the race condition issues when *multiple password managers* are in use by harmonising password manager registration and selection prompts. We have implemented Berytus in Mozilla Firefox. All project artefacts, including the code, are available at https://github.com/alichry/berytus.

The remainder of the paper is structured as follows: We cover the related work in Section 2. The framework architecture is discussed in Section 3. Security and functionality evaluations are given in Sections 4 and 5, respectively, conclusions in Section 6, and further information on our implementation in Appendix A.

2 Background and Related Work

We give an overview of frameworks governing password managers and their security and functionality properties. Since password managers store various types of credentials besides passwords, we use the term *secret manager* henceforth.

2.1 Existing Frameworks

We give further details on the inner workings of HTML Autofill, Credential Management API, and ByPass. Figure 1 provides a high-level comparison of the architectures of these major frameworks (as well as a preview of that of Berytus).

HTML Autofill: A UI-based Heuristic. Here, HTML input fields are filled by the browser (technically, the "user agent") with relevant data, e.g., personal information and secrets, on behalf of the user. The filled data is either generated on the fly (e.g. a proposed password) or retrieved from what has been captured and stored during an earlier browsing session. The HTML Standard outlines the autofill guidelines for user agents [26], however, secret manager extensions are at liberty to provide the autofill functionality. Web apps can support the filling process by integrating the HTML `autocomplete` attribute into relevant input fields. This aspect of the HTML Standard might not be implemented for some login forms, leaving secret managers to rely on ad hoc heuristics for input field classification to determine which input fields to fill. This leads to interaction issues between web apps and secret managers causing inconvenience for users, e.g., the absence of input *hints* hinders the filling process [7]. Therefore, while Autofill is highly *deployable* [4], it is prone to behaving erratically due to imperfect and varying heuristics. Besides, Autofill is *forceful*: web apps cannot officially disable its behaviour. The HTML Standard concurrently specifies a method for web apps to disable Autofill (by setting `autocomplete` to `off`) and a suggestion for user agents to ignore such a declaration at their discretion [26].

Password-Manager Friendly: An Autofill Extension. Motivated by the lack of required declarative hints for input field classification in HTML Autofill, Stajano et al. proposed Password-Manager Friendly (PMF), an additional set of HTML semantic labels to ensure correct secret management behaviour [19]. Unlike the HTML Autofill, PMF aids secret managers in detecting submission errors and different form types, including login, registration, password reset, and password change. If web apps incorporate those additional semantics into their forms, secret managers would no longer need to rely heavily on heuristics and this would lead to the reduction of interaction issues in HTML Autofill.

Credential Management API: A Credential Storage API. As Autofill was designed for user agents to aid users in *HTML forms*, it became difficult to detect sign-in ceremonies leveraging the Fetch API [17,10]. When JavaScript Fetch API is used, credential submission over HTTP is not necessarily tied to an HTML form, making it troublesome for Autofill heuristics to detect username and password fields since they are now separated. As a result, secret managers may fail to fill and save passwords. Besides, user agents lacked support for federated sign-ins in HTML Autofill, and password change could be further supported by requiring web apps to notify user agents when credentials have been changed. Credential Management API was proposed to ensure improved credential management and to support users with federated sign-ins [17]. Fundamentally, it offers a programming interface for web apps to store and retrieve credentials into and from

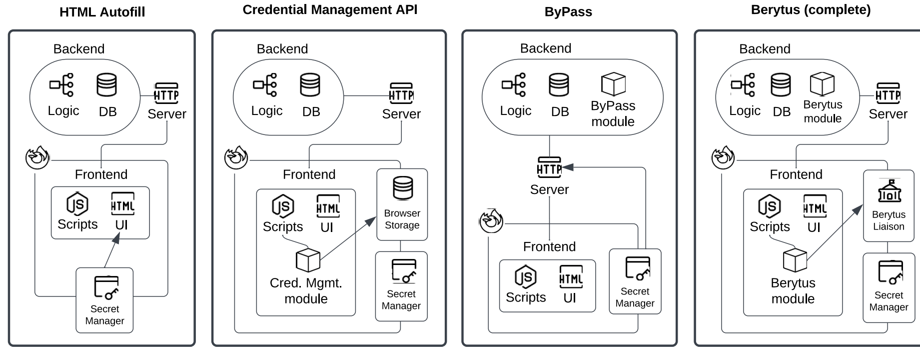

Fig. 1. Comparing HTML Autofill, Cred. Mgmt API, ByPass, and Berytus architecture

the user agent. Using JavaScript, a web app can insert a `PasswordCredential`, consisting of a username and a password, into the browser storage. When the user visits the web app in the future, the web app can programmatically retrieve the password credential from the browser storage. *Web Authentication* (WebAuthn) [1] is an extension to Credential Management API that enables digital signature based authentication by introducing the new `PublicKeyCredential`. Currently, some secret manager extensions act as third-party public-key credential providers by intercepting the WebAuthn API calls [23].

ByPass: A Secret Manager – Website Back-end Interface. Motivated to address the usability issues of secret managers, Stobert et al. proposed a re-imagined secret management model where their manager, ByPass, communicates directly with the web app's back-end [20]. As ByPass directly interacts with the web app's back-end, front-end security vulnerabilities such as XSS are eliminated from the attack surface. To log into a website, the user needs to open ByPass and search for the website they wish to visit within ByPass's UI, instead of navigating to the website through the browser. ByPass then handles the account registration or authentication process under its own UI and not the web app's, and only if the process succeeds, ByPass opens the website homepage. Fundamentally, all account-related operations such as password change or account deletion can also be carried out by the secret manager and the user only needs to interact with the manager's interface. As a result, users are now required to develop a different mental model and web apps lose control over their login user interfaces.

2.2 Security Threats and Mitigations

The above governance frameworks facilitate the communications between secret managers and web apps. Two prominent security threats to such communications are code injection and MitM attacks, discussed here in more detail.

Code Injection: XSS and ECI. Both HTML Autofill and Credential Management API are frameworks that require secret managers to communicate credentials in plaintext with the web app front-end. By design, these frameworks leave the user's credential accessible to JavaScript code in the web page. Therefore, an adversary that is able to inject such code is able to steal the user's credential [18,21,17,13]. Such code injection can occur through cross-site scripting (XSS) or by a malicious extension, an attack that we call extension code-injection (ECI). *Content Security Policy* (CSP) [24] is a standardised web security policy designed to mitigate against XSS, but not ECI. Alternatively, in 2014, Stock and Johns [21] proposed a *credential tokenisation* mechanism for secret managers conducting HTML Autofill which mitigates against credential theft by malicious JavaScript code. The credentials are provided on the HTML document as *tokens* and substituted with the genuine credentials in the dispatched HTTP request payload 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 cross-site scripting attackers and malicious browser extensions. In 2020, Oesch and Ruoti hunted for XSS-safe secret managers implementing this credential tokenisation mechanism and found that none did [13]. They justify such absence based on the limitations imposed by current browsers on extensions, disallowing them from manipulating HTTP request bodies. In general, tokenisation assumes that passwords are communicated without any transformations such as hashing applied to them, which is at odds with modern recommendations for password treatment such as dedicated password hashing algorithms (see e.g., [11]) and is incompatible with any other password-based protocol (e.g., SRP [27]) that the client-side scripts may wish to execute. This may well be the reason that none of the native secret managers in Chrome, Safari, or Firefox implement tokenisation either.

MitM and TPitM. Credentials travel from the user's browser to the web app's back-end, and hence are at the risk of credential theft through MitM network interception. As is widely known, TLS renders MitM interception futile as the transmitted data is encrypted. However, TLS proxies are capable of decrypting HTTP requests before relaying them to the final destination. Hence, a subverted TLS proxy in the middle (TPitM) enables an adversary to steal the transmitted credentials [14]. TLS proxies are routinely deployed in IT-managed environments, e.g., in corporate offices, and thus TPitM poses a security threat to corporate users. Unfortunately, none of the frameworks support a mitigation strategy to combat credential theft through TPitM.

2.3 Functionality and Security Comparison

Here, we give a comparison between the existing frameworks in terms of functionality and security, and provide an overview of the properties Berytus aims to achieve. The discussed properties are summarised in Table 1.

From the functionality point of view, we consider three main properties. Firstly, as we have seen, a programmable interface allows programmatic man-

Table 1. Comparison of existing frameworks based on selected functional properties (1st group), security properties (2nd group), and extra desirable properties (3rd group)

Property	Autofill	CredMgmt	ByPass
Provides programmatic secret management	-	●	●
Compatible with extension secret managers	●	-	●
Preserves UX control & user mental model	●	●	-
Enforces uniform baseline security policies	-	●	●
Provides mitigation against code injection	-	-	●
Provides further mitigation against MitM	-	-	-
Enables web app based credential mapping	-	-	-
Supports signature based authentication	-	●	-
Supports credential customisation	-	-	-

● : provides property, - : lacks property

agement of secrets, which in turn makes web app and secret manager behaviours predictable and eliminates the need to use heuristics. Only Credential Management API and ByPass provide such programmable interface. Secondly, an open framework must not only support native secret managers but also extension secret managers. Credential Management API is not currently available to extensions, but HTML Autofill and ByPass are agnostic to whether the secret manager is native or an extension. Finally, any framework should ideally be compatible with established user mental models and leave the control over the UX to the web app developers. HTML Autofill and Credential Management API do this, but ByPass radically changes the UX.

We consider three security services the frameworks can provide. Firstly, any security-conscious framework must enforce certain baseline security policies such as only allowing credentials on HTTPS web apps. Otherwise, credentials would be exposed to simple MitM adversaries. Credential Management API and ByPass do this. Secondly, mitigation against code injection is desirable as CSP is not effective against ECI and tokenisation is unlikely to gain any popularity. ByPass eliminates such threats through "bypassing" the front-end, but HTML Autofill and Credential Management API do not provide any mitigation. Finally, providing further mitigation against more sophisticated MitM attacks such as TPitM would be a bonus that unfortunately no current framework offers.

We also consider three extra desirable properties. Firstly, all existing frameworks employ *domain-based* credential mapping, which could be problematic for a web app residing under different domains, or for multiple web apps residing under the same domain, as Huaman et al. demonstrate [7]. A more accurate mapping strategy would be based on individually identified and authenticated *web apps* rather than domains. Secondly, support for automated forms of authentication such as authentication based on digital signatures, as provided by Credential Management API (WebAuthn), is a further desirable property. Finally, none of the existing frameworks are designed to support custom credential

structures, such as those with multiple usernames or multiple passwords, which could be an impediment for less conventional web apps.

As Table 1 shows, none of the existing platforms provides a good coverage of the main functional and security properties, let alone the extra desirable ones. The design aim for Berytus is to provide all these properties.

3 Proposed Governance Framework

Berytus is designed as a governance framework for programmable account registration and authentication sessions through secret managers. Berytus offers two integration pathways for web apps: base integration requiring front-end changes only, and full integration with enhanced security requiring front-end and back-end changes. Here, we will discuss the architecture and operational design.

3.1 Architectural Overview

As an orchestrator, Berytus operates natively in the browser, sitting between the web app front-end and the secret manager client. Berytus introduces two APIs, a Web API [9] for web apps and a WebExtensions API [22] for secret managers. Essentially, Berytus relays the instructions given by the web app via the Web API to the secret manager via the WebExtensions API. In this section, we unpack the components, routines, and facilities of Berytus.

Components. The building blocks of Berytus are shown in Figure 2. We conceptualise account-related processes, e.g., authentication or registration, as *operations*, each a series of one or more actions, resembling a form with multiple steps. Furthermore, we introduce the concept of a *channel* to reflect an active logical link between the web app and the secret manager. The channel holds the two *actor* objects, one for the web app and one for the secret manager, containing identifying information of each party, allowing the two sides to identify each other. Berytus also provides routines for authenticating web apps as explained later. Web apps can rely on the browser's attestation to trust the declared identity of the secret managers. Mutual identification and authentication are important for web apps such as internet banking which may wish to set a security policy that restrict their interaction to specific secret managers, and for secret managers to locate the corresponding account records in their databases.

There are two actor specialisations. The first specialisation is the *origin actor* and is exclusive for web apps. It reflects the web page's Uniform Resource Identifier (URI). The second specialisation, *crypto actor*, can be used by both the web app and secret manager. It requires the backing of a (cryptographic) signing key. A web app hosted at distinct resource locations will produce distinct origin actors, one distinct actor for each distinct URI. Conversely, if a web app 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 environments.

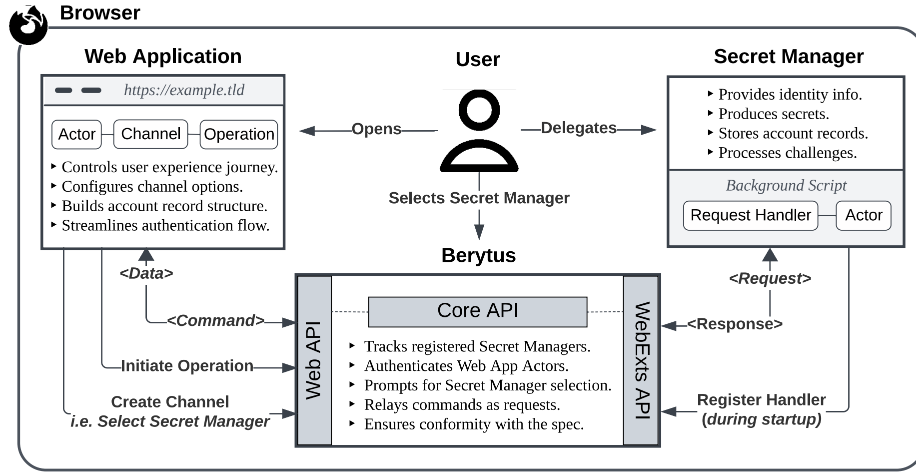

Fig. 2. Illustration of the Berytus communication model between the web application and the secret manager along with their components.

Finally, we appoint secret managers to construct a *request handler* function, i.e., a callback function, to process the web app's instructions programmatically. This is invoked to process requests such as constructing an account password field. Taking this approach ensures that secret managers are afforded programmability. Programmability is a pivotal functionality, allowing secret managers to react dynamically, instead of merely behaving as a reactive credential provider, and to fulfill business requirements, e.g., sending an email notification once a credential is transferred to the web app.

Routines. We describe the Berytus routines from a high-level perspective.

Secret Manager Registration. To track registered and running secret managers, the Berytus WebExtensions API provides two primary methods, one for registration and one for de-registration, and both are available to an extension if it specifies the `berytus` permission in its manifest file [8]. The secret manager passes a request handler during registration and, subsequently, it gets stored in Berytus. The request handler is invoked when Berytus Web API calls are dispatched by the web app. At a future point in time, the secret manager can de-register if needed and the stored request handler gets disposed.

Authenticating Web App Crypto Actors using Digital Certificates. In Berytus, web apps can be identified using origin actors or crypto actors. Origin actors are authenticated using the well-established certificate-based TLS website origin authentication. For web app crypto actors, we propose a new X.509 v3 certificate extension: Berytus Signing Key Allowlist. This certificate extension specifies a

list of one or more keys that can be used by the certificate's subject and under which URIs each key can be used. Overall, at a logical level, the web app crypto actor public key should be certified by a trusted certificate authority.

Prompting for Secret Manager Selection. When the web app initiates the creation of a web app – secret manager orchestration channel, the user is prompted through the browser UI to select a secret manager from the list of registered managers (see Figure 5 (left) in the Appendix). Following the secret manager selection, the channel is created and one or more account-related operations, e.g., registration or authentication, can be initiated.

Mediating an Authenticated Key Exchange. Berytus mediates an authenticated Diffie–Hellman key exchange, specifically the elliptic curve-based X25519 [2], using the web app's front-end as a medium to communicate cryptographic material passed between the web app's back-end and the secret manager. The exchanged keys are authenticated by each party's signing key.

Facilities. The above enable Berytus to offer the following facilities:

Unified Secret Management. By requiring secret managers to register, the browser can keep track of running managers and users can select a secret manager on a per-session basis, ensuring coordinated support for *multiple* secret managers.

Web App to Credential Mapping. Secret managers can leverage the web app actor's identifying material to distinguish between web apps and perform web app based credential mapping. The origin actor is used for website origin based identification which enables domain-based mapping, and the crypto actor is used for key-based identification which enables web app based mapping. The web app is at liberty to pick from the two actor specialisations as appropriate.

Application-level End-to-End Encryption. Following an authenticated Diffie–Hellman key exchange, Berytus facilitates end-to-end encryption (E2EE) between the secret manager and the web app. The E2EE channel is set up at the *application* level between the secret manager and the web app back-end. This provides a separate encrypted channel *within* the network-level encrypted channel provided by TLS between the browser and the web server. The web app back-end and the secret manager have exclusive access to the shared secret key for the application-level E2EE. To realise this, the web app back-end codebase needs to be changed to implement the necessary cryptographic functions.

3.2 Operational Design

We designed and implemented two operations: account creation for singing up, and account authentication for signing in. These two operations are generalised into the *login operation*. In this section, we describe the operation approval pattern followed by an overview of the login operation, including intent, account creation, and account authentication.

Table 2. Overview of supported account field specialisations in Berytus

Field Specialisation	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 [27])	salt: `0edb53..` verifier: `29c792..`	manager only
Key	public key: `010101...`	manager only
Private Key	private key: `010101...`	app or manager

Operation Approval Pattern. Once an operation has been initiated by the web app, through interaction with the channel instance, the secret manager's request handler is invoked to approve the operation creation. The secret manager should resolve this request or reject it.

Login Operation Intent. Generally, login forms prompt the user for their intent, i.e., to authenticate using an existing account, or to create a new account. We assume a similar design for the login operation as a branched operation dependent on user intent. The secret manager retrieves user intent and resolves the operation approval request as outlined in Figure 3. Alternatively, the web app can dictate the intent, e.g., in case the web app only allows authentication.

Account Creation Operation. Inspired by how web apps typically conduct registration, we design the registration process using fields and user attributes.

Credential as Fields. An account field resembles a single input field in a registration form. A field typically represents either an identity value, e.g., a username, or a secret value, e.g., a password. An account can have multiple fields, and thus a credential can be conceptualised as a set of fields. This design choice enables a high degree of flexibility for web apps in defining any combination of fields as an account record. Table 2 lists the field specialisations Berytus supports.

Field Value Production. The main piece of information for each field, the field value, can be set by either the secret manager or the web app. 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 streamlines relevant field options, e.g., a password composition policy, to enable web apps to specify and secret managers to comply with validity requirements. The decision to delegate field value production to the secret manager is made by the web app. However, as shown in Table 2, the web app cannot produce a field value for the Key field or Secure Password field as the field value's corresponding secret is not and should not be known to the web app.

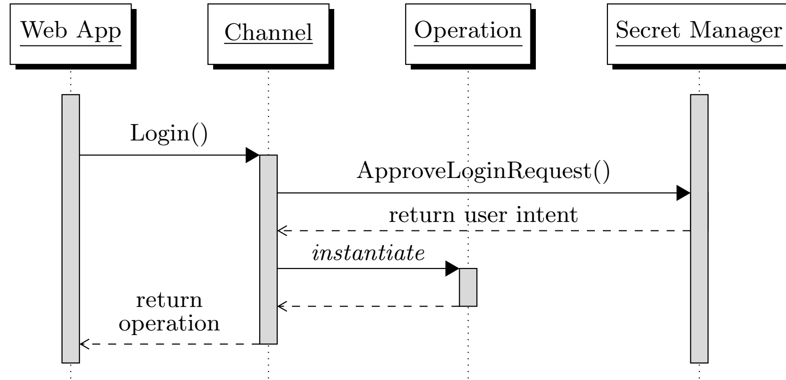

Fig. 3. (Simplified) sequence diagram showing the interactions between the web app, channel, operation and the secret manager during the Berytus login operation initiation process. The relayed user intent is either an authentication intent or registration intent.

Registering Fields. To register an account field, the web app must construct a specialised field object (`id`, `options`, `value`) and transmit it to the secret manager. The secret manager processes the received fields, produces field values for the fields with unspecified values, and returns them to the web app.

Rejecting and Revising Fields. A field value, whether produced by the secret manager or the user, can adhere to the value format but may be unusable due to app-specific constraints, e.g., username uniqueness. Hence, for robustness, the operation enables the web app to reject a produced field value by the secret manager and *request* a revision providing a rejection reason. Alternatively, the web app can *propose* a usable field value as a revision to the secret manager.

Retrieving Identity Information. Apart from fields, web apps often request identity information such as name and address during registration. As such, the web app can retrieve the user's identity information, based on the OpenID Standard Claims [16, Section 5.1], from the secret manager.

Account Categorisation. Web apps can set a category on the account record. The account category is used as an arbitrary account type identifier, e.g., specifying a user role. This can be specified as an option in the account authentication operation to assist the user in selecting an appropriate account in cases where users may have multiple accounts with the same web app.

Saving the Account and Transitioning to Account Authentication. To finalise the registration operation, the web app instructs the secret manager to save the account record into its database. If successfully saved, the finalised account creation operation can be transformed into an account authentication operation for the newly-created account record.

Account Authentication Operation. The account authentication operation encompasses the steps involved in user authentication. At a minimum, a web app typically employs two challenges, an identification challenge and an authentication challenge. In the same spirit, we design and implement a *challenge-based* authentication paradigm in which authentication is composed of a number of challenge–response rounds carried out sequentially.

Account Selection. To process an authentication operation, the secret manager must first assume a registered account. It is expected that the secret manager would prompt the user to select an account using its own UI facilities during the early phases of the operation, e.g., in the login operation approval process, the secret manager can prompt the user to select an account if any exists.

Challenge Communication Flow. The challenge is designed as a message passing interface: the web app sends a message and the secret manager responds to it with the expected data. This enables the implementation of multi-step protocols such as the Secure Remote Password (SRP) protocol [27].

Supported Challenges. Berytus is designed to support password authentication, digital signature-based authentication, SRP authentication, and off-channel one-time password authentication (e.g., by email or phone). Furthermore, web apps can initiate custom authentication challenges backed by a messaging JSON schema to validate the communicated messages. Hence, an acquainted web app and secret manager pair can organise custom challenges at run time.

Approving the Challenge. Once a web app initiates a challenge, Berytus contacts the secret manager to approve the challenge initiation request. Following its approval, the web app can begin sending messages to the secret manager.

Closing or Aborting the Challenge. When the web app is satisfied by the response, it can close the challenge to imply successful completion. Otherwise, the web app can abort the challenge by providing an abortion reason code.

4 Security Evaluation

To evaluate the security aspects of Berytus, we discuss the security services it provides and how they help mitigate against prominent credential-theft attacks in comparison with existing frameworks.

4.1 Security Services

As an orchestrator, Berytus is able to provide security services to web apps and secret managers, including authentication of web applications and application-level end-to-end encryption. We discuss these services here in more detail.

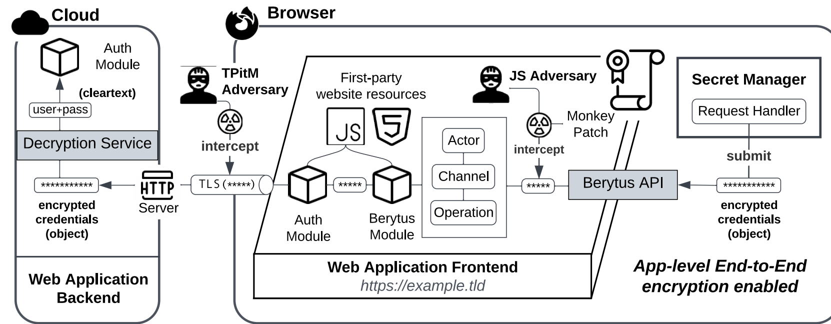

Fig. 4. Berytus E2EE & its effectiveness against TPitM and monkey-patching attacks

Certificate-based Web App Authentication. Berytus enables *web app* authentication through certified key identifiers (see crypto actor in Section 3.1). This is in contrast with using a certified domain name for *website origin* authentication via TLS certificates. This provides the following benefits.

Accurate Credential Mapping. Domain-based credential mapping, used currently by secret managers, has been shown to produce several issues including inaccurate mappings when the same authentication system is used on multiple domains [7]. Authenticating web apps rather than their website origin enables the more accurate *web app based credential mapping* (see Facilities in Section 3.1). This is an origin-agnostic credential mapping approach that is better tailored to the distributed nature of web apps and addresses the identified need in the literature for "using credentials in multiple environments" [7].

Security Against Web App Impersonation. Malicious web apps may impersonate other apps to phish user credentials. Both web app authentication and website origin authentication protect against such impersonation. However, the former is tailored towards web apps possibly deployed on multiple domains, while the latter is tailored towards websites solely hosted on distinct domains.

Application-level End-to-End Encryption. As Berytus supports equipping web apps with cryptographic keys, it can facilitate an authenticated Diffie–Hellman key exchange between the secret manager and the web app back-end (see Routines in Section 3.1). This provides the following benefit.

Encryption-based Security Against MitM Attacks. Although network-level encryption provided by TLS can guard against general MitM attacks, it is ineffective against the more sophisticated TPitM attacks. Application-level E2EE facilitated by Berytus establishes a secure (logical) channel between the secret manager and the web app back-end, providing an effective mitigation against credential theft by any adversaries on this path, regardless of their security privilege. Figure 4 clarifies how E2EE works in Berytus.

4.2 Resistance Against Attacks

In this section, we discuss relevant mitigation strategies in Berytus and other frameworks to combat prominent credential phishing and theft attacks.

Threat Model. There are three broad types of entities in the web user authentication and secret management ecosystem: web apps, the browser, and browser extensions (including secret managers). We assume that the browser's native code and its privileged execution context, including Berytus, are trusted. We also assume that secret managers are trusted entities, but web apps and other browser extensions could be 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 acting as their secret manager, the extension is given such permission and hence trusted. Moreover, we assume that the user's secret manager will only transfer credentials under the same credential mapping strategy as that when it was stored, 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 also assume that certificates are trusted.

Compared Frameworks. Here, we focus on comparing HTML Autofill and Credential Management API with our proposed framework, Berytus, as these three are all client-side frameworks and many of the security services we consider here are client-side services. ByPass uses a significantly different architecture (see Figure 1), bypassing the client-side, and is discussed towards the end.

Credential Theft Attacks. We consider code injection attacks, specifically XSS and ECI, and MitM attacks, specifically by general non-privileged adversaries (denoted by MitM) and by privileged adversaries such as TLS proxies (denoted by TPitM). For code injection attacks, CSP is effective against XSS, but not against ECI, as CSP can prevent code injection from entities external to the browser, but extensions will still have the ability to inject JavaScript code in the execution context. CSP is available in all three frameworks and can be considered an orthogonal service for extra assurance. Credential tokenisation however, while effective against both XSS and ECI, is not available in any of the frameworks, and given its limitations, is not expected to be widely-deployed. For MitM attacks, network-level TLS encryption is effective against general MitM attacks, but not against TPitM attacks. The application-level E2EE service provided by Berytus on the other hand is effective both against code injection attacks and against TPitM attacks. E2EE ensures that only the secret manager and the web app back-end have access to plaintext credentials. This means that the web-app front-end (and hence any code injected into it) or any party in the middle (such as a TLS proxy) can only get their hands on encrypted credentials and both types of attacks are rendered ineffective. These discussions are summarised in Table 3.

Table 3. Selected security services, frameworks on which they are available (out of HTML Autofill, Cred. Mgmt. API, and Berytus), and security service effectiveness against credential-theft attacks (XSS, ECI, MitM, and TPitM)

Security Service	Framework	XSS	ECI	MitM	TPitM
Content Security Policy	all	●			
Credential Tokenisation	none	●	●		
Network-level TLS encryption	all			●	
Application-level E2EE	Berytus	●	●	●	●

●: credential theft via given attack is mitigated

Credential Phishing Attacks. In these attacks, users are lured into malicious websites that resemble other legitimate ones in the hope to steal the user's credentials for the legitimate websites. Web app origin authentication, via TLS certificates, along with domain-based credential mapping can effectively protect users against such attacks and are widely used by secret managers across all frameworks. Berytus provides a further alternative mitigation in this regard via the combination of app authentication and app-based credential mapping. As discussed before, in certain scenarios where there is no clear one-to-one mapping between web apps and domains, e.g., same apps on multiple domains, or multiple apps on the same domain, app-based mapping may be more appropriate and hence more accurate. Secret managers are free to choose the best mapping strategy in Berytus for the web apps with which they are communicating.

ByPass. As in ByPass, secret managers directly communicate with web app back-ends, the code injection threat is eliminated as no login webpage is rendered. TLS is still effective against general MitM attacks, however a TLS Proxy will still be able to decrypt the communications between the secret manager and the web app back-end and since ByPass does not provide any further protection in this regard, it remains vulnerable to TPitM attacks.

5 Functionality Evaluation

Deployability is discussed as a crucial property of security-enhancing technologies [4] and a significant factor for secret manager adoption [15,4]. A framework that is designed to orchestrate the relationship between secret managers and web apps hence need to consider the developers on both sides. In this section, we discuss this aspect and evaluate Berytus in terms of functionalities it provides for secret managers and web apps. Comparison between existing frameworks and Berytus with respect to these functionalities is presented in Table 4.

Support for Multi-Step and Multi-Factor Authentication. Existing frameworks only support "one-shot" scenarios, where all credentials are expected to be transmitted in one instruction. Berytus models provision of multiple authentication

Table 4. Functional capability comparison between existing frameworks and Berytus.

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

credentials as successive *challenges*, where each challenge is responded to in a series of instructions. This allows flexible support for multi-step and multi-factor authentication protocols, including multi-round protocols such as SRP.

Developer Control over UX. Berytus leaves the web app and secret manager developers in charge of UX and does not enforce any requirements on their UIs. The automation is to a large extent transparent to the user and occurs at the script level. This is in contrast with ByPass which does away with the web app UI and enforces a predetermined UX. Although web app developers have control over the login UI in HTML Autofill, they cannot control when the credential transfer is conducted, and hence they only have partial control over the UX.

Maintaining the Conventional Usage Pattern. A typical user expects to visit a web page on their browser to log in to the corresponding web app. Client-side frameworks 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. Berytus is designed to work with both native and third-party secret managers that are installed as extensions. This, in turn, supports choice for end users and encourages an open ecosystem. Credential Management API however is designed for native secret managers.

Flexibility over Account Design. Berytus allows web apps to define their account structure as any combination of the supported fields. Besides, Berytus has built-in support for categorisation which allows the possibility of multiple role-based accounts for the same user on the same web app.

Support for Custom Authentication. Berytus allows for custom authentication challenges to be initiated following approval from the selected secret manager. These challenges are in the form of custom messages validated using an app-specified JSON schema, and allow flexibility in extending the supported authentication mechanisms. Existing frameworks 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 an allowlist of approved secret managers. Such a facility is not provided by any of the existing frameworks.

6 Conclusions and Future work

We presented Berytus, a novel framework for web-based authentication, which supports the registration and authentication processes, including identity information management and external identity (e.g., phone or email) attestation. It supports extension secret manager as well as native user agents, thereby maintaining ecosystem openness for users of any browser-based secret manager.

Our security evaluation demonstrated that Berytus provides valuable security services which mitigate against prominent attacks. Notable among these services is the provision of end-to-end encrypted channels between secret managers and web app back-ends which render credential theft attacks via code injection ineffective, even when they are carried out by entities with privileged access such as browser extensions and TLS proxies.

Through our functionality evaluation, we showed that Berytus is the only platform of its kind that provides programmatic access to web apps and secret managers while supporting both native and extension secret managers and preserving control over user experience for developers and usage mental models for users. Flexible account design and support for multi-step, multi-factor, and custom authentication protocols are among other benefits provided.

While it is expected that such a solution will have some integration effort required both from web applications and secret managers, Berytus tries to minimise such effort, keeping various services requiring additional effort, e.g., web app authentication and authenticated key exchange, entirely optional. A detailed analysis of the integration effort can be found in [6, Section 4.2.2]. We suggest future work to conduct studies with developers and end users to investigate the usability of Berytus integration and utilisation in more detail.

References

1. Balfanz, D., et al.: Web authentication: An API for accessing public key credentials, W3C recommendation (2019), https://www.w3.org/TR/webauthn-1
2. Bernstein, D.J.: Curve25519: New diffie-hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography - PKC 2006. pp. 207–228. Springer Berlin Heidelberg (2006)
3. Blanchou, M., Youn, P.: Password managers: Exposing passwords everywhere (2013), white paper, iSEC Partners
4. Bonneau, J., Herley, C., Oorschot, P.C.v., Stajano, F.: The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In: IEEE Symp. Security and Privacy. IEEE (2012)
5. Carr, M., Shahandashti, S.F.: Revisiting security vulnerabilities in commercial password managers. In: IFIP SEC '20, AICT 580. pp. 265–279. Springer (2020)

6. Cherry, A.: A Secure Password Manager Governance Framework for Web User Authentication. MSc by research thesis, University of York (2024)
7. Huaman, N., Amft, S., Oltrogge, M., Acar, Y., Fahl, S.: They would do better if they worked together: The case of interaction problems between password managers and websites. In: IEEE Symp. Secuity & Privacy. pp. 1367–1381. IEEE (2021)
8. Mozilla: manifest.json, https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json
9. Mozilla: Web APIs (2023), https://developer.mozilla.org/en-US/docs/Web/API
10. Mozilla: Using the fetch API, web APIs, MDN (2024), https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
11. NIST: Password hashing competition, https://www.password-hashing.net/
12. Oesch, S., Gautam, A., Ruoti, S.: The emperor's new autofill framework:a security analysis of autofill on IOS and android. In: ACSAC '21. pp. 996–1010. ACM (2021)
13. Oesch, S., Ruoti, S.: 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. pp. 2165–2182. USENIX Association (2020)
14. O'Neill, M., Ruoti, S., Seamons, K.E., Zappala, D.: TLS proxies: Friend or foe? In: Internet Measurement Conference. pp. 551–557. ACM (2016)
15. Pearman, S., Zhang, S.A., Bauer, L., Christin, N., Cranor, L.F.: Why people (don't) use password managers effectively. In: SOUPS '19. pp. 319–338. USENIX (2019)
16. Sakimura, N., et al.: OpenID connect core 1.0 incorporating errata set 1 (2023), https://openid.net/specs/openid-connect-core-1_0.html#StandardClaims
17. Satragno, N.: Credential management, W3C working draft, https://www.w3.org/TR/credential-management-1
18. Silver, D., Jana, S., Boneh, D., Chen, E., Jackson, C.: Password managers: Attacks and defenses. In: USENIX Security '14. pp. 449–464. USENIX Association (2014)
19. Stajano, F., Spencer, M., Jenkinson, G.: Password-manager friendly (PMF): Semantic annotations to improve the effectiveness of password managers. In: Technology and Practice of Passwords (Passwords 2014). LNCS, vol. 9393, p. 61 (2014)
20. Stobert, E., Safaie, T., Molyneaux, H., Mannan, M., Youssef, A.: ByPass: Reconsidering the usability of password managers. In: Park et al. (ed.) Security & Privacy in Communication Networks. pp. 446–466. Springer (2020)
21. Stock, B., Johns, M.: Protecting users against XSS-based password manager abuse. In: AsiaCCS'14. pp. 183–194. ACM (2014)
22. Swan, A.: Webextensions logins API proposal (2016), https://github.com/web-ext-experiments/logins/blob/master/api.md
23. W3C WebAuthn Adoption Group: Terms: Third-party passkey provider (2024), https://passkeys.dev/docs/reference/terms/#third-party-passkey-provider
24. West, M., Sartori, A.: Content security policy, W3C working draft, https://www.w3.org/TR/CSP/
25. WHATWG: Autofill section, HTML standard (2024), https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#autofill
26. WHATWG: Form submission section, HTML standard. (2024), https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#form-submission-2
27. Wu, T.D.: The secure remote password protocol. In: NDSS. Internet Society (1998)

A Proof of Concept Implementation

We briefly discuss the proof of concept implementation in this section. All project artefacts are available at https://github.com/alichry/berytus. Build instructions are provided as well as binaries for our extended Firefox browser.

API Implementation. To demonstrate the feasibility of our framework, we have extended the Mozilla Firefox browser (v116.0a1) to implement the Web API and the WebExtensions API natively. These two ends are linked with one another through the Berytus Core API. The Core API handles secret manager selection (Figure 5 (left)), channel and operation management, inter-process communication and other functional requirements. The Web API is implemented in C++, the WebExtensions API and the Core API in TypeScript/JavaScript.

Secret Manager Implementation. We developed *Secret** ("secret star"), a secret manager that integrates with the Berytus WebExtensions API. Secret* has three components: (1) The background script, where the request handler is implemented and registered; (2) The user interface facility which hosts the extension pop up pages used for user prompts; (3) The storage facility to store and retrieve data when processing requests. Figure 5 (right) shows how Secret* retrieves the user's intent during a login operation.

Web Application Implementation. We developed a full-stack web application with an authentication subsystem and proceeded to integrate Berytus. This serves as a real-world example, demonstrating feasibility and showing the changes needed on the server-side and client-side to complete the integration.

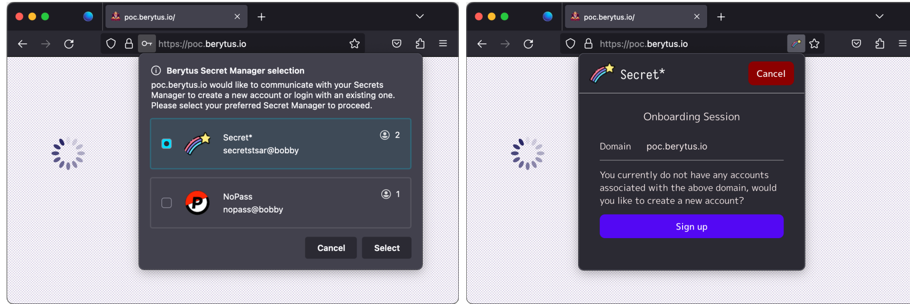

Fig. 5. Left: Berytus secret manager selection prompt. Each listed secret manager displays the number of registered accounts associated with the web app. Right: Secret* login operation approval prompt when the user does not have any registered accounts. In both cases, domain-based credential mapping was used.