Vault Raider: Stealthy UI-based Attacks Against Password Managers in Desktop Environments

Andrea Infantino, Mir Masood Ali, Kostas Solomos, and Jason Polakis
University of Illinois Chicago
{ainfan5, mali92, ksolom6, polakis}@uic.edu

***Abstract*—Password managers significantly improve password-based authentication by generating strong and unique passwords, while also streamlining the actual authentication process through autofill functionality. Crucially, autofill provides additional security protections when employed within a traditional browsing environment, as it can trivially thwart phishing attacks due to the website's domain information being readily available. With the increasing trend of major web services deploying standalone native apps, password managers have also started offering *universal autofill* and other user-friendly capabilities for desktop environments. However, it is currently unknown how password managers' security protections operate in these environments. In this paper, we fill that gap by presenting the first systematic empirical analysis of the autofill-related functionalities made available by popular password managers (including 1Password and LastPass) in major desktop environments (macOS, Windows, Linux). We experimentally find that password managers adopt different strategies for interacting with desktop apps and employ widely different levels of safeguards against UI-based attacks. For instance, on macOS, we find that a high level of security can be achieved by leveraging OS-provided APIs and checks, while on Windows we identify a lack of proper security checks mainly due to OS limitations. In each scenario, we demonstrate proof-of-concept attacks that allow other apps to bypass the security checks in place and stealthily steal users' credentials, one-time passwords, and vault secret keys through unobservable simulated key presses. Accordingly, we propose a series of countermeasures that can mitigate our attacks. Due to the severity of our attacks, we disclosed our findings and proposed countermeasures to the analyzed password manager vendors, which has kickstarted the remediation process for certain vendors and also been awarded a bug bounty. Finally, we will share our code to facilitate additional research towards fortifying password managers.**

I. Introduction

As the modern web revolves around predominantly dynamic and personalized content, authentication remains an integral and critical aspect of users' browsing experience [1], [2], [3], [4]. While passwordless authentication paradigms have emerged in recent years [5], [6], passwords remain the *de facto* method for end-user authentication in many settings [7], [8]. Password-based authentication has been extensively analyzed in prior research spanning more than two decades, detailing their various shortcomings and the challenges users face [9]. This includes users choosing weak or guessable passwords [10], [11], their inability to remember strong passwords [12], the impact of an ever-increasing number of accounts and passwords [13] which leads to password reuse across services [14], [15], and, finally, users' inherent susceptibility to phishing attacks [16], [17].

At the heart of these issues lie the natural limitations of humans, which motivated the proposal of password managers as a means to address the aforementioned shortcomings. Even though their core functionality and goals remain unchanged, password managers have evolved considerably compared to early academic proposals [18], [19], [20], with a particular focus on improving usability concerns [21], [22], [23] that can hinder wider adoption. As a result, popular password manager browser extensions have millions of downloads in Google Chrome's Web Store. Apart from the ability to generate strong and unique passwords, these browser extensions also automatically fill out (i.e., *autofill*) user credentials in websites' account registration and log-in forms, thereby significantly streamlining the authentication process. By verifying the visited website's domain, which is made readily available by browsers through the `URL` Web API [24], password managers protect against phishing attacks. Importantly, certain password managers also operate as authenticators for two-factor authentication, further mitigating the effects of phishing or credential stuffing attacks [25].

While web browsers continue to mediate a significant portion of users' online activities, many major web services have started to offer standalone desktop apps as a means to improve the user experience (this can, at least partially, be attributed to the declining performance of browsers due to bloat [26], [27]). Accordingly, this migration from browsers to native apps has necessitated that password managers also pivot to desktop ecosystems. Given the key importance of autofill functionality and the paramount importance of usability, certain password managers have started offering *universal autofill* functionality, wherein they will autofill users' credentials in the login forms of non-browser applications. Despite autofill functionality being conceptually straightforward, *securely* completing the autofill process *outside* of the sandboxed and isolated confines of a browser requires introducing new types of safeguards. Moreover, it is unknown what system-level features and APIs are made available across different operating systems that could facilitate, or undermine, this process.

In this paper, we address this gap by offering the first,

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231067
www.ndss-symposium.org

to the best of our knowledge, empirical security analysis of password managers for native desktop application environments. Specifically, we focus on autofill and other functionality features that aim to obviate the friction that can arise during the authentication process. To that end, we first explore the various user-friendly features offered by popular managers across different platforms. Subsequently, we analyze popular password managers in-depth and uncover what mechanisms are in place for preventing applications from exploiting their autofill functionality for credential-stealing attacks. We also explore what, if any, OS-level APIs and features are leveraged as part of their security posture. Our analysis uncovers a tale of extreme divergence in terms of safeguards employed by password managers within the same operating system, as well as for the same password managers across different operating systems. Indicatively, we find that 1Password on macOS employs a series of countermeasures against phishing apps, including detecting multiple active processes with the same `Bundle ID`, mapping applications to trusted `Associated Domains`, and verifying apps through macOS' `Code Signing` APIs. In contrast, 1Password for Windows does not employ comparable security checks due to the lack of corresponding OS-level mechanisms, highlighting the challenge of implementing autofill when faced with insufficient platform support.

Guided by our findings, we demonstrate novel attacks that exploit the functionality offered by desktop password managers for exfiltrating users' data, including their credentials, 2FA codes, and vault secret key. For less secure password managers (e.g., Keeper), the attacks can be as simple as spoofing the target application's `Bundle ID`, while in more secure apps (e.g., 1Password), we use synthetic keyboard clicks for leveraging the password manager's *Quick Access* functionality. We then detail how a series of system-specific UI-related features can be leveraged by malicious applications for *hiding* the password manager functionalities that they trigger from users, thereby rendering our attacks stealthy. Interestingly, we find that macOS and Windows support multiple `window` manipulation techniques that allow for fully stealthy attacks without visual artifacts. To make matters worse, we also demonstrate that our attacks enable local privilege escalation by harvesting system passwords, and allow an attacker to replicate a user's vault on a separate device by extracting the secret key. Guided by our attack techniques, we propose mitigations that would allow password managers to detect our attacks and avoid exposing users' data. Due to the critical role of password managers and the severe implications of our attacks, we have already disclosed our findings and proposed mitigations to the respective vendors. Certain vendors have acknowledged the severity of our attacks and have initiated remediation efforts to mitigate the underlying weaknesses that could significantly impact their user bases.

In summary, our research contributions are:

- We present the first in-depth empirical analysis of password manager functionality in desktop application ecosystems. We also explore what mechanisms are exposed by operating systems and adopted by password managers for safeguarding autofill-related functionality.
- We demonstrate novel attacks that exploit password managers' functionality and manipulate UI features for stealthily exfiltrating secrets from their vaults.
- We propose countermeasures that can be incorporated by password managers for effectively mitigating our attacks.
- We have responsibly disclosed our findings to the affected vendors to kickstart remediation efforts. To enable additional research on password manager security, our source code and attack demonstrations are available on our artifacts page [28].

II. Background and Threat Model

Here we provide background information on password manager features and functionality, and detail our threat model.

Form Autofill Functionality. Autofill functionality is a core feature in modern web browsers, designed to store and automatically populate data such as login credentials, addresses, payment details, and other frequently used information in web forms [29]. When users input data into form fields, browsers typically prompt them to save this information for subsequent use. Once stored, the autofill mechanism identifies matching fields across different web pages and retrieves the corresponding data to streamline form completion. This browser functionality is integrated across devices, enabling consistent user experiences. Sensitive data, including login credentials, is stored locally within the browser and can be synchronized with cloud services [30], [31], [32].

The autofill process operates by identifying and interpreting form field elements through the analysis of visible and hidden attributes, such as usernames and passwords, and associating them with previously stored credentials. Detection mechanisms adapt to variations in web forms by analyzing structural features and semantic markers (e.g., the HTML5 autofill attribute) [33]. These mechanisms also handle additional obfuscated or dynamically generated fields to support more advanced web structures. Once fields are classified, the system selects the best-matching stored data for automatic completion.

Password Managers. Password managers are available both as integrated browser features and as standalone applications for desktop and mobile platforms. Browser-based password managers focus on credential autofill while offering additional features such as password generation, secure storage, and password strength analysis, for enhancing account protection [34]. In addition to their browser-focused implementations,password managers also operate as standalone desktop applications, integrating into native environments. These applications, such as 1Password [35], LastPass [36], and Keeper [37], provide cross-platform synchronization, enabling users to securely access their data across multiple devices. Password managers also support the management and secure storage of a wide range of sensitive information, including credit card information, one-time passwords (OTPs), and identity information, ensuring comprehensive protection and accessibility. Given the sensitivity of the information they manage, password managers implement robust security

measures, including master passwords and two-factor authentication (2FA) [38]. User authentication is required to access encrypted data, with advanced tools employing zero-knowledge encryption, ensuring that even service providers cannot access stored information [39].

Threat Model. Our research focuses on the operation of standalone password managers that offer autofill functionality *outside* of the confines of web browsers. As such, we adopt the typical threat model used in security studies that focus on native app ecosystems (i.e., non-browser environments). We assume that the user has installed a malicious application — the specific propagation method is outside the scope of our study. Malicious or invasive software is a common occurrence in desktop environments, including macOS despite the wide misconceptions that exist about its purported immunity to malware [40]. In practice, attackers can achieve this through straightforward social engineering attacks, or more advanced attacks that pass Apple's app vetting process [41], or a man-in-the-middle attack against Apple's Wireless Direct Link [42]. It is important to note that macOS's default app protection mechanism, Gatekeeper, *does not* prevent users from installing malware. Instead, it primarily serves as a warning system, verifying that downloaded apps are signed by an identified developer and notarized by Apple [43]. Notarization is a one-time verification process that does not provide continuous monitoring, allowing apps to introduce malicious behavior post-installation. In practice, while Gatekeeper can issue warnings for unsigned or unnotarized applications, signed malicious software continues to be distributed and executed [44], [45], [40]. Recent macOS malware campaigns have, in fact, explicitly leveraged signed binaries to evade Gatekeeper's protections [46]. Moreover, users can override system protections to install unverified applications [47], [48], [49]—a common practice for software distributed outside the App Store.

Prerequisites. For simplicity, in our analysis we assume that the user's password manager application is running, which is a reasonable assumption as these applications are often left open in the background for convenience. Nonetheless, malicious apps can leverage system-level APIs (e.g., `ps` on macOS or `tasklist` on Windows) to monitor active processes and launch the attack only after confirming that the password manager is running, thereby reducing the risk of detection. Moreover, on macOS, we assume that the malicious app obtains the `Accessibility` permission during installation, a typical permission required by a wide range of legitimate software, including screen recording and remote control applications [50]. Notably, permission requests and UI automation have been leveraged in real-world malware campaigns to perform tasks such as simulated input events and data exfiltration [51], [52]. In our attack, Accessibility access is incorporated into the broader workflow that ultimately exploits password managers' autofill behavior. We further discuss our assumptions' practicality in §IV and how to relax our propagation vector assumptions in §VIII.

Malicious application. The application appears to be legitimate and can be bundled with popular software or disguised as a common utility. Once active, it targets specific password managers and exfiltrates stored credentials (usernames and passwords), banking information, one-time passwords, and vault secret keys (if applicable). Our malicious application prototype operates entirely at the user-level space, without exploiting kernel-level vulnerabilities or requiring special administrative privileges (beyond the Accessibility permission in macOS). Our attacks exploit design flaws in password managers' autofill-related functionality and UI-related features for extracting sensitive data in a stealthy manner. We emphasize that the attack *does not* rely on zero-day exploits or implementation bugs, but instead leverages existing functionality intended for users. Through variations across different password managers and OSes, our research demonstrates an advanced threat that exploits vulnerabilities in the handling of UI elements and synthetic user events. By manipulating these components, the attacker bypasses existing security checks and tricks the password managers into exposing sensitive data.

TABLE I: Summary of analyzed password managers.

Password Manager	Version	OS			Popularity
1Password	8.10.48	✓	✓	◑	>15M users [53]
Keeper	16.10.13	✓	✓	N/A	>1M paying users [54]
LastPass	4.14.1.0	N/A	✓	N/A	>33M users [55]
KeePassXC	2.7.9	✓	✓	✓	>691K Linux installs [56]
MacPass	0.8.1	✓	N/A	N/A	>6.8K GitHub stars [57]

◑ indicates partial support, where a password manager is available on the operating system but lacks full feature compatibility.

III. Experimental Setup

Selecting and Configuring Password Managers. We selected password managers based on two criteria: (i) the presence of autofill functionality in native applications and (ii) broad platform support. While password managers provide autofill capabilities in web browsers—typically through browser extensions—fewer have implemented this feature for native applications. Since our study focuses on attacks targeting autofill functionality beyond the browser, we prioritized password managers that explicitly support autofill in native applications. Table I summarizes the key characteristics of the evaluated password managers, including their supported platforms and popularity. Notably, a subset of password managers offer full cross-platform support, while others, like MacPass, are restricted to specific operating systems.

Attack Workflow. Our credential harvesting attack is executed through a custom phishing application that performs the required steps to exfiltrate credentials. The attack proceeds through the following stages: (i) Upon launch, the application renders its interface and embeds input components for credential capture, such as hidden form fields. (ii) To impersonate a trusted application and bypass verification checks, it modifies identity attributes such as the window title, bundle metadata, or display name. (iii) It generates synthetic keyboard inputs to trigger the password manager's autofill or credential access components (e.g., Quick Access). (iv) Once verification suc-

TABLE II: Overview of our attacks across different password managers and platforms. *Credential Types* indicates the categories of sensitive data that can be harvested: usernames, passwords, one-time passwords (OTPs), and credit card information. *Permissions* indicates whether the attack requires additional OS-level permissions (✓) or not (✗) for the respective platform; N/A denotes that the password manager is not supported on that platform. *Stealthiness* denotes whether the attack is fully stealthy (●), partially stealthy (◑), or not stealthy (✗) on each OS. *Performance* reports the time required to harvest a pair of credentials under the primary attack variant on macOS, where applicable.

Password Manager	**Credential Types**				**Permissions**			**Stealthiness**			**Performance**
	Username	**Password**	**OTP**	**CC Info**							
1Password	✓	✓	✓	✓	✓	✗	✗	●	●	✗	11s
Keeper	✓	✓	✓	✓	✓	✗	N/A	◑	●	N/A	4s
LastPass	✓	✓	✓	✓	N/A	✗	N/A	N/A	●	N/A	5s
KeePassXC	✓	✓	✓	✗	✓	✗	✗	●	●	✗	17s
MacPass	✓	✓	✓	✗	✓	N/A	N/A	●	N/A	N/A	5s
macOS Keychain	✓	✓	✗	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A

ceeds, the password manager either injects credentials directly into the phishing application's hidden fields or displays the vault contents for selection. The phishing application then simulates interactions to extract the displayed credentials and transfer them into its own interface for exfiltration (e.g., copy-paste actions). (v) To maintain stealthiness, the application overlays distracting content (e.g., a video player) and leverages window layering attributes to remain in the foreground and/or hide password manager windows. The methodology is consistent across password managers; differences arise from OS-level constraints or manager-specific behavior.

Testing Environment. We created test accounts with each password manager to ensure that no real user data was at risk. All evaluated password managers supported keyboard shortcuts to trigger autofill by default, except KeePassXC, which disables this feature by default; for consistency, we enabled autofill for our tests. Prior to each experiment, we ensured the password managers were running, unlocked, and populated with fake credentials associated with real services. We deployed our malicious application using the Electron framework and its window management capabilities [58], enabling consistent cross-platform deployment. To simulate user interactions and trigger autofill mechanisms, we integrated the RobotJS library [59]. All experiments were conducted locally on macOS (MacBook Pro, M3, Sonoma 14.6.1), Windows (Galaxy Book Pro 360, Windows 11 Home), and Linux (HP 15-bw0xx, Ubuntu 22.04.5 LTS).

Overview. Table II summarizes our attacks, detailing the targeted OSes, required privileges, stealthiness, and performance for the primary attack variant (e.g., email-password credentials). For password managers available in multiple OSes, we report the performance on macOS. In the following sections, we discuss each scenario in detail, highlighting any unique features employed by each password manager, as well as the intricacies of our attack in each scenario.

IV. macOS Password Managers

Here, we analyze the autofill functionalities of macOS password managers. In §IV-A, we demonstrate our analysis of 1Password to achieve effective credential harvesting. In §IV-B, §IV-C, and §IV-D, we extend our analysis and introduce variations of the attack across other password managers, highlighting commonalities and differences in their verification mechanisms. We focus our presentation on 1Password for the following reasons: (i) it is one of the most popular password managers, (ii) it is highly recommended (e.g., The New York Times has recommended it multiple times as the best password manager [60], [61]), and (iii) it incorporates the most OS-specific safeguards, thereby providing a prime example of the level of security that can be achieved in macOS.

macOS Features and Security Mechanisms. Autofill functionality in standalone password managers introduces *additional* security challenges not present in their typical browser-based usage, where credentials are filled within a well-structured and sandboxed environment. Here we outline structural and security-relevant aspects of macOS applications that are pertinent to our threat model.

App Components. macOS apps are built upon a structured packaging system and robust security mechanisms, designed to ensure their integrity, usability, and interaction with the operating system. Specifically, they are distributed as bundles—self-contained directories that include executable code, resources, and metadata required for the app's functionality. A fundamental component of the bundle is the Information Property List (`Info.plist`), a configuration file that stores essential app information [62]. It specifies the version number, supported architectures, localization settings, and entitlements for accessing system resources.

Each app includes a unique `Bundle ID` [63], which the OS uses to manage app identity, enforce permissions, and isolate inter-application interactions. Similarly, the `Bundle Display Name` serves as the primary label that users rely on to interact with apps in the system's interface, such as the Finder, ensuring accurate app identification. macOS also enforces a robust security model to ensure the integrity and authenticity of apps. Another key component is *code signing*, a process that allows developers to cryptographically sign their apps [64], [65]. This mechanism verifies app integrity and authenticity, ensuring it has not been modified by other apps, malware, or during distribution.

Permissions. The Accessibility API serves as a foundational component that enables apps to support assistive technologies [66]. It allows developers to programmatically test and enhance app usability by simulating interactions, identifying missing accessibility labels, and optimizing layouts. Access to the Accessibility API requires explicit user approval, managed through macOS privacy settings, where users must explicitly grant the permission for the API to become available.

Credential Autofill Mechanisms. Across evaluated password managers, we identify two primary credential mechanisms: `Autofill` and `AutoType`. Autofill populates UI fields directly by leveraging Accessibility API and interacts with input elements based on their context (e.g., email) [67]. In contrast, AutoType emulates user-triggered keystrokes for autofill, without programmatic interaction or field validation [68], [69]. In our evaluation, we determine the adopted mechanism for each password manager and exploit it accordingly.

Attack Vectors. As macOS enforces stricter authentication and verification models, it provides a representative platform for evaluating attack feasibility under robust OS-level constraints. We focus our analysis on macOS to demonstrate that password managers are vulnerable to credential-harvesting attacks, despite operating within environments that adopt advanced platform-level defenses. Specifically, our analysis reveals inconsistencies in autofill validation that allow attackers to bypass verification mechanisms and extract credentials. Our attacks exploit three primary flaws: (i) manipulation of application identification (e.g., spoofed Bundle IDs), (ii) exploitation of UI interactions, and (iii) abuse of system-level configurations (e.g., window layering). By leveraging these techniques, an attacker can systematically and stealthily harvest credentials from multiple password managers. We note here that all variations of our attack leverage the macOS Accessibility permission to programmatically interact with the password manager and automate the credential harvesting.

Given that legitimate applications request Accessibility permissions, we consider this assumption to be realistic. Specifically, our analysis of 100 of the most popular free apps from the macOS App Store revealed that approximately 20% request this permission, which suggests that users are accustomed to often granting this permission to apps they install. Moreover, prior studies in different domains (e.g., Android) have assumed or explored the presence of malicious applications with the Accessibility permission [70], [71], [72], [73] or reported its use by malware in the wild [52].

A. 1Password

We focus our initial analysis on uncovering potential security and verification flaws that would allow effective credential harvesting attacks against 1Password.

Autofill Verification. The autofill mechanism provided by 1Password is activated through dedicated keyboard shortcuts [74], enabling users to launch a search interface known as the *Quick Access Window* (Figure 1). This interface offers a list of stored credentials that users can select for autofilling into the corresponding application. To mitigate unauthorized access and credential exfiltration, 1Password incorporates protections against malicious applications that attempt to exploit its autofill functionality. These protections align with the types of attacks outlined in our threat model, further emphasizing the realistic threat that such attacks pose to users. Due to the proprietary closed-source nature of the 1Password application, we conduct an empirical black-box analysis to identify its internal security mechanisms and verification processes.

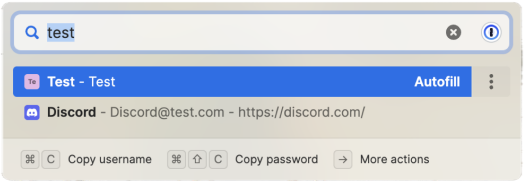

Fig. 1: 1Password's Quick Access window.

Application Verification. 1Password associates credential records with applications based on their Bundle ID, each app's unique identifier, and this mapping ensures that credentials are only autofilled into the intended app. However, our analysis reveals that Bundle IDs are not a reliable indicator of application integrity, as they can be *tampered with* by modifying an app's `Info.plist` file or *spoofed* by changing another app's Bundle ID to match that of a legitimate application. We experimentally found that 1Password rejects autofill when conflicting Bundle IDs are detected. Additionally, we confirmed the specific conditions enforced by its verification mechanism. Based on our experimental insights, we have reconstructed the autofill verification workflow employed by 1Password, which we summarize in Algorithm 1.

Algorithm 1: 1Password Autofill Workflow on macOS

Input: **A**: Target Application, **BID**: Bundle ID
if *No records linked to* **BID** **then**
  Select a record to link
if *Multiple active apps are linked to* **BID** **then**
  Abort autofill
if *A is devoid of Code Signature* **then**
  Abort autofill
Link the selected record with **BID** if not yet linked
if *A's Code Signature is untrusted* ***or*** *invalid* **then**
  Abort autofill
$DevID \leftarrow$ Apple Developer ID associated with Code Signature
if $DevID \in$ *Verified Developers* **then**
  if $BID \notin$ *Verified Bundle IDs for* $DevID$ **then**
    Abort autofill
else if $BID \in$ *Existing Developer ID — Bundle ID associations* **then**
  Abort autofill
Perform autofill

The process begins by identifying the currently running application and extracting its Bundle ID – if no records are linked, the user is prompted to select a record for future autofill actions. 1Password also performs checks to detect whether multiple running applications are sharing the same Bundle ID. If such a conflict is detected, the autofill process is aborted and the ID is flagged as potentially spoofed. Subsequently, 1Password verifies the application's code signature by access-

ing the designated folder in its contents. If the application is not digitally signed, the autofill process is aborted. Otherwise, it checks the code signature to ensure it is both (a) `valid`, confirming that the code has not been tampered with, and (b) `trusted`, indicating that it is signed by a verified Apple Developer. In the final step, 1Password checks the Apple Developer ID associated with the code signature, and ensures that the Bundle ID matches a verified list of IDs for that developer. For popular applications (e.g., Discord, Slack), 1Password maintains a list of known Developer IDs to manage verification effectively. When encountering a new unrecognized Bundle ID, it treats the first valid and trusted signature as legitimate and prompts the user to confirm the association. Once this verification is complete, future autofill requests from the same Developer ID are processed without additional prompts. When all checks are successfully passed, 1Password proceeds with the autofill operation. The multitude of safeguards in place strongly suggest that local deceptive apps are a valid concern for desktop password managers, and also reveal the numerous macOS features that can be leveraged to enhance their robustness.

User Confirmation. The first time a user attempts to autofill credentials into an app, 1Password displays an alert window prompting the user to confirm the action (see Figure 3 in the Appendix). Upon confirmation, 1Password establishes an association between the credentials and the app, blocking any future attempts to modify this existing association.

Experimental findings. Here, we detail a series of empirical experimental findings, which constitute the individual "ingredients" that will then be combined into our proof-of-concept attack. While 1Password conducts multiple verification checks before performing an autofill operation, we have identified omissions in their verification process. Accordingly, we have developed an automated phishing application exploiting those weaknesses, specifically in the *Quick Access* feature.

Application Validation. In 1Password's primary autofill workflow, the password manager verifies the identity of the target application before injecting credentials, establishing a binding between the credential item and its intended destination. In contrast, the Quick Access interface does not perform application-level validation; it does not verify whether the application receiving the credentials matches the expected origin (e.g., bundle identifier). This omission allows any local application to retrieve stored credentials without triggering user alerts or verification prompts. The absence of target validation in this component weakens the security guarantees provided by the primary autofill mechanism.

Interaction Origin. 1Password does not differentiate between physical and synthetic keyboard inputs. This allows an attacker to programmatically simulate user interactions, such as key presses, to trigger the Quick Access window and access stored credentials. The attack leverages crafted interactions with the 1Password vault and automatically harvests credentials without additional user input, thereby bypassing Quick Access's built-in security measures.

Window management. The Quick Access window is designed to remain in the foreground, appearing above all other active windows to ensure that it remains accessible and visible to the user. However, macOS allows developers to assign *different* window levels to control the visibility and layering of app windows [75], [76]. By default, windows are assigned the `normal` level, but 1Password elevates the Quick Access window to the `pop-up-menu` level. Our attack exploits this capability by elevating the attacker's window to the *screen-saver* level, to conceal the Quick Access interface from the user while harvesting credentials in the background.

Hidden Text Fields. Hidden text fields are HTML input elements with their visibility intentionally concealed from the user interface, typically through CSS rules (e.g., `display:none`). Although not visible to users, these fields remain accessible to scripts, allowing applications to access, store, and transmit data. As such, attackers can leverage them to exfiltrate credentials within an application, without users being made suspicious or alerted [77].

Attack execution. To assess the practical implications of 1Password's security flaws, we developed a proof-of-concept phishing application that systematically extracts stored credentials while evading detection, by combining all of our aforementioned findings. The attack targets the Quick Access feature, leveraging window management inconsistencies, synthetic user interactions, and distraction techniques to stealthily exfiltrate credentials. The application automates the triggering of Quick Access, bypassing user interaction requirements by simulating keyboard input events. Once activated, it retrieves credential records by programmatically searching stored entries and extracting usernames, passwords, and two-factor authentication (TOTP) tokens. Unlike default autofill, Quick Access does not validate the requesting application, allowing credentials to be retrieved without enforcing application verification. To conceal the exfiltration process, the attack exploits window layering mechanisms, by dynamically elevating the attacker's window *above* the Quick Access interface, it ensures that credential retrieval remains hidden from the user. Additionally, the extracted credentials are stored in the phishing application's hidden input fields, not visible by the user. After credential exfiltration, the application restores its window to the default window level to minimize post-execution traces. This harvesting approach can be extended to exfiltrate stored payment information from the vault. The phishing application enumerates vault entries and accesses items labeled with payment-related keywords (e.g., Visa, Card). Autofill proceeds if the target window includes the expected form fields (e.g., card number, expiration date). However, the availability of sensitive fields depends on the user's stored data and settings (e.g., CVV autofill requires explicitly storing it [78]). Once populated, payment information becomes accessible to the attacker. Finally, to enhance stealthiness, the application introduces obfuscation mechanisms during execution, and overlays distraction content, such as a video player UI, to divert user attention. This mechanism can be further augmented with additional realistic content, such as a simulated software installation or documentation text, to maintain user engagement.

Attack Performance. To evaluate the performance of our attack, we populated 1Password with five credential records corresponding to different services and conducted 10 iterations targeting different sets each time. On average, credential extraction requires 11 seconds per username-password pair. We further extend our evaluation to include two-factor authentication (2FA) codes, by populating 1Password with three records containing Time-Based One-Time Passwords (TOTPs). We then measured the time required to harvest a record comprising a username, password, and TOTP. Our results show that harvesting a TOTP adds approximately four seconds to the attack execution, resulting in a total time of 15 seconds for extracting a complete credential record. Since TOTPs typically remain valid for at least 30 seconds [79], attackers have sufficient time to leverage the stolen credentials before expiration. Overall, our evaluation demonstrates that the proposed attack is practical, highly stealthy, and effective at harvesting credentials from 1Password.

Credential Binding Flaw. We identified an additional design flaw in 1Password's credential association logic that enables a denial-of-service attack that affects legitimate applications. Although this issue does not allow credential harvesting, it can prevent users from accessing their vault. We describe this flaw in detail in Appendix C.

Privilege Escalation: Harvesting System Password. 1Password supports storing system credentials, which enables users to autofill directly into terminal apps, such as the default `Terminal`. In 1Password, system credentials are saved as standard vault entries and are assigned a record name based on the device's name by default. This naming convention introduces a security risk since attackers can predict record names and directly locate system credentials in the vault. For example, a user named "Alice" using a MacBook Pro would have their system password saved under a record named `Alice's MacBook Pro`. Since device names often contain personally identifiable information, including usernames, this feature significantly reduces the effort required for an attacker to identify and extract system credentials.

Attack Evaluation. This attack is a variation of the previously introduced credential-harvesting attack and relies on the same configuration and permissions. The attacker application first retrieves the system's device name and logged-in user information using standard macOS APIs, such as `scutil --get ComputerName` and `whoami`. Using this information, it constructs the expected system credential record name. Subsequently, it queries the 1Password vault through the Quick Access feature and searches for records matching the device name. Since the naming scheme is predictable, the attacker can directly identify system credentials without the need for exhaustive searching. After locating the system credential record, the attacker executes the aforementioned credential exfiltration technique. With access to the system password, the attacker can execute privileged commands using `sudo`, and disable system security features, modify configurations, or install persistence mechanisms.

To further assess the attack's performance, we populated 1Password with system credential records under different configurations. In ten iterations targeting five different user profiles, the attack required an average of 6 seconds to locate and extract system credentials. The attack was highly accurate in detecting the default naming conventions and the associated credentials. This highlights the practicality and stealthiness of the attack, allowing the attacker to escalate privileges once they gain access to the system's credentials.

1Password Command Line Interface. A key feature of 1Password is its Command Line Interface (CLI) tool, accessible through the `op` command, which allows users to interact with their vault directly from the terminal [80]. Primarily designed to automate vault operations and manage credentials programmatically, the CLI tool provides flexibility for advanced users but can also introduce new security risks if improperly secured. Our analysis revealed major flaws in the 1Password CLI tool that can be exploited to extract sensitive credentials, including the `Secret Key` — a critical component required to decrypt the vault. Unlike standard credentials, the Secret Key is not transmitted to 1Password's servers and must be used alongside the master password to unlock the vault [81]. If both the master password and Secret Key are compromised, an attacker can authenticate on a different device, gaining full access to the user's 1Password account and resulting in a total compromise of the user's data.

Authorization Bypass. When a process triggers the 1Password CLI tool for the first time, a pop-up window is shown to confirm access to the vault. However, this authorization pop-up inherits the same flaws as the *Quick Access* window. Specifically, it is assigned the `pop-up-menu` window level, allowing it to be hidden by higher-level windows. Critically, it *does not* differentiate between physical and synthetic user interactions, allowing an attacker to programmatically simulate user inputs and approve CLI access *without* the user's consent. By exploiting this flaw, an attacker can stealthily enable CLI functionality and gain persistent access to the vault.

Default User Records. Once authorized, the CLI provides access to the vault, including a record named `1Password Account`, which is automatically created during account setup. This record stores both the master password and the Secret Key, retrieved in plaintext upon access. While 1Password enforces restrictions on retrieving the Secret Key through the Quick Access window, these restrictions do not extend to the CLI tool, which allows programmatic extraction of the Secret Key through command-line queries. The ability to extract the Secret Key has severe implications for the user, as it allows an attacker to recreate the user's vault on a separate device and completely decrypt its contents by bypassing 1Password's authorization mechanisms. Essentially, this enables full account impersonation, bypasses recovery protections, and grants persistent access—even if the master password is changed.

Attack Execution & Evaluation. To exploit these vulnerabilities, an attack begins by executing the `op` command to trigger the CLI authorization prompt. Since the prompt is not protected against synthetic interactions, the attacker

programmatically approves the request without requiring user intervention. After authorization is granted, the attack retrieves the record ID associated with the user's account by executing the op item list command, followed by `op get item <recordID> --reveal`, which exposes the Secret Key and master password. The attack completes in approximately three seconds, with two seconds required for the authorization bypass and less than one second for the credential extraction. The efficiency and low interaction overhead render this attack highly practical for stealthy credential exfiltration.

B. Keeper

Features and Protections. Keeper is a popular password manager that provides autofill functionality for login credentials in macOS applications. Keeper verifies applications using the `Bundle Display Name` rather than an immutable identifier, such as the Bundle ID. This introduces a critical vulnerability, as an attacker can spoof the Bundle Display Name to bypass authentication checks. Notably, modifying the Bundle Display Name requires neither elevated privileges nor OS-level permissions. To initiate autofill, Keeper provides keyboard shortcuts for different credential types, such as usernames, passwords, payment info, and one-time passwords (OTPs), allowing users to fill these fields individually [82]. While this approach improves usability, it also constitutes an attack vector since a malicious application can programmatically simulate the required keyboard inputs to exploit the autofill process and harvest credentials without user consent.

Bypassing Autofill Protections. To evaluate Keeper's autofill functionality, we analyzed its reliance on an application's Bundle Display Name for credential autofill validation. We developed a malicious macOS application that mimicked a legitimate one by modifying its Bundle Display Name to match a trusted target, and tested whether Keeper would autofill credentials into the spoofed application's text fields. Our analysis confirmed that the malicious application successfully triggered the autofill functionality, populating the fields with credentials associated with the legitimate target. Unlike 1Password, which implements stricter validation, Keeper's reliance on a modifiable identifier allows unauthorized access to stored credentials with minimal effort, extending to other sensitive record types, such as payment information.

Stealthiness. While the baseline attack is technically effective, its practicality for undetected execution is impacted by visual artifacts during the autofill process. Text fields remain visible, the malicious application's window briefly flashes on the screen, and its icon momentarily appears in the Dock during system restarts, increasing the risk of detection. To address these issues, we developed an auxiliary application, which we call `Hider`, designed to mask the malicious app's activity and enhance the attack's stealthiness (we provide a detailed description in Appendix B). The auxiliary application operates in the foreground to maintain user engagement, while the malicious application executes exclusively in the background to perform the attack. Also, to achieve stealthier deployment, we deploy the malicious app within the auxiliary application's

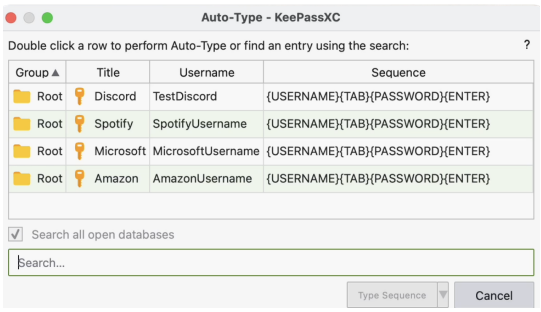

Fig. 2: KeePassXC pop-up window.

`Contents` folder. To further minimize detection, `Hider` leverages techniques from our 1Password attack, setting its window level to `screen-saver` to remain above all other windows. Moreover, the window frame is omitted to prevent visual artifacts, such as blurring when focus shifts between applications. Additionally, the malicious app's icon is omitted from the Dock, using Electron's `app.dock.hide()` API [83], effectively hiding itself from the user.

Attack Evaluation. The attacker configures a JSON file listing target applications, with each entry containing the Bundle Display Name of a target. During initialization, the malicious application adopts the first target's Bundle Display Name from the list. Once active, it simulates key presses to trigger Keeper's autofill and extract the credentials associated with the current Bundle Display Name. After harvesting, the application modifies its Bundle Display Name to the next target in the list and restarts, iterating through all specified targets. Our analysis indicates that the phishing application requires an average of three seconds to retrieve a pair of credentials per target, with an additional one second to restart and update its identity. This results in a total execution time of four seconds per target, demonstrating the attack's efficiency and feasibility for large-scale credential harvesting.

Limitations. Despite the aforementioned strategies to enhance the attack's stealth, certain visual indicators may still reveal its presence to the users. First, macOS automatically hides the mouse pointer when a text field is focused using the `Tab` key. Since the malicious application relies on this behavior to trigger autofill, users may notice the pointer disappearing at regular intervals. Second, macOS adjusts a window's shadow opacity when it is inactive. Although the auxiliary application remains on top level, it cannot maintain continuous focus, leading to minor visual effects that users might detect. Finally, while the malicious application's icon is hidden from the Dock to avoid detection, brief flashes or momentary appearances may still occur during application restarts due to macOS's built-in app launching behavior.

C. KeePassXC

Features. Next, we focus our analysis on KeePassXC, another widely used password manager, to evaluate its autofill mechanism and security flaws. KeePassXC leverages the `AutoType` feature and supports usernames, passwords, and one-time passwords (OTPs). However, OTP autofill is

not enabled by default and requires users to configure each record individually [84]. Also, it disables autofill by default, requiring users to manually enable the feature by configuring a keyboard shortcut in the settings. Additionally, users are required to manually associate credentials with specific applications, similar to Keeper (§IV-B) and MacPass (§IV-D). This association relies on the application's Bundle Display Name rather than an immutable identifier like the Bundle ID, making it vulnerable to manipulation. Once configured, the autofill process is triggered via a user-defined keyboard shortcut and requires manual confirmation through a pop-up prompt.

Verification Mechanisms. Similar to Keeper, KeePassXC leverages the app's Bundle Display Name to match and autofill credentials without additional validation. However, if the requesting application's window is positioned `above` KeePassXC's window (i.e., has a higher window level), the `AutoType` feature disables autofill and instead displays a manual credential selection popup, effectively blocking automatic credential injection. Instead of completing the autofill, KeePassXC displays a pop-up window (Figure 2) listing all available vault credentials where the user must manually select the desired credentials before autofill is completed. Finally, KeePassXC does not provide a default autofill shortcut, requiring attackers to identify the user's specific keyboard configuration before launching an attack.

AutoType Exploitation. To detect the keyboard shortcut configured for the `AutoType` feature, we implemented a brute-force technique that targets commonly used shortcuts [85], [86]. The malicious application emulates keyboard combinations while monitoring for focus changes, which indicate that a shortcut has successfully triggered the autofill. Once the appropriate shortcut is identified, the attack proceeds by automating the credential extraction process. The malicious application navigates the pop-up list using synthetic key presses to select entries and confirms the autofill operation. This process is systematically repeated until all stored credentials are extracted. The primary overhead in this attack lies in inferring the user-configured shortcut for Autotype. To evaluate this process in detail, we conducted a dedicated performance analysis measuring the time required to exhaust each phase of our shortcut inference strategy (we provide more details in Appendix E). Testing the most common default shortcuts requires approximately ten seconds. Once the correct shortcut is triggered, credential extraction proceeds at an average rate of seven seconds per credential pair. These results demonstrate that the attack is both practical and scalable under realistic conditions for targeting a user's account credentials for a set of popular services.

D. MacPass

Next, we analyze MacPass and explore its internal verification flaws. Similar to KeePassXC, MacPass adopts the AutoType feature for credential injection. However, AutoType is enabled by default and can be triggered using a predefined keyboard shortcut [57]. MacPass employs a unique credential association method, linking credentials to an application's window title (i.e., the label displayed on the window frame) instead of using immutable identifiers. When an application with a matching window title triggers the autofill, MacPass submits the associated credentials directly into the application. This linking mechanism introduces a critical verification flaw, making MacPass vulnerable to credential exfiltration.

Window Title Spoofing. In this attack variation, the primary requirement for the attacker is to modify the malicious application's window title to impersonate a trusted application—a straightforward task that requires no elevated permissions. To enhance stealthiness, we omit the window frame entirely from the malicious application, preventing the window title from being displayed to the user. Furthermore, an attacker can also recreate the window frame through custom UI design, making the phishing application visually indistinguishable from the legitimate target.

Attack Evaluation. The attack follows a series of coordinated steps to effectively harvest the credentials. Initially, the malicious application dynamically modifies its window title to match a predefined list of target applications – once MacPass detects the spoofed title, the application triggers autofill using the default keyboard shortcut, capturing and storing the credentials. This process repeats iteratively, with the malicious application updating its window title for each subsequent target in the list. Our evaluation finds that each cycle—including modifying the window title, triggering the `AutoType` feature, and capturing a pair of credentials—requires approximately five seconds. This efficiency allows attackers to harvest multiple credentials while bypassing any need for user interaction.

E. Alternative Local Credential-Stealing Attack Vectors

To assess the broader scope and impact of our attacks, we compare them to a traditional local attack vector for stealing passwords: keyloggers, which monitor user inputs to capture credentials, have limited effectiveness against password managers as credentials are autofilled or typed programmatically. Importantly, the macOS Secure Input Mode adds another layer of protection by blocking keyloggers from accessing keyboard events when sensitive fields, such as password inputs, are in focus [87]. Moreover, the Secure Keyboard Entry in the macOS Terminal provides protection against keylogging by preventing other applications from intercepting keystrokes; however, this feature is not enabled by default [88]. Nonetheless, enabling Secure Keyboard Entry reduces the risk of unauthorized access to sensitive inputs, such as `sudo` passwords, but may also introduce compatibility issues with legitimate tools that rely on keyboard input interception [89]. In summary, keyloggers have limited effectiveness on macOS, while our attacks are capable of exfiltrating critical user data from password managers.

V. Windows Password Managers

We shift our focus to the Windows platform, analyze how the selected password managers implement their functionality, and identify vulnerabilities that enable credential exfiltration.

Security Mechanisms & Features. Contrary to macOS, Windows does not enforce strict application verification. Apps

are not required to use immutable identifiers, and code signing is not mandatory or enforced [90]. Instead, password managers rely on mutable properties like window titles for autofill verification. In addition, Windows permits inter-app interactions without elevated privileges, allowing any app to simulate keystrokes, manipulate window focus and position, or change titles. Following the priorly established exploitation methodology for macOS, we demonstrate attacks that systematically extract credentials by exploiting these design flaws.

A. 1Password

On Windows, 1Password exposes its autofill functionality via the Quick Access interface, while it differs from its macOS implementation, where autofill behavior is tied to application identity and verified more strictly. Upon activation, the user selects a vault record, and the corresponding credentials are inserted into the target application. Notably, this process does not create any persistent association between the application and the credentials being filled.

Platform-specific Mechanisms. In contrast to macOS, the Windows version performs no target validation, and our analysis confirms that *any* active window receiving focus is eligible for autofill, regardless of its origin or developer's identity. Moreover, the Quick Access window persists as long as the 1Password background process is running, enabling repeated interactions without user reauthentication. These properties highlight critical security flaws in 1Password's Windows implementation compared to its macOS distribution, regarding permission controls and validation during autofill.

Exploiting AutoType. 1Password's `AutoType` feature fills credentials into any active text field, without verifying the identity of the target app. As a result, credentials can be injected into arbitrary windows under the attacker's control. We replicate our approach for MacPass (§IV-D), automating input events to trigger `AutoType` and direct output into hidden fields within a phishing app.

Stealthiness. Windows does not enforce window layering constraints, and the most recently focused window is always positioned in the foreground. As a result, prior stealthiness techniques that rely on display-level manipulation are ineffective. However, the Quick Access window remains active as long as 1Password's background process is running. Leveraging this persistence, we use the Node.js library `node-window-manager` [91] in our application to reposition the Quick Access interface beyond the visible screen boundaries. Since Windows does not restrict these operations, the interface remains fully functional even when hidden from view. Additionally, 1Password preserves the window's last known position, ensuring all subsequent activations occur off-screen without user awareness.

The attack follows the priorly established methodology (§IV), adapted to the Windows environment. Regarding stealthiness in this attack variant, the malicious application repositions the Quick Access window off-screen at the start of execution, preventing the user from noticing its presence. The absence of security checks in 1Password's Windows implementation allows the attack to execute more efficiently than on macOS. While the macOS attack requires separate interactions to access usernames and passwords sequentially, the Windows variant completes autofill with a single keyboard shortcut. As a result, credential harvesting takes approximately 7 seconds per record, improving performance while maintaining the same level of effectiveness.

B. Keeper

Keeper's autofill functionality on Windows relies on the `AutoType` mechanism where users associate credential entries with application windows based on their titles, and autofill is triggered via a keyboard shortcut [82].

Bypassing Autofill Mechanism. Unlike its macOS counterpart, Keeper associates credentials with window titles rather than application metadata. However, it does not continuously verify titles—instead, validation is triggered only when the application loses and then regains focus. As a result, if the title is modified while the application remains in focus, the change goes undetected. Attackers are able to manipulate window titles and trick Keeper into autofilling credentials in unauthorized applications. For example, a malicious application can dynamically alter its window title to impersonate a trusted target without triggering a re-validation check. Additionally, Windows' permissive security model allows malicious apps to exploit Keeper's autofill mechanism without requiring additional permissions. We leverage these weaknesses to deploy a fully automated phishing attack that triggers autofill by spoofing trusted window titles. The phishing application emulates legitimate targets by modifying its title at runtime, bypassing validation and causing Keeper to inject credentials into unauthorized contexts without user interaction.

Stealthiness. To achieve stealthiness, the phishing application omits its window frame to conceal spoofed titles and avoid visual cues. The attack exploits a flaw in Keeper's window title verification logic, which performs validation only when the application loses and regains focus. At launch time, the phishing application spawns two windows: a visible main window presenting benign content and a secondary, hidden window used for title spoofing. The hidden window is concealed using two techniques: (i) it is configured to be omitted from the system taskbar and (ii) it is programmatically repositioned outside the visible screen area using the `node-window-manager` library [92]. This configuration ensures that the spoofing window remains undetectable, allowing the phishing application to manipulate window titles and trigger autofill without user awareness.

Attack Evaluation. The malicious app maintains a predefined list of target apps and their corresponding window titles. For each target, it dynamically modifies its visible main window's title to mimic a legitimate app. To trigger autofill, the attack simulates a focus-switching sequence, ensuring Keeper registers the new title. The app momentarily shifts focus to the hidden secondary window and then returns focus to the main window, exploiting Keeper's reliance on user-

driven focus changes for title verification. Also, crafted timing delays ensure the spoofed title is detected accurately. Keeper identifies the spoofed title instantly, and the focus-switching process completes in under 1 second. The remaining time is spent autofilling credentials into text fields. Overall, the attack successfully extracts a set of credentials in approximately 3 seconds, demonstrating both efficiency and effectiveness.

C. LastPass & KeepassXC

LastPass. On Windows, LastPass deploys the autofill feature using the `AutoType` mechanism, and users are expected to explicitly associate credentials with target apps. Similar to Keeper, LastPass identifies target windows by their titles, and it additionally monitors title changes in real time, eliminating the need for focus-switching to trigger verification. While this improves autofill responsiveness, it does not address the underlying vulnerability of relying on mutable window titles. Our analysis shows that LastPass exhibits identical vulnerabilities to those previously identified in MacPass (§IV-D). As a result, they are similarly susceptible to credential harvesting via phishing apps. This attack variation executes with comparable efficiency across platforms, with credentials extracted in approximately five seconds.

KeePassXC. KeePassXC on Windows and macOS share the same deployment model, relying on the `AutoType` feature, which is disabled by default and requires user confirmation for autofill (§IV-C). Once enabled, triggering `AutoType` prompts the user before credentials are inserted. However, the underlying implementation differs across platforms, resulting in distinct attack surfaces.

Platform-Specific Mechanisms. On Windows, KeePassXC identifies target apps by their window titles. When a match is found, it displays an authorization prompt requiring user confirmation. However, this behavior can be disabled via a configuration setting. If the confirmation prompt is turned off, KeePassXC automatically fills credentials into any matching window title without user interaction. We identified an additional flaw in KeePassXC configuration storage. Both the confirmation prompt setting and the keyboard shortcut used to trigger `AutoType` are stored in a plaintext configuration file located in the user's app folder[1]. This file is unprotected and can be modified by any app with user-level access. A malicious app may disable the confirmation prompt and assign a known shortcut. Even though these changes require a restart of KeePassXC, this can be triggered programmatically or may occur during regular user activity.

Attack Workflow & Performance. The attack follows the strategy deployed against LastPass and MacPass (§IV-D). The phishing app first modifies the configuration file to disable the prompt and set a predefined shortcut, eliminating the need for user interaction. After restarting KeePassXC, the app triggers `AutoType` and captures the injected credentials. The attack outperforms the brute-force shortcut inference approach required on macOS, since the keyboard shortcut is explicitly configured. File modification completes in under one second, and the total execution time is approximately five seconds, which is comparable to the MacPass attack.

[1] *C:\Users\<user>\AppData\Roaming\KeePassXC*

VI. Linux Password Managers

On Linux, only 1Password and KeePassXC support autofill for native applications, making it significantly more limited than on Windows or macOS. We applied the previously introduced attack methodology, and while the attacks remained effective, platform-specific constraints prevented them from being stealthy. Specifically, Linux does not support display-level distinctions, such as `screen-saver` and `pop-up-menu`, preventing overlay-based concealment of password manager windows. Additionally, Linux enforces an OS-level policy that restricts window positioning within the visible screen area. We further explored alternative methods to manipulate window behavior using tools such as `xdotool` [93]. This approach was ineffective since 1Password's Quick Access window and KeePassXC's authorization prompt remained visible throughout the attack. While KeePassXC remains vulnerable to the same `AutoType`-based credential extraction technique, the attack is noticeable due to the persistent visibility of password manager windows. Due to space constraints, we provide additional details in Appendix D.

VII. Attack Mitigation

Given the significant implications of our attacks, we propose a series of countermeasures that can be incorporated by password managers or enforced at the OS level.

Security checks enhancement. Our analysis of 1Password on macOS uncovered a series of security checks that are employed for ensuring the legitimacy of the application triggering autofill. Surprisingly, the majority of the password managers did not employ the extensive security measures that 1Password does, resulting in highly insecure applications that can be trivially tricked into autofilling user credentials. The first step for fortifying password managers on macOS is to leverage all of the OS-level mechanisms that are readily available. This countermeasure can only be implemented by macOS password managers, as Windows and Linux currently do not expose the required OS-level properties and callbacks. Finally, specifically for 1Password on macOS, we emphasize that the checks performed on the main autofill process should also be employed for the secondary ones that we exploit. In other words, the robustness of the autofill process should not be undermined by additional usability features.

Window placement: z-index level. As mentioned before, macOS distinguishes between `pop-up-menu` and `screen-saver` window levels. As a result, `screen-saver` level windows can hide any focused windows at the `pop-up-menu` level, even the ones meant to always stay at the screen's top level. A straightforward mitigation involves modifying password manager implementations to reposition their pop-up interfaces (e.g., 1Password's Quick Access window and CLI authorization prompt) to the `screen-saver` window

level, rather than the default `pop-up-menu` level. At this elevated level, attacker-controlled windows placed at the same layer will appear behind the password manager interface. Since the system brings to the front the most recently interacted window when autofill is triggered via a keyboard shortcut, this approach ensures that the password manager's interface remains visible to the user.

Window Placement and Screen Boundaries. In Windows, the `pop-up-menu` and `screen-saver` window levels are treated equivalently, preventing the use of window layering for reliably prioritizing sensitive UI elements. Our attack is stealthy since it leverages the `node-window-manager` library to reposition the 1Password Quick Access window beyond the visible screen boundaries. Preventing such off-screen placement at the OS level (e.g., by constraining window coordinates to the physical display area) would significantly limit our attack's stealthiness.

Synthetic Click Detection. Our attacks rely on emulated keyboard presses, making synthetic input detection a critical countermeasure. However, legitimate apps, such as terminal emulators and accessibility tools, also use synthetic input for automation and usability. Password managers should integrate detection mechanisms that differentiate between user-triggered autofill and external manipulation, rather than enforcing uniform OS-level restrictions. For instance, MacPass (§IV-D) relies on synthetic input for autofill, and rejecting synthetic events would break its intended functionality. Instead, password managers should validate the requesting process ID and restrict autofill to explicitly linked applications, preventing unauthorized input injection while preserving legitimate automation. To evaluate this defense, we develop a prototype tool for macOS and Windows that detects synthetic input from running applications and issues security warnings to users.

macOS prototype. We have developed a prototype detection tool as a Swift application. The primary mechanism employed by the detection tool is to listen for Key Pressed events (i.e., when the `CGEventType` instance is set to `.keyDown`) and check if at least one of the following conditions is true: (i) The Process ID of the application that generated the synthetic event is equal to 0, which indicates that the event originated at the system level. (ii) The User ID associated with the process is equal to 0, i.e., the root. (iii) The User ID associated with the process is equal to 244, i.e., the specific system user account associated with input device handling (e.g., keyboard).

If any of the conditions is satisfied, it indicates that the user has physically pressed the detected key; otherwise, a potentially untrusted application has triggered the interaction. In the latter case, our defense mechanism triggers a visual alert and notifies the user. However, in a production deployment, this signal can be leveraged to enforce stricter mitigations, such as immediately locking the vault and requiring re-authentication (e.g., master password, biometrics, OS credential) to prevent unauthorized autofill. Our prototype interface is implemented as a system-level window to ensure it is always rendered above all other windows, including those manually elevated to the `screen-saver` level. While system-level windows offer limited customization, they benefit from privileged rendering behavior enforced by the OS.

Windows prototype. We developed our defensive tool as a C++ application. The core mechanism intercepts keyboard events at a low level and analyzes them to determine their origin. The tool utilizes a low-level keyboard hook procedure by calling the `SetWindowsHookEx` function with the `WH_KEYBOARD_LL` hook type [94]. This allows the application to monitor all keyboard input events at the system level before they reach any applications. The hook procedure (`LowLevelKeyboardProc`) processes each keyboard event encapsulated in the `KBDLLHOOKSTRUCT` structure. We focus on events where the `nCode` parameter equals `HC_ACTION`, indicating a keyboard event that should be processed. Within this procedure, we verify the flags field of the `KBDLLHOOKSTRUCT` to determine if the event was generated synthetically. Specifically, we access the `LLKHF_INJECTED` flag, which signifies that the event was injected by an application rather than originating from a physical keyboard press. If such a synthetic event is detected, our tool launches an alert window to warn the user about the synthetic keyboard input. We designed the alert window to be displayed at the topmost level of the screen by using the `MessageBox` function with the `MB_TOPMOST` flag. This ensures that the alert window stays above all other windows, including those set to the topmost level by other applications.

VIII. Discussion

Disclosures. We have responsibly disclosed our findings to the developers of all the evaluated password managers, all of whom have acknowledged our reports. Keeper and MacPass found our suggestions especially helpful and are actively working on implementing fixes. Keeper further acknowledged our efforts with a bug bounty reward. 1Password stated that while they want to deploy countermeasures, they are "really restricted by the platform as to what defences we can deploy to defend against attacks such as this." We will continue to work with all vendors upon request, sharing countermeasures and assisting them in navigating OS limitations to improve security.

Stealthiness. Our attack employs various concealment techniques to eliminate visible execution traces. The majority are stealthy, with two exceptions: Keeper on macOS, which exhibits minor visual artifacts, and Linux, which is limited by internal UI constraints. On macOS, window layering, Dock suppression, and visual overlays effectively eliminate all attack indicators. On Windows, the absence of layering constraints allows the application's windows to be repositioned off-screen without triggering visual alerts. Video demonstrations of all the attack variants are available on our artifact page [28].

Real-world Applicability. To evaluate the scalability of our attacks, we include details about the performance of each attack variant to highlight the limited time each of our exploits takes to complete exfiltration. We include performance values for the primary attack variant against each password manager in Table II. Additionally, our attacks on macOS rely on users

enabling the Accessibility permissions for our malicious app, which we found to be a common practice and hence, a reasonable assumption, since 20% of the 100 most popular free apps on the App Store request this permission (see §IV.)

Additional Credential Mechanisms. Our attack targets traditional autofill mechanisms in standalone password managers. We also evaluated whether similar techniques could be applied to OS-level credential stores and modern authentication alternatives. Specifically, macOS Keychain Access [95] does not expose autofill capabilities for native applications, and credential retrieval typically requires explicit user interaction via system dialogs, which falls outside our threat model. Similarly, passkey-based authentication [96], [97], increasingly supported by password managers, relies on challenge-response protocols (e.g., WebAuthn [98]) and bypasses form-based credential injection. Since they do not rely on typical input field submission, and password managers interface with browser or app APIs, our credential harvesting attacks do not directly apply to passkeys. We leave a systematic evaluation of passkey-specific threat models and additional credential management mechanisms to future work.

IX. Related Work

UI Attacks. Early works addressing secure UI proposed isolated window systems via solutions like EROS [103] and TrustedX [104]. Recent work on UI security has focused on mobile platforms [105], [106], [107], [101], [108], [109], [110]. Bianchi et al. [106] exploited Android's full-screen overlay to trick users into interacting with phishing apps that mimic the GUI of legitimate apps. Fratantonio et al. [101] demonstrated users' susceptibility to granting permissions that could enable malicious apps to take over the UI feedback loop. Lee et al. [107] found iOS apps that use stealthy and malicious *crowdturfing* UIs to manipulate app ranking. Researchers have also studied UI attacks on non-traditional display settings. Mahdad et al. [111] used deceptive overlays on limited-display FIDO2 authenticators that trick users into authorizing malicious sign-in requests. Cheng et al. [112] demonstrated UI attacks on popular AR platforms from Apple, Google, Meta, Microsoft, and within the WebXR API in the browser.

Desktop Apps. Our threat model exploits password managers using a malicious desktop app. Prior work has shown several vulnerabilities in desktop apps; Xiao et al. [113] demonstrated that vulnerable cross-context flows can allow attackers to perform Remote Code Execution (RCE) and take over an operating system. Ali et al. [114] showed that popular Electron apps may have insecurities that can lead to RCEs, and Paloscia et al. [115] explored how migrating web application code to desktop apps suffers from inherent flaws due to the differences that exist between the two different execution environments. Jin et al. [116] demonstrated an exploit within Microsoft Teams and developed a DOM-based defense mechanism. Ahmadpanah et al. [117] found vulnerabilities in local deployments of trigger-action platforms, e.g., Node-RED, that execute code across multiple applications. Finally, Wang et al. [118] exploited lax privilege checks on Windows to launch cross-platform attacks from a desktop against mobile super apps like WeChat.

Password Managers. In 2012, Bonneau et al. [119] found that password managers are not resilient to internal observation, i.e., an attacker can impersonate a user by intercepting from inside the user's device. Li et al. [120] in 2014 and Oesch and Ruoti [121] in 2020 found that web-based password managers use insecure defaults and include unencrypted metadata, which makes them vulnerable to credential stealing and clickjacking attacks. Fabrega et al. [122] used injection attacks against password managers that allowed attackers to extract confidential information from observing protected data. Finally, studies have also demonstrated vulnerabilities in password managers' autofill implementations.

A. Comparison with Prior Work

Next, we provide a detailed comparison between our work and prior research on credential-stealing attacks against password managers, which we summarize in Table III.

Silver et al. [99] evaluated desktop and mobile password managers' functionality within web browsers. Their exploits can be largely categorized based on attacker placement — first, a malicious website carefully crafts a web page to extract information from autofilled values; and second, a malicious attacker intercepts traffic when on the same network as the victim (e.g., public Wi-Fi in a coffee shop). While the attacks require initial user interaction to trigger the attack (e.g., the user interacts with the website or joins a new Wi-Fi network), subsequently the attack can be executed in a stealthy manner, by triggering password managers to autofill hidden iframes or browser windows.

Lin et al. [77] demonstrated vulnerabilities in password managers' handling of autofill on websites. They developed curated web forms that remain hidden from the victim, with the help of CSS attributes, overlays, and off-screen placement. When the victim visits and interacts with the web page, password managers autofill hidden form fields, enabling stealthy credential exfiltration. **Fu and Wang [100]** extended prior work by exploring password managers' handling of hidden `<input>` elements when autofilling forms on sites within web browsers.

Fratantonio et al. [101] demonstrated vulnerabilities in Android's implementation of Accessibility permissions. While they did not explicitly demonstrate attacks against password managers, the same techniques the paper demonstrated could be used to access the vault of a password manager installed on the mobile device. Gangwal et al. [102] exploited the WebView functionality used to load web pages within Android apps. They demonstrated that while an app may load an external login page within a WebView, an autofill attempt on this WebView can leak credentials to the underlying app. Their work demonstrated shortcomings in the autofill design of mobile password managers and Android's system-level intermediation for autofill.

This work. Native password managers expose a fundamentally different attack surface compared to their browser-

TABLE III: Comparison with prior work on attacks against password managers. Here, we consider **Device** (🖥 = Desktop, 📱 = Mobile) evaluated in the study, the attacker's **Placement** in regard to the victim (🖥 / 📱 = on-device, 🌐 = in a web browser, 🖧 = on-network), whether the victim needs to interact with the vulnerable platform (i.e., app or web browser) to **Trigger** the attack (👤✓ = Requires UI, 👤✗ = No UI), whether the attacks are **Stealthy** (🙈 = Stealthy, 👁 = Not stealthy), and the **Platform** (🌐 = web browser, 📱 = mobile phone, 🖥 = desktop), along with the corresponding exploited **Feature**.

Reference	Device	Attacker Placement	Victim		Vulnerability	
			UI Trigger	Stealthiness	Platform	Feature
Silver et al. [99]	🖥 📱	🌐 🖧	👤✓	🙈	🌐	Hidden iframes and windows
Lin et al. [77]	🖥	🌐	👤✓	🙈	🌐	Hidden form fields
Fu and Wang [100]	🖥	🌐	👤✓	🙈	🌐	Hidden `<input>` fields
Fratantonio et al. [101]	📱	📱	👤✗	🙈	📱	Android's Permission Model
Gangwal et al. [102]	📱	📱	👤✓	👁	📱	Android's Autofill Intermediation
Vault Raider (this work)	🖥	🖥	👤✗	🙈	🖥	Native Desktop Autofill

based or mobile counterparts. Prior work has focused on those environments by exploiting form heuristics, insecure inter-app communication, or overlay-based UI attacks [77], [102], [100], [101]. Their attacks operate within different security contexts and constraints, such as the browser sandbox or Android's permission architecture. Also, prior threat models assume additional conditions, such as user interaction or compromised origins. In contrast, our attack targets standalone desktop password managers, for which completely different safeguards exist (e.g., see §IV for macOS-specific mechanisms). We exploit intended autofill-related functionality and the absence of robust application identity verification to harvest credentials without user interaction. Our attack leverages native OS features and verification inconsistencies, exposing a novel and previously unexplored threat vector. Nonetheless, our attack also incorporates hidden form elements and manipulates UI-related features to achieve stealthiness, and UI-based deception has been used against autofill [77]. Importantly, we demonstrate attacks that target the credentials of a password manager's vault itself. Overall, we highlight features that remain unavailable in web browsers and mobile apps, and therefore uncover vulnerabilities left unexplored by prior research.

Despite extensive research into password managers' usability and security, their analysis has been limited to mobile and browser-based platforms. Our work highlights unique challenges that password managers face on desktop platforms, which have variable security models that make them susceptible to UI-based attacks. We hope that our work inspires further research on the nuances of desktop-based deployments and incentivizes operating systems to standardize and secure native autofill functionality similar to mobile platforms.

X. Conclusions

While years of research have identified barriers to password managers' adoption, developers have made notable progress in addressing these issues by introducing desktop applications and features like *universal autofill* and *quick access*. However, these advancements also introduce new attack vectors. In this work, we highlight how desktop operating systems' unique architectures, features, and constraints can augment or undermine the security mechanisms of password managers. We demonstrate a range of UI-based attacks that compromise sensitive user credentials, financial information, and even bypass 2FA. Our work examines multiple popular password managers across major operating systems, uncovering critical security vulnerabilities that enable compromising users' online accounts. To mitigate these risks, we have designed straightforward techniques for detecting synthetic user input events and preventing our attacks. Accordingly, we have disclosed our findings and mitigations to the respective developers, encouraging them to implement the necessary safeguards. Our work advances ongoing efforts of enhancing the security of password managers and ensuring that they remain a trustworthy tool for users.

Ethical Considerations

All experiments were conducted using desktop environments, configurations, and applications under our control. Password manager evaluations relied on synthetic credentials and test accounts created for this study. No real user data, accounts, or devices were used at any stage, and our attacks did not affect any actual users. Moreover, as detailed in §VIII, we have notified all of the affected password manager vendors of our findings and our proposed mitigation techniques.

Acknowledgements

We thank the anonymous reviewers and shepherd for their helpful feedback. This project was supported by the National Science Foundation (CNS-2211574, CNS-2143363). The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF.

References

[1] M. Ghasemisharif, C. Kanich, and J. Polakis, "Towards automated auditing for account and session management flaws in single sign-on deployments," in *2022 IEEE Symposium on Security and Privacy (SP)*.

[2] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[3] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, and J. Polakis, "O single {Sign-Off}, where art thou? an empirical analysis of single {Sign-On} account hijacking and session management on the web," in *27th USENIX security symposium (USENIX Security '18)*.

[4] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: Http cookie hijacking and the exposure of private information," in *2016 IEEE symposium on security and privacy (SP)*.

[5] L. Lassak, E. Pan, B. Ur, and M. Golla, "Why aren't we using passkeys? obstacles companies face deploying FIDO2 passwordless authentication," in *USENIX Security '24*.

[6] L. Lassak, A. Hildebrandt, M. Golla, and B. Ur, ""it's stored, hopefully, on an encrypted server'': Mitigating users' misconceptions about FIDO2 biometric WebAuthn," in *USENIX Security '21*.

[7] C. Herley and P. Van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Security & privacy*, vol. 10, no. 1, pp. 28–36, 2011.

[8] J. Blessing, D. Hugenroth, R. J. Anderson, and A. R. Beresford, "Sok: Web authentication in the age of end-to-end encryption," 2024. [Online]. Available: https://arxiv.org/abs/2406.18226

[9] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman, "Of passwords and people: measuring the effect of password-composition policies," in *ACM CHI '11*.

[10] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *2012 IEEE S&P*.

[11] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *ACM CCS '16*.

[12] J. Bonneau, E. Bursztein, I. Caron, R. Jackson, and M. Williamson, "Secrets, lies, and account recovery: Lessons from the use of personal knowledge questions at google," in *WWW '15*.

[13] D. Florêncio, C. Herley, and P. C. van Oorschot, "Password portfolios and the Finite-Effort user: Sustainably managing large numbers of accounts," in *USENIX Security '14*.

[14] D. Florencio and C. Herley, "A large-scale study of web password habits," in *WWW '07*.

[15] A. Nisenoff, M. Golla, M. Wei, J. Hainline, H. Szymanek, A. Braun, A. Hildebrandt, B. Christensen, D. Langenberg, and B. Ur, "A {Two-Decade} retrospective analysis of a university's vulnerability to attacks exploiting reused passwords," in *USENIX Security '23*.

[16] D. Florêncio, C. Herley, and B. Coskun, "Do strong web passwords accomplish anything?" *HotSec*, vol. 7, no. 6, p. 159, 2007.

[17] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki *et al.*, "Data breaches, phishing, or malware? understanding the risks of stolen credentials," in *Proceedings of the 2017 ACM CCS*.

[18] J. A. Halderman, B. Waters, and E. W. Felten, "A convenient method for securely managing passwords," in *WWW '05*.

[19] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions." in *USENIX Security '05*.

[20] S. Chiasson, P. C. van Oorschot, and R. Biddle, "A usability study and critique of two password managers." in *USENIX Security '06*.

[21] A. Karole, N. Saxena, and N. Christin, "A comparative usability evaluation of traditional password managers," in *Information Security and Cryptology-ICISC 2010: 13th International Conference*.

[22] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: design, implementation, and usability evaluation of a password manager," ser. ACSAC '12.

[23] 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)*, 2019.

[24] "Mdn web docs - url," https://developer.mozilla.org/en-US/docs/Web/API/URL, 2024.

[25] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh *et al.*, "Protecting accounts from credential stuffing with password breach alerting," in *USENIX Security '19*.

[26] J. Nejati, M. Luo, N. Nikiforakis, and A. Balasubramanian, "Need for mobile speed: A historical analysis of mobile web performance," 2020.

[27] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, "Slimium: debloating the chromium browser with feature subsetting," in *ACM CCS '20*.

[28] A. Infantino, M. M. Ali, K. Solomos, and J. Polakis, "[Artifact] Vault Raider: Stealthy UI-based Attacks Against Password Managers in Desktop Environments," 2026. [Online]. Available: https://doi.org/10.5281/zenodo.16996391

[29] web.dev, "Autofill in forms," https://web.dev/learn/forms/autofill/, 2023.

[30] T. Verge, "Google chrome password manager integrates with android autofill," https://www.theverge.com/2024/10/18/24273369/google-chrome-android-password-manager-native-autofill, 2024.

[31] M. Foundation, "Firefox sync features," https://www.mozilla.org/en-US/firefox/features/sync/, 2024.

[32] A. Inc., "Set up icloud keychain to autofill information on mac," https://support.apple.com/guide/mac-help/set-icloud-keychain-autofill-information-mac-mh43699/mac., 2024.

[33] M. W. Docs, "Html attribute: autocomplete," https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/autocomplete, 2024.

[34] TechTarget, "Password manager," https://www.techtarget.com/searchsecurity/definition/password-manager, 2024.

[35] 1Password, "1Password: Password Manager for Families, Businesses, Teams," 2024, accessed: 2024-11-06. [Online]. Available: https://1password.com/

[36] LastPass, Inc., "Lastpass," https://www.lastpass.com/, 2024, accessed: 2024-11-06.

[37] Keeper Security, Inc., "Keeper security," https://www.keepersecurity.com/, 2024, accessed: 2024-11-06.

[38] Wikipedia contributors, "Multi-factor authentication — Wikipedia, the free encyclopedia," 2024, [Online; accessed 6-November-2024]. [Online]. Available: https://en.wikipedia.org/wiki/Multi-factor_authentication

[39] 1Password, "Zero-knowledge encryption in 1password," https://1password.com/features/zero-knowledge-encryption/, 2024.

[40] C. Topcuoglu, A. Martinez, A. Acar, S. Uluagac, and E. Kirda, "Macos versus microsoft windows: A study on the cybersecurity and privacy user perception of two popular operating systems."

[41] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on iOS: When benign apps become evil," in *22nd USENIX Security Symposium (USENIX Security '13)*.

[42] M. Stute, S. Narain, A. Mariotto, A. Heinrich, D. Kreitschmann, G. Noubir, and M. Hollick, "A billion open interfaces for eve and mallory: MitM, DoS, and tracking attacks on iOS and macOS through apple wireless direct link," in *28th USENIX Security Symposium (USENIX Security '19)*.

[43] A. Cunningham. (2025) macos sequoia makes you jump through more hoops to disable gatekeeper app checks. [Online]. Available: https://arstechnica.com/gadgets/2024/08/macos-15-sequoia-makes-you-jump-through-more-hoops-to-disable-gatekeeper-app-checks/

[44] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, "{KextFuzz}: Fuzzing {macOS} kernel {EXTensions} on apple silicon via exploiting mitigations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5039–5054.

[45] Y. Wang, Y. Hu, X. Xiao, and D. Gu, "iservice: Detecting and evaluating the impact of confused deputy problem in appleos," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.

[46] D. Winder, "Forbes - mac users warned as 'fully undetectable' security backdoor confirmed," https://www.forbes.com/sites/daveywinder/2025/02/04/mac-users-warned-as-fully-undetectable-security-backdoor-confirmed/, 2025.

[47] C. Z. Harris. (2024) Apple's macos sequoia changes how you install unsigned apps. [Online]. Available: https://www.idownloadblog.com/2024/08/07/apple-macos-sequoia-gatekeeper-change-install-unsigned-apps-mac/

[48] T. Holwerda. (2024) Bug or intentional? macos 15.1 completely removes ability to launch unsigned applications. [Online]. Available: https://www.osnews.com/story/141055/bug-or-intentional-macos-15-1-completely-removes-ability-to-launch-unsigned-applications/

[49] Apple Support Communities. (2024) How to run unsigned apps in macos 15.1? [Online]. Available: https://discussions.apple.com/thread/255759797?sortBy=rank

[50] TeamViewer, "Remote control a mac," https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-classic/remote-control/remote-control-a-mac/, 2023.

[51] P. Wardle, "Proton: Osx rat," https://objective-see.org/blog/blog_0x14.html, 2017.

[52] C. W. Charlie Osborne, "Mami malware targets mac os x dns settings," https://www.zdnet.com/article/mami-malware-targets-mac-os-x-dns-settings/, 2018.

[53] Team Signhouse, "1Password Revenue and Growth Statistics (2024)," Aug. 2024. [Online]. Available: https://usesignhouse.com/blog/1password-stats/

[54] Keeper Security, "Keeper Reaches 1 Million Customers Worldwide," May 2020, publisher: Keeper Security. [Online]. Available: https://www.keepersecurity.com/blog/2020/05/27/keeper-reaches-1-million-customers-worldwide/

[55] M. Kapko, "What's at stake for 33M compromised LastPass users?" *Cybersecurity Dive*, Jan. 2023. [Online]. Available: https://www.cybersecuritydive.com/news/lastpass-breach-high-stakes/639838/

[56] KeePassXC Team, "Install KeePassXC on Linux," Dec. 2024. [Online]. Available: https://flathub.org/apps/org.keepassxc.KeePassXC

[57] HicknHack Software GmbH, "MacPass," Jan. 2025, original-date: 2012-07-21T00:48:01Z. [Online]. Available: https://github.com/MacPass/MacPass

[58] Electron. (2024) Browserwindow: `setAlwaysOnTop` flag, level, and relativelevel. Accessed: 2024-10-11. [Online]. Available: https://www.electronjs.org/docs/latest/api/browser-window#winsetalwaysontopflag-level-relativelevel

[59] J. Stallings, "RobotJS: Desktop automation for node.js," https://robotjs.io/, 2015, accessed: October 11, 2024.

[60] M. Eddy, "The Best Password Managers," *The New York Times*, Oct. 2024. [Online]. Available: https://www.nytimes.com/wirecutter/reviews/best-password-managers/

[61] S. Gilbertson, "The Best Password Managers to Secure Your Digital Life," *Wired*, Apr. 2024. [Online]. Available: https://www.wired.com/story/best-password-managers/

[62] Apple Developer, "Managing your app's information property list," accessed: 2024-11-06. [Online]. Available: https://developer.apple.com/documentation/bundleresources/information_property_list/managing_your_app_s_information_property_list

[63] Apple Inc., *The Bundle Identifier*, 2021. [Online]. Available: https://developer.apple.com/documentation/bundleresources/information_property_list/cfbundleidentifier

[64] Code Signing Guide, "About Code Signing," Sep. 2016, publisher: Apple Inc. [Online]. Available: https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html

[65] "Code signing process of macos applications," https://support.apple.com/guide/security/app-code-signing-process-sec3ad8e6e53/web, 2024.

[66] M. Developers, "Macos accessibility api," https://developer.apple.com/accessibility/, 2024.

[67] 1Password, "How to use universal autofill on mac," https://1password.com/features/how-to-use-universal-autofill-on-mac/, 2024.

[68] M. Developers, "Autotype," https://github.com/MacPass/MacPass/wiki/Autotype, 2024.

[69] K. Developers, "Auto-type — keepassxc user guide," https://keepassxc.org/docs/KeePassXC_UserGuide#_auto_type, 2024.

[70] M. Naseri, N. P. Borges Jr, A. Zeller, and R. Rouvoy, "Accessileaks: Investigating privacy leaks exposed by the android accessibility service," in *PETS 2019-The 19th Privacy Enhancing Technologies Symposium*.

[71] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee, "A11y attacks: Exploiting accessibility in operating systems," in *Proceedings of the 2014 ACM CCS*.

[72] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, "Kindness is a risky business: On the usage of the accessibility {APIs} in android," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*.

[73] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: from two permissions to complete control of the ui feedback loop," in *2017 IEEE Symposium on Security and Privacy (SP)*.

[74] 1Password Support, "How 1password fills information in apps on your mac," https://support.1password.com/mac-universal-autofill-settings/, November 2022, accessed: 2024-10-11.

[75] A. D. Documentation, "Nswindow.level," https://developer.apple.com/documentation/appkit/nswindow/level-swift.struct, 2024.

[76] J. Fisher, "What is the order of nswindow levels?" https://jameshfisher.com/2020/08/03/what-is-the-order-of-nswindow-levels/, 2020.

[77] X. Lin, P. Ilia, and J. Polakis, "Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill," in *Proceedings of the 2020 ACM CCS*.

[78] 1Password Support, "Save and fill credit cards and addresses," https://support.1password.com/credit-card-address-filling, accessed: 2025-04-12.

[79] D. M'Raihi, J. Rydell, M. Pei, and S. Machani, "TOTP: Time-Based One-Time Password Algorithm," Internet Engineering Task Force, Request for Comments RFC 6238, May 2011. [Online]. Available: https://datatracker.ietf.org/doc/rfc6238/

[80] 1Password, "1password cli documentation: Get started," https://developer.1password.com/docs/cli/get-started/, 2025.

[81] ——, "Secret key security," https://support.1password.com/secret-key-security/, 2025.

[82] Keeper Security, "Keyboard shortcuts: Tips and tricks," 2024. [Online]. Available: https://docs.keeper.io/en/user-guides/tips-and-tricks/keyboard-shortcuts

[83] Electron Contributors, "app.dock.hide() - electron documentation," https://www.electronjs.org/docs/latest/api/app#appdockhide-macos, 2024.

[84] KeePassXC Team, "Keepassxc user guide," 2024. [Online]. Available: https://keepassxc.org/docs/KeePassXC_UserGuide

[85] K. Team, "Keyboard shortcuts," 2025. [Online]. Available: https://keepass.info/help/kb/keyb_shortcuts.html

[86] 1Password, "Keyboard shortcuts — 1password support," 2024. [Online]. Available: https://support.1password.com/keyboard-shortcuts/

[87] N. Turner, "macos secure input mode: Understanding its purpose and implications," https://nickjvturner.com/macos-secure-input-mode, 2024.

[88] Apple Inc., *Use Secure Keyboard Entry in Terminal on Mac*, https://support.apple.com/guide/terminal/use-secure-keyboard-entry-trml109/mac#, 2024.

[89] Tenable, "5.10 Ensure Secure Keyboard Entry Terminal.app Is Enabled," https://www.tenable.com/audits/CIS_Apple_macOS_12.0_Monterey_Cloud-tailored_v1.0.0_L1, 2025.

[90] Microsoft Docs, "Use code signing for better control and protection," Microsoft Learn, 2025, https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/deployment/use-code-signing-for-better-control-and-protection.

[91] npm, "node-window-manager," 2024. [Online]. Available: https://www.npmjs.com/package/node-window-manager

[92] npm contributors, "node-window-manager: Node.js library for managing native application windows," https://www.npmjs.com/package/node-window-manager.

[93] J. Sissel, "xdotool: Fake keyboard/mouse input, window management, and more," https://github.com/jordansissel/xdotool, 2025.

[94] Microsoft Developer Network, "SetWindowsHookExA function (winuser.h)," https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa, 2024.

[95] Apple Inc., "What is keychain access on mac?" 2024. [Online]. Available: https://support.apple.com/guide/keychain-access/what-is-keychain-access-kyca1083/mac

[96] 1Password, "Passkeys: The future of secure sign-in," 2024. [Online]. Available: https://1password.com/product/passkeys

[97] LastPass, "Passwordless authentication with passkeys," 2024. [Online]. Available: https://www.lastpass.com/features/passwordless-authentication

[98] M. W. Docs, "Web authentication api," https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API, 2024.

[99] D. Silver, S. Jana, D. Boneh, and E. Chen, "Password Managers: Attacks and Defenses," in *USENIX Security 14*.

[100] Y. Fu and D. Wang, "Leaky autofill: An empirical study on the privacy threat of password managers' autofill functionality," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2024.

[101] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *2017 IEEE Symposium on Security and Privacy (SP)*.

[102] A. Gangwal, S. Singh, and A. Srivastava, "AutoSpill: Credential Leakage from Mobile Password Managers," in *Proceedings of ACM CODASPY*, 2023.

[103] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, "Design of the eros Trusted Window System," 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/design-eros-trusted-window-system

[104] J. Epstein, J. McHugh, R. Pascale, C. Martin, D. Rothnie, H. Orman, A. Marmor-Squires, M. Branstad, and B. Danner, "Evolution of a trusted B3 window system prototype," in *1992 IEEE Computer Society Symposium on Research in Security and Privacy*.

[105] F. Roesner and T. Kohno, "Securing Embedded User Interfaces: Android and Beyond," in *USENIX Security Symposium '13*, 2013.

[106] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *2015 IEEE S&P*.
[107] Y. Lee, X. Wang, K. Lee, X. Liao, X. Wang, T. Li, and X. Mi, "Understanding iOS-based Crowdturfing Through Hidden UI Analysis," in *USENIX Security Symposium '19*.
[108] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks," in *23rd USENIX Security Symposium (USENIX Security '14)*.
[109] M. Jubur, P. Shrestha, N. Saxena, and J. Prakash, "Bypassing Push-based Second Factor and Passwordless Authentication with Human-Indistinguishable Notifications," in *ACM AsiaCCS '21*.
[110] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, "Phishing Attacks on Modern Android," in *Proceedings of the 2018 ACM CCS*.
[111] A. T. Mahdad, M. Jubur, and N. Saxena, "Breaching Security Keys without Root: FIDO2 Deception Attacks via Overlays exploiting Limited Display Authenticators," in *ACM CCS '24*.
[112] K. Cheng, A. Bhattacharya, M. Lin, J. Lee, A. Kumar, J. F. Tian, T. Kohno, and F. Roesner, "When the user is inside the user interface: An empirical study of ui security properties in augmented reality," in *33rd USENIX Security Symposium (USENIX Security '24)*.
[113] F. Xiao, Z. Yang, J. Allen, G. Yang, G. Williams, and W. Lee, "Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem," in *Proceedings of the 2022 ACM CCS*.
[114] M. M. Ali, M. Ghasemisharif, C. Kanich, and J. Polakis, "Rise of Inspectron: Automated Black-box Auditing of Cross-platform Electron Apps," in *USENIX Security Symposium '24*.
[115] C. Paloscia, K. Solomos, M. M. Ali, and J. Polakis, "Lost in translation: Exploring the risks of web-to-cross-platform application migration," *Proceedings on Privacy Enhancing Technologies*, 2025.
[116] Z. Jin, S. Chen, Y. Chen, H. Duan, J. Chen, and J. Wu, "A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities," in *Proceedings 2023 Network and Distributed System Security Symposium*, 2023.
[117] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld, "SandTrap: Securing JavaScript-driven Trigger-Action Platforms," in *USENIX Security Symposium '21*.
[118] C. Wang, Y. Zhang, and Z. Lin, "RootFree Attacks: Exploiting Mobile Platform's Super Apps From Desktop," in *ACM AsiaCCS '24*.
[119] 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 *2012 IEEE S&P*.
[120] Z. Li, W. He, D. Akhawe, and D. Song, "The Emperor's New Password Manager: Security Analysis of Web-based Password Managers," in *USENIX Security Symposium '14*.
[121] 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 *USENIX Security '20*.
[122] A. Fabrega, A. Namavari, R. Agarwal, B. Nassi, and T. Ristenpart, "Exploiting Leakage in Password Managers via Injection Attacks," in *USENIX Security '24*.

APPENDIX

A. Password Manager Notifications Examples

Figure 3 shows the alert displayed when a credentials set is selected from the Quick Access window.

B. Keeper: Phishing Application Structure

Figure 4 illustrates the internal structure of the phishing application used to exploit Keeper's vault. The outer wrapper, named `Hider.app`, serves as a benign-looking macOS application designed to conceal the embedded malicious component. Within `Hider.app`, a nested application (`Malicious.app`) contains the core functionality responsible for credential harvesting. Both applications follow the standard macOS app bundle structure, with the `Contents` folder that holds essential files and directories required for execution. The `Malicious.app` includes two critical files, highlighted

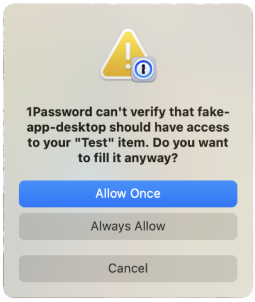

Fig. 3: Alert window.

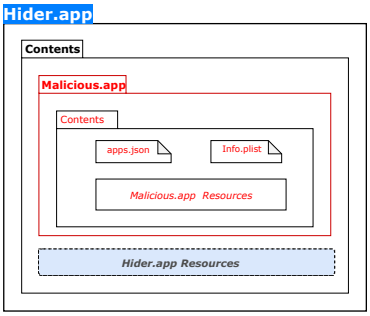

Fig. 4: Phishing app structure.

in red: `apps.json` and `Info.plist`. The `apps.json` file contains a predefined list of target applications and their corresponding configurations, such as window titles or Bundle Display Names. The phishing application uses this file to identify which credentials to harvest from Keeper's vault. The `Info.plist` file stores essential metadata about the application, including its Bundle Display Name. By modifying this file, the attacker can impersonate trusted applications, effectively bypassing Keeper's verification checks.

C. Vault Integrity

We detected an additional design flaw in 1Password's credential association process on macOS. The application links credential records to Bundle IDs based solely on the presence of a code signature, even when that signature is invalid or untrusted. Specifically, 1Password establishes an association between a credential record and a Bundle ID upon verifying the presence of a Code Signature. Notably, this association is created even if subsequent verification checks fail, such as assessing the validity of the signature. To evaluate the security implications of this behavior, we tested a scenario involving credentials for a legitimate service that had not yet been used for autofill in the corresponding legitimate application. We verified that an attacker could exploit this mechanism by tricking a user into installing a malicious application. When the user attempts to autofill credentials for the legitimate service, 1Password verifies that the malicious application includes a valid Code Signature and subsequently associates the service's credentials with the malicious app's Bundle ID. However, in its subsequent validation steps, 1Password detects that the Code Signature is either *untrusted* or *unverified* and blocks autofill into the malicious application. Although this validation prevents immediate credential exfiltration, it introduces a flaw, as once the association between the credentials and the malicious Bundle ID is established, 1Password subsequently blocks any attempt to update the association when the user attempts to autofill the same credentials into the legitimate application. This results in a persistent *Denial of Service (DoS)* condition, effectively preventing users from accessing their credentials within the intended, legitimate application. While this issue is not part of our credential stealing attack, nonetheless it presents an additional design flaw in 1Password's verification workflow.

TABLE IV: Default and user-defined autofill shortcuts in macOS password managers.

(a) Default password manager shortcuts.

Shortcut	Password Manager
Cmd + \	1Password, Dashlane, Enpass
Cmd + Shift + L	Bitwarden
Cmd + Shift + Space	LastPass
Cmd + Shift + U	Keeper
Cmd + Shift + X	NordPass
Cmd + Shift + P	RoboForm
Cmd + Shift + M	MacPass

(b) User-defined shortcuts.

Shortcut		
Cmd + /	Cmd + .	Cmd + ,
Cmd + ‘	Cmd + ’	Cmd + D
Cmd + Return		

D. Linux Password Managers

For Linux systems, we applied the techniques outlined in §IV and §V to evaluate the availability and behavior of autofill functionality in desktop password managers. Among the evaluated password managers, 1Password and KeePassXC offer native Linux desktop applications, while Keeper provides limited functionality, and LastPass does not support the platform. The 1Password client does not implement autofill on Linux but provides a Quick Access interface, allowing users to copy credentials to the clipboard via keyboard shortcuts. As previously assessed in §IV-A, this interface can be manipulated by a malicious application to extract credentials through automated interaction. We replicated this attack on Linux and confirmed its effectiveness. Similarly, KeePassXC supports autofill functionality, and we confirmed that credential injection can be programmatically triggered. However, in both cases, OS-level constraints on Linux significantly limit the stealthiness of such attacks. Specifically, Linux lacks a privileged UI layer—such as the `screen-saver` level available on macOS—that could be used to obscure the attack interface.

Additionally, the Linux window manager prevents applications from positioning windows outside the visible screen boundaries. We also tested workspace-based hiding strategies, but triggering either the Quick Access or autofill interface causes it to appear in the user's current workspace. Although automated credential extraction is technically feasible, stealthy exploitation under default Linux configurations is impractical. Finally, the Keeper desktop application does not support autofill on Linux and is therefore not susceptible to the class of attacks explored in this work. LastPass does not provide a Linux desktop application and is thus excluded from our evaluation.

TABLE V: Average time to infer KeePassXC's autofill shortcut per phase (avg across five iterations).

Shortcut Set Tested	Avg. Time (s)
Default manager shortcuts	10.54
Common user-defined shortcuts	21.12
Special key-based combinations	99.75

Algorithm 2: Shortcut inference for KeePassXC.

Input: *commonDefaultShortcuts*, *commonUserDefinedShortcuts*
Output: Detected autofill shortcut

```
foreach (primaryKey, modifierKeys) in commonDefaultShortcuts ∪ commonUserDefinedShortcuts do
    Trigger modifierKeys + primaryKey
primaryKeys ← extract all unique keys from prior sets
modCombos ← {CTRL, CTRL+SHIFT, CTRL+OPTION, CTRL+SHIFT+OPTION}
foreach key in primaryKeys do
    foreach mods in modCombos do
        Trigger mods + key
```

E. KeePassXC: Inferring the Autofill Shortcut

Unlike other password managers, KeePassXC does not define a default autofill shortcut. Consequently, successful credential harvesting requires an attacker to infer the user's configured keyboard shortcut. Exhaustive brute-force across all potential combinations is not practical due to the high number of permutations. To address this, we develop a shortcut inference strategy that prioritizes realistic configurations based on common user behavior and commonly deployed shortcuts.

Our method proceeds in three phases. First, it tests the default autofill shortcuts used by other popular password managers, as shown in Table IVa, under the assumption that users may reuse familiar key bindings. Second, it creates a set of user-defined shortcuts commonly recommended in online discussions and documentation and user guides (Table IVb). Finally, it expands the search space by pairing frequently used primary keys with combinations of modifier keys such as `Control`, `Option`, and `Shift`. Algorithm 2 outlines our inference methodology.

We evaluate the effectiveness of each phase independently by measuring the average completion time over five iterations, as shown in Table V. Testing default manager shortcuts completes in 10.5 seconds on average, and represents the most efficient phase due to its low overhead. The user-defined subset completes in ≈20 seconds. The final phase, which explores a broader space of modifier-key combinations, completes in under 100 seconds. These results confirm the practicality of the default-only phase for credential harvesting, while also highlighting that exhaustive exploration of complicated input combinations remains possible within a short time window.