Exploiting Leakage in Password Managers via Injection Attacks

Andrés Fábrega[1], Armin Namavari[1], Rachit Agarwal[1], Ben Nassi[2,3], Thomas Ristenpart[1,2]
[1] *Cornell University* [2] *Cornell Tech* [3] *Technion - Israel Institute of Technology*

Abstract

This work explores *injection attacks* against password managers. In this setting, the adversary (only) controls their own application client, which they use to "inject" chosen payloads to a victim's client via, for example, sharing credentials with them. The injections are interleaved with adversarial observations of some form of protected state (such as encrypted vault exports or the network traffic received by the application servers), from which the adversary backs out confidential information. We uncover a series of general design patterns in popular password managers that lead to vulnerabilities allowing an adversary to efficiently recover passwords, URLs, usernames, and attachments. We develop general attack templates to exploit these design patterns and experimentally showcase their practical efficacy via analysis of ten distinct password manager applications. We disclosed our findings to these vendors, many of which deployed mitigations.

1 Introduction

Password-based authentication suffers from well-know pitfalls, such as the fact that users tend to choose passwords that can be easily guessed by attackers [28]. Password managers are often cited as the default solution to this problem [32, 56], as users can offload to them the complexities of password generation, storage, and retrieval. Indeed, password managers have enjoyed a noticeable rise in popularity [28, 51], placing them among the most ubiquitous security-oriented tools.

Password managers have benefited from academic attention [8, 9, 13, 22, 24, 25, 27, 38, 43, 46, 52, 53], which has helped understand and improve their security along various dimensions. The attacks uncovered by prior work broadly fall under two general threat models. First are attacks that use a client-side resource controlled by the adversary, such as a malicious website visited by the client [38, 53], a rogue application in the victim's device [9, 22], or the client's WiFi network [52]. Second are adversaries that somehow acquire a copy of a user's encrypted vault, and exploit leakage from unencrypted vault metadata [24, 43] or by offline cracking attacks of a user's master password [8, 13, 27]. State-of-the-art password managers are therefore designed to resist both kinds of threats and, notably, use slow cryptographic hashing to prevent cracking attacks for well-chosen master passwords.

In this work, we consider a new kind of threat model in which an adversary (1) controls their own application client, through which they can send chosen payloads to the victim (for example, via the password sharing feature found in most modern password managers); and (2) can observe some form of encrypted state and associated metadata, such as the user's encrypted vault backups or network requests received by the application servers. Borrowing terminology from prior work in other domains (see Section 2), we refer to attacks in this threat model as *injection attacks*.

The core idea behind injection attacks is that the adversary can use injections to trigger subtle interactions in the application logic between their data and target victim data (e.g., other passwords used by the target), which are reflected in their observations of ciphertexts (e.g., inspecting their lengths) and metadata in a way that allows recovering sensitive information. We argue that this threat model is increasingly important as password managers become more complex and feature-rich, which provides new avenues for injection mechanisms and vulnerable cross-user interactions.

To understand whether this threat model is of practical concern or not, we performed a security analysis of ten popular password managers that support sharing—LastPass, Dashlane, Zoho Vault, 1Password, Enpass, Roboform, Keeper, NordPass, Proton Pass, and KeePassXC. Together these reportedly account for over 30% of all password manager users [51]. We uncover a series of exploitable vulnerabilities that implicate all of the password managers investigated.

Our first class of attacks exploits the fact that a common feature of password managers is for clients to periodically log outside the device various metrics about the "health" of a user's vault, such as the number of duplicate passwords. We show how an adversary can leverage these benign-looking metrics to perform an efficient binary-search-based dictio-

Attack vector	Leakage	Adversary type	Vulnerable applications
Vault-health logs	Passwords	Eavesdropper	L, D, Z, E, R, K, N
Vault-health logs	Passwords	Network	Z
URL icon fetching	URLs	Network	D, 1P, E, R, P, N
Attachment deduplication	Attachment	Eavesdropper	KX
Compression	URLs and usernames	Eavesdropper	KX

Figure 1: Summary of the vulnerabilities discovered in this work, which lead to efficient attacks that recover sensitive information from a victim's vault. These vulnerabilities were present in ten applications we studied: LastPass (L), Dashlane (D), Zoho Vault (Z), 1Password (1P), Enpass (E), Roboform (R), Keeper (K), NordPass (N), Proton Pass (P), and KeePassXC (KX).

nary attack that recovers the target user's saved passwords. Our attacks do not require the adversary to know additional information about the victim's saved credentials beforehand (for example, URLs nor usernames). Seven out of the ten applications are vulnerable to this attack. In most cases, the adversary must be a passive *eavesdropper* that observes these metrics directly (for example, by having a persistent foothold in the application servers), while for one application the attack is feasible by a passive *network adversary* that simply observes the HTTPS channels under which the E2EE data is transmitted. We note that both eavesdropping and network adversaries are within scope of the threat models under which password managers are designed, and the ubiquity of server-side breaches [37, 54, 58], combined with the difficulty of detecting such breaches [50], make it critical that password managers resist such attacks.

Our second class of attacks exploits another feature of password managers: clients often display a small identifying icon, such as a company logo, alongside each of a user's saved credentials. Importantly, such icons are only fetched once per URL, and subsequent credentials reuse the icon stored in the client. We show how this fact allows an adversary to perform an efficient dictionary attack on the URLs in a victim's vault. The attack always succeeds in our experiments, and mounting it requires no additional assumptions about the victim's saved credentials. Six of our case study applications are vulnerable to this attack, and in all cases exploitation only requires observations by a network adversary.

We then turn our attention to adversaries that have an encrypted copy of the entire vault, such as compromising a local password-protected database file or backup of it. In this case, we analyze the security of KDBX [12], which is a file format used by many password managers, notably KeePass [33] and its derivatives [12, 57]. To optimize for storage, KDBX employs a variety of storage-saving techniques, such as file deduplication and compression. We show two attacks exploiting these features to recover URLs, usernames, and attachment contents. Compression and deduplication have led to attacks against other systems before (see Section 2), but our work is the first to show that these types of vulnerabilities also arise in the context of password managers. Our attacks target features of the underlying file format itself, and thus can potentially be leveraged against any application that uses KDBX. We implement a proof-of-concept for our attacks in the case of KeePassXC, and experimentally show that its accuracy is sufficiently high to make it a practical threat.

A summary of our attacks is shown in Figure 1. They exploit common design patterns found in password managers, and as such other applications that employ these can be vulnerable to our attacks. Indeed, for each of our attacks, we describe a general template for it, which is agnostic to lower-level application details, and that can be used to target any application that follows the relevant design pattern. More broadly, our findings uncover higher-level issues in password manager design, and we discuss the future work that will be required to provide generally applicable mitigations for injection attacks.

Summary of contributions. We begin the study of a new threat model for password managers called injection attacks. We identify three design patterns that lead to attacks, and we implement practical attacks that affect a variety of applications. We stress, however, that our attack vectors are features of password managers, instead of bugs that are specific to our case study applications, and thus other password managers can potentially be vulnerable to them. Some of our vulnerabilities reveal new attack vectors, whereas others exploit known malpractices (compression and file deduplication). Thus, for the latter, our work is the first to show that these issues are also present in password managers.

Our attacks highlight broader classes of design malpractices found in password managers (and E2EE applications more generally). We close this work by identifying these higher-level issues, and outlining a series of takeaways for application designers. Our results thus pave the way for future work along various dimensions: identifying other password managers that are vulnerable to our attacks, uncovering other patterns that lead to injection attacks, and designing general tools to study and mitigate injection attacks. We expand on these ideas, and other opportunities for future work, in Section 8.

Ethics. All of our attacks were performed against isolated research accounts. We kept the scale of our proof-of-concept experiments as minimal as possible, so as to confirm the attacks' efficiency without overloading any application or cloud servers. Some of our experiments required a high volume of

network traffic, for which we implemented local simulators that we validated via smaller experiments with real clients.

We first studied four applications (LastPass, Dashlane, Zoho Vault, and KeePassXC), and disclosed our findings to these vendors, who have since all deployed mitigations for our vulnerabilities (see Section 8). We later expanded the scope of our study to include more applications, and are thus currently in a second round of disclosures with the other six vendors, who are investigating our results. We have made ourselves available to help with mitigations before public release of our findings. We will update the paper to document the results of the disclosure processes with the six remaining vendors once they are complete.

2 Related Work

Security analysis of password managers. Prior work has investigated attacks on password managers in threat models involving some combination of client-side attacks, adversarial networks, and a malicious service provider. Such examples include attacks that rely on malicious applications in the victim device [22], malicious websites that the victim is tricked into visiting [38], XSS adversaries that inject code into a website's login page [53], and a rogue WiFi network under the adversary's control [52]. The goal of the adversary is to exploit some feature of the application, such as password generation [43], autofill policies [52], and clipboard vulnerabilities [9, 22], to exfiltrate user passwords.

More relevant to our findings, some existing attacks on password managers involve an adversary that obtains the encrypted database of the victim [8, 13, 24, 27, 43]. However, most of these attacks focus on offline cracking of the master password, using the recovered vault as a decryption oracle [8, 13, 27]. Other works, such as [24] and [43], simply document unencrypted metadata in the encrypted database files or consider a much weaker adversarial goal, namely, producing a new, valid database after observing other valid ones (for example, by tweaking metadata headers). Neither attack violates the confidentiality of the password vault contents.

Our injection attack threat model is different from settings explored in prior work on password managers. We consider an adversary that, in addition to potentially compromising the platform and/or network, can spin up clients of its own that interact with the victim client using standard password manager features. The adversary uses these cross-user interactions to mix data of its choosing with sensitive victim data. Then, via a leakage channel, the adversary learns information about the combination of the victim data and its own injected data.

Injection attacks. Although we are the first to apply injection attacks to password managers, prior work has studied the injection threat model in other contexts. For example, [10, 59, 60] present attacks against searchable encryption schemes and encrypted search indexes, where the adversary is able to inject payloads that trigger a query into the encrypted store. Our attacks take inspiration from techniques used in this prior work, such as the binary search attack presented in [60]. However, our contribution lies in how we apply these techniques through the leakage channels and injection vectors we identify, specific to the password manager setting.

A setting closer to ours is that of [20], which introduces attacks against E2EE backups of messaging applications, in the presence of an adversary that can message the victim and subsequently observe their encrypted chat backups. A few of our attacks (Section 7) rely on similar attack vectors (file deduplication and compression). However, beyond targeting different types of applications, our attacks represent a different class of injection attacks: our work exploits encrypted state *synchronization*, whereas [20] exploits encrypted state *backups*. As such, our setting represents a richer attack surface, with a higher frequency and granularity of observations, and so our attacks are significantly more practical, as we explain further in this section. Furthermore, our attacks exploit features such as health metric logs and URL icon fetching, going beyond the attack vectors explored in prior work on encrypted backups.

Attacks on compression before encryption. It has long been known that compression before encryption can lead to vulnerabilities [34], which has resulted in exploits against real-world systems [20, 26, 31, 34, 44, 49, 61], such as TLS [26, 49] and the iMessage E2E encrypted messaging protocol [23]. Our work, however, is the first to exploit compression in the context of password managers (Section 7.2). Our attacks combine techniques from prior works (highlighted, as needed, in the attack descriptions) with novel insights in order to exploit this new setting.

Some works have studied compression in the broader context of encrypted databases, such as [20, 31, 44]. The attacks in both [44] and [31] rely on assumptions that are not present in our setting, such as physical access to the target's handset and the ability to unlock it [44], and little to no noise in the side channel [31, 44]. Further, their attacks are tailored to the specific systems that are being targeted. The setting in [20] is much more limited than our work, as the adversary is limited by daily backups, every injection results in much more noisy metadata that is added to the database, and the adversary cannot edit past injections. Thus, we devise new attacks that are significantly more practical: our attacks handle larger dictionary sizes (hundreds of items instead of, e.g., 10 to 20 items), have higher accuracy (for example, for a dictionary of size 20—the largest [20] experiments with—our attacks succeed with 90% probability instead of 20-30%), and have an additional confirmation step to verify if the found item is the correct one or not. Further, their compression-based attacks require the victim not to send or receive external messages for multiple days, whereas our attacks run in a matter of minutes.

File deduplication, a common form of compression before

encryption, has been exploited in other contexts, such as client-side encrypted file storage [29, 30] and E2EE messaging [20]. Our injection attacks exploiting deduplication (Section 7.1) require new techniques due to details of KDBX 4's architecture. In particular, KDBX 4 employs both deduplication *and* compression, which required mitigating noise from the latter. We also note that the attack from [20] is not applicable to the applications we consider in this work.

3 Password Managers Background

We describe the general architecture of password managers in this section.

Password manager abstraction. We denote by U and PW a user of a password manager service S and their account password, respectively. The user U owns one or more devices with an application client for S installed in each. The types of clients available vary by service, but these typically consist of mobile applications, desktop applications, web applications, and browsers extensions. Each client stores U's saved passwords and other information in the local storage of its respective device in the form of a *local vault*, which is (often) encrypted. We represent the contents of a vault by $\mathcal{V} = \{e_1, ..., e_m\}$, where each e is a vault entry storing U's credentials for a website. Each e contains various fields such as username (e_{user}), URL (e_{url}), password (e_{pw}), and a list of attachments (e_{attach}).[1]

In order to have consistent local vaults, U's devices need to synchronize their state. For most password managers, this involves clients periodically communicating with platform servers, which serve as intermediaries that facilitate synchronization. To do so, clients export an E2EE version of (only) the latest state changes (e.g., new passwords added), using an application-specific symmetric encryption scheme, and send this to S's servers using HTTPS. The server then forwards the updates to all other devices the next time they come online; the receiving clients decrypt the changes and update their local vaults accordingly.

Alternatively, if the service provider does not directly facilitate synchronization, U has to ensure that their clients are consistent, either by propagating updates manually, or by using out-of-band synchronization mechanisms such as storing the encrypted database file on an external cloud-storage provider. In this case, state updates are less granular, as these consist of a re-encryption and a re-upload of the entire database file to the cloud via HTTPS. Note that this setup is explicitly suggested by various applications that do not automatically synchronize clients [19, 55].

State synchronization is very frequent: exports generally occur after every modification to the local database, and clients automatically check for imports (from S or from modifications to their cloud-stored file) every couple of seconds; the exact periodicity varies across services.

Database file formats. Most password managers use simple data structures and file formats for their local vaults, which typically consist of encrypting each field of each $e \in \mathcal{V}$ separately, and sequentially organizing these entries using some lightweight file format like JSON or XML. Some file formats then additionally employ a variety of storage-saving mechanisms, such as attachment deduplication and compression, to minimize the size of the database file. This pattern is most often seen in applications that do not route information through S's servers, as everything is stored on the user's device and (potentially) exported to some third-party cloud service. Conversely, other applications can leverage S's servers for storage; for example, instead of storing the binary content of all attachments, local vaults can just store a pointer to a blob store managed by S, which contains the encrypted file itself.

Credential sharing. A common feature of modern password managers is *cross-user sharing*, which allows users to jointly hold entries in their vaults. For this, whenever some user wants to share a credential e from their vault with U, the former derives an ephemeral key that is used to encrypt e, and sends this ciphertext alongside the ephemeral key (itself encrypted under a public key associated with U). Next time one of U's devices comes online, its client will download the shared items and decrypt them locally to recover e.

We note that, generally, U must first accept the shared entry e before it gets added to their vault.[2] However, subsequent updates to e require no approval, and are propagated automatically to all clients. Further, many password managers allow users to share *folders* in addition to individual entries. In this case, U must once again accept the initial share of the folder, but all future updates, including adding or deleting new entries to the folder, require no approval.

For some applications, credential sharing is restricted to accounts that are part of the same "organization", e.g., where both accounts are affiliated with a corporate or family license for the application. In this case, shared credentials should still provide E2EE guarantees, such as cryptographic access control, even across all organization members. These guarantees should hold even in the presence of privileged users such as organization administrators (who may have access to organization-wide metadata).

4 Threat Model and Case Studies

While password managers have traditionally been non-interactive applications, newer features like credential sharing require us to re-think their security model: what attacks, if

[1] A common feature of password managers is for users to be able to attach arbitrary files, e.g., sensitive documents, to their vault entries.

[2] Some managers place additional trust constraints on password sharing: Zoho Vault only allows sharing with users who are part of the same "organization", which are groups that the user can belong to.

any, arise from *interaction with other malicious clients*? This question is the starting point for our threat model.

4.1 Threat Model

Our threat model, which we describe in detail in this section, assumes an adversary that (1) can inject content into the victim's vault, and (2) subsequently observe some form of protected application state.

Injections. The injection channel of our attacks will be the aforementioned *cross-user sharing* feature of password managers. That is, the adversary can share with the victim a credential e, which gets incorporated and synchronized across all victim vaults, resulting in e being "injected" into $\mathcal{V}$. We stress that, in our setting, the adversary exclusively controls an (unmodified) application client, and sends their payloads through the standard interface provided by the application. Thus, tampering with either the victim or the adversary's clients, or controlling any other parts of the environment, is explicitly out of scope.

As mentioned earlier, password managers often require the victim to first accept the shared items before they are incorporated into their vault, which is thus a necessary assumption required to establish an injection channel. We stress, however, that all of our attacks require the victim to accept a *single* shared item, as all subsequent updates to it, through which injections are performed, require no approval. As such, our threat model assumes some initial degree of trust between the user and the adversary, or that the latter can trick the former into accepting a share request. Accepting such a request, however, should not lead to disclosure of a user's vault content.

Observations. Our threat model then assumes that the adversary has *persistent access* to some function of the data that leaves the victim's device, e.g., as a result of recurring backups or state synchronization between devices. We distinguish between two variants, depending on the trust assumptions required for each attack: (1) an *eavesdropping adversary*, who has access to the E2EE data itself, e.g., ciphertexts of new passwords, as well as other plaintext metadata; and (2) a weaker *network adversary* that can only observe the HTTPS packets under which the E2EE data is transmitted. We highlight the difference between both settings in Figure 2. We discuss the periodicity of synchronization (and, hence, the frequency of adversarial observations) for all case study applications in the relevant attack sections.

Password managers promise end-to-end confidentiality [5, 7, 35], i.e., a user's data is compromised only if their master password is leaked, and so password managers are designed to protect against eavesdropping and network adversaries. A network adversary can arise from any number of traffic analysis techniques, such as ARP or DNS spoofing, BGP hijacking, router compromise, a malicious ISP, etc. An eavesdropping adversary generally corresponds to a malicious or breached

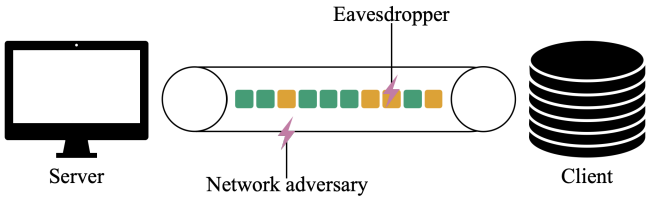

Figure 2: A network adversary can observe the HTTPS packets under which a mix of E2EE data (green squares) and plaintext metadata (orange squares) is transmitted; an eavesdropper has direct access to these.

service or cloud provider, or from a privileged user within an organization with access to metadata (e.g., an administrator). However, they may also arise from other attacks against the TLS layer of traffic, e.g., attacks on certificate authorities (CA) or malicious client-side proxies. Breaches on password managers and cloud services [37, 54, 58] suggest that such an eavesdropping adversary is a realistic threat model, and indeed *persistent* access is a practical concern: an IBM report from 2022 [50] found that the average time it takes to identify a breach is 212 days.

The process of injections and (passive) observations occurs iteratively: the adversary sends a payload to the victim (either by sharing a new credential, or modifying older ones), observes the resulting encrypted state——which is now a function of both adversarial and sensitive data—adaptatively chooses the next payload to inject, and so on. The adversary's goal is then to back out confidential information from their observations.

A priori, no sensitive user data should be leaked, since the victim's confidential information is encrypted before leaving their device. However, our attacks exploit the fact that application logic often *mixes* personal and externally-received data in subtle ways which may leak information. Thus, an adversary can use injections to trigger these *cross-user interactions*, which are then reflected in the exported application state, revealing confidential data. We stress that injections attacks assume strong cryptographic primitives, and thus low-level, cryptanalytic vulnerabilities are outside their scope; instead, the *lengths* of ciphertexts and plaintext metadata form the basis of our attacks.

In addition to the adversary's injections, the state changes between observations may additionally contain new, benign data added by the victim while the attack is running; we refer to this as the "noisy device" setting, and to the case where the only additions are the adversary's injections as the "quiet device" setting. Some attacks are robust to noise, while others require the victim's client to be quiet while the attack is running; we will specify the noise assumptions required for each attack in the relevant sections.

Out-of-scope attacks. We do not consider attacks in which a compromised service attempts to deploy client code con-

taining backdoors. All E2EE threat models implicitly or explicitly assume trusted client software. Improved assurance here can be aided via mechanisms such as public auditability of software or monitoring via binary transparency services (e.g., [42]). Without trustworthy client side software, no confidentiality is possible. More pragmatically, we note that such client subversion would require an active adversary that hijacks client code distribution, while our injection attacks would be easier to mount, requiring only a passive adversary that either compromises some platform server (e.g., a web or storage server) to enable eavesdropping or has visibility into the target's network communications.

In our threat model, we assume that the adversarially controlled client that performs injections honestly follows the protocol. One could also consider a fully malicious client that deviates from the E2EE protocol when performing injections. We are unaware of any additional attacks that this would enable.

4.2 A Corpus of Password Managers

We gathered a set of password managers to experimentally investigate the feasibility of injection attacks. To build a list of targets, our starting point was a report by Security.org [51] that includes a list of password managers ranked by popularity. We also considered informal online polls on password manager popularity [47, 48]. We investigated the advertised features of each of the resulting 16 applications, and excluded those that do not meet either of two inclusion criteria: (1) the application must support cross-user credential sharing, and (2) the application must target cryptographic access control for shared credentials. The first requirement rules out applications that are unlikely to have any way to inject adversarial content into a target, and the second requirement rules out applications for which simpler attacks would work in our threat model, due to lack of E2EE security guarantees.

All browser-integrated password managers were excluded due to the first criteria, as they do not yet support credential sharing. The second criteria ruled out one application, Bitwarden, which does support credential sharing, but not in a cryptographically secure way: Bitwarden has support for establishing separate "collections" within an organization, but surprisingly all collections use the same secret key. So, in our threat model, any organization member can already decrypt all organization credentials.[3]

Our final list of applications consisted of LastPass (v4.123), Dashlane (v6.2346), Zoho Vault (v3.8), 1Password (v2.25.0), Enpass (v6.11.0), Roboform (v9.6.2), Keeper (v116.18.0), NordPass (v.5.16), Proton Pass (v1.17.4), and KeePassXC (v2.7.6). These applications cover over 30% of all password manager users according to [51], and all follow the basic abstraction presented in Section 3. All application except for KeePassXC and Enpass rely on the application servers for stateful storage and synchronization. These tend to use simple file formats for their local vaults (see [43] for an overview), with no additional storage-saving mechanisms. Conversely, KeePassXC and Enpass do not natively handle synchronizing across clients. Instead, they suggest [19, 55] storing the database file in an external cloud provider to which all user devices have access.

KeePassXC's file format, KDBX [12] (detailed in Section 7), supports attachment deduplication and vault compression. We note that KDBX is used by other password managers, which are ports and derivatives of KeePass [33], a popular open-source password manager that introduced this file format. See [12, 57] for comprehensive lists of applications that use KDBX. Even though we implement our attacks on one such example, KeePassXC (which is a more cross-platform and feature rich version of the original KeePass), our attacks target the underlying file format itself, and thus other applications that use it may be vulnerable to our attacks.

4.3 Overview of Attacks

Our security analysis of the case study applications uncovered three main classes of attacks in the context of injection attacks. We provide an overview of these here, and discuss them in detail in the next three sections.

Attack #1: application-wide metrics (Section 5). The first attack arises from the fact that many password managers compute sensitive metrics, such as the number of duplicate passwords in the vault, across both personal *and* shared vault entries. If these metrics are logged somewhere outside the device (e.g., the application servers), an adversary can trigger fluctuations in these metrics with injections, and observe how they are updated in the external location. As such, besides injecting credentials, in this attack the adversary needs to have access to the location where the metrics are logged (e.g, a foothold in the application servers.)

Attack #2: URL icon fetching (Section 6). The second attack arises from the fact that many password managers display a graphical icon next to each credential, identifying the website for it. This icon is only fetched once from the application servers, and future entries for the same website reuse the image that is stored in the client. Thus, an adversary can use this to test whether the victim has a credential for a particular URL or not, depending on whether the victim's client re-fetches the icon of an injected URL. So, besides updating credentials, in this attack the adversary needs to be able to observe the HTTPS requests that leave the victim's client, or have a foothold in the server where the icons are fetched from.

Attack #3: storage-saving mechanisms (Section 7). The third attacks arise from storage-saving mechanisms that an

[3] We disclosed this to Bitwarden, who confirmed that privilege separation across collections is "at the authorization level, not at the encryption level". Thus, they target a weaker security model than other applications.

Attack	Pre-conditions	LastPass	Dashlane	Zoho Vault	1Password	Enpass	Roboform	Keeper	NordPass	Proton Pass	KeePassXC
App-wide metrics (Section 5)	Support for reports of duplicate password	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	No. of duplicates computed across all passwords	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
	No. of duplicates logged outside the device	✓	✓	✓	✓	✓	✓	✓	✓	×	×
URL icon fetching (Section 6)	Support for URL icons	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	URL icons fetched from servers	✓	✓	×	✓	✓	✓	×	✓	✓	✓
	Fetched URL icons re-used across all vault items	×	✓	×	✓	✓	✓	×	✓	✓	×

Figure 3: Summary of the pre-conditions required for an application to be vulnerable for two of our attacks, each represented by a set of rows, and the conditions satisfied by each application we analyzed. An application that has a ✓ for all pre-conditions in a set of rows is thus vulnerable to the attack.

application may use to decrease the size of the encrypted vault. At a high-level, since these mechanisms remove redundancy in the plaintext vault, the adversary can tell whether their content matched the victim's other credentials or not by looking at fluctuations in the size of the encrypted vault. So, besides injecting credentials, in this attack the adversary would need to have persistent access to the victim's encrypted vault (for example, in a cloud provider where the victim uploads backups of their vault). In our work, we show attacks against two such mechanisms: database compression and attachment deduplication.

In Table 3, we summarize the main pre-conditions required for an application to be vulnerable to our first two attacks, as well as the pre-conditions satisfied by each application in our study. We leave implicit the fact that an application must satisfy the inclusion criteria for our threat model to begin with, e.g., support credential sharing. For our compression-based attack, the only pre-condition is that the application compresses the database file across all vault entries; for our deduplication based attack, the preconditions are (1) the application has support for attachment deduplication, and (2) deduplication occurs across attachments in all vault entries, irrespective of sender. From the applications we investigated, only KeePassXC meets the pre-conditions for these two attacks.

5 Attacks From Application-wide Metrics

Equipped with the background from the preceding sections, we proceed to describe our attacks, starting with our first class of attacks in this section.

Our close study of the network traffic of various password managers revealed that many of them periodically log outside the user's device a variety of metrics about the overall "health" of a user's vault, most notably the number of reused passwords. These logs are sent to either the application servers directly, or to some other member of the user's organization, e.g., an administrator with access to a company-wide security dashboard. These metrics, however, are computed across both personal *and* shared vault entries. As we will show, the adversary can leverage this fact to perform an efficient dictionary attack on a user's passwords.

Dictionary attacks are a standard adversarial goal in the context of password-based authentication, as it is common for users to choose weak or reused passwords [28]. Importantly, and perhaps surprisingly, this is a problem even among users of password managers [39, 45]. This arises from, for example, users that import but do not update old passwords; or users that use password managers for convenience features (e.g., autofill) rather than for security features (e.g., password generation), which is a common practice [21].

We now proceed to describe the attack. We first explain the general structure of it, which serves as an attack template that can be used against any password manager that satisfies the pre-conditions for the attack. For now, we will assume that the adversary is an eavesdropper, and has a persistent foothold to wherever the vault metrics are stored, as described in Section 4.1

Attack description. Given a dictionary of candidate passwords $P := \{p_1, ..., p_n\}$, the adversary $\mathcal{A}$ wants to determine which $p_i \in P$ is present in U's vault $\mathcal{V}$. Let $\mathsf{dup}(\mathcal{V})$ be the number of duplicate passwords across all entries in U's vault, i.e., $\mathsf{dup}(\mathcal{V}) = \frac{1}{2}\sum_{e',e'' \in \mathcal{V}} (e'_{pw} = e''_{pw})$. In this setting, we assume U's clients periodically send $\mathsf{dup}(\mathcal{V})$ to some external location (e.g., automatically every couple of seconds or as a result of a particular action). The key idea behind this vulnerability is that, since $\mathsf{dup}(\mathcal{V})$ is computed across both personal and shared passwords, an eavesdropping adversary with access to this metric can use it as an oracle to determine whether a candidate password p is in the victim's vault or not. Namely, they can compare the value of $\mathsf{dup}(\mathcal{V})$ before and after injecting an entry e such that $e_{pw} = p$; if $\mathsf{dup}(\mathcal{V})$ stays the same, it must be the case that p is not yet in $\mathcal{V}$, and thus is not one of the personal passwords of U. Conversely, $\mathsf{dup}(\mathcal{V})$ increases if and only if p matches one of U's personal passwords.

The adversary can leverage this oracle to find all passwords in P that are in U's vault. There are two injection strategies that $\mathcal{A}$ can use. If the application supports shared folders, instead of "querying" the oracle with each candidate password, one at a time, $\mathcal{A}$ can use a binary-search injection strategy to arrive at the target password more quickly, as follows:

(1) *Setup:* establish a shared folder F with U, which requires U to accept the share.

(2) *Baseline measurement:* wait for a network request from U that contains the initial $n' = \mathsf{dup}(\mathcal{V})$.

(3) *Inject:* split P into two halves P', P'' of equal size. For

each $p \in P'$, create an entry e in F such that $e_{pw} = p$.

(4) *Measure:* wait for a network request from U with an updated $n'' = \mathsf{dup}(\mathcal{V})$.

(5) *Recurse:* set $P_{found} = P'$ if $n'' > n'$, and $P_{found} = P''$ if $n'' = n'$. If $|P_{found}| = 1$, output it as the target password. Otherwise, repeat steps 3–4 with $P = P_{found}$ as the input set and n'' as the baseline measurement.

If the application instead only supports sharing individual entries, $\mathcal{A}$ must use a slower, sequential injection strategy. Let t be the number of individually-shared entries between $\mathcal{A}$ and U, where $t < \frac{n}{2}$ (otherwise, $\mathcal{A}$ can just use the binary-search strategy from above). In this case, $\mathcal{A}$ can inject t candidate passwords at a time, interleaved between observations of $\mathsf{dup}(\mathcal{V})$, by updating the password field in each of the shared entries with a new batch of t candidate items. Eventually, one of these injections will trigger an increase in $\mathsf{dup}(\mathcal{V})$. Then, to find which of the t items of the prior injection is actually the reused password, $\mathcal{A}$ can run the binary-search attack from before, using the t shared entries as a "folder".

Since $\mathsf{dup}(\mathcal{V})$ is deterministic, both variants of the attack always find the target password, if present, or confirm that no password is in the vault. The binary-search injection strategy requires $\lceil \log_2 n \rceil$ injections and observing the same number of network requests containing $\mathsf{dup}(\mathcal{V})$; the sequential injection strategy instead requires $\frac{n}{t} + \lceil \log_2 t \rceil$ injections and observations. Which strategy to use depends on the target application, and whether it supports shared folders or not. Further, as we discuss in Section 5.1, in some cases $\mathsf{dup}(\mathcal{V})$ reveals additional information, which allows $\mathcal{A}$ to speed up our generic injection strategies even further.

The exact wall-clock time of the attack depends, of course, on how often these requests are triggered, which varies across applications (we discuss some examples in Section 5.1). Note that the runtime of the attack is independent of the number of user accounts in $\mathcal{V}$. That is, since $\mathsf{dup}(\mathcal{V})$ is computed across all accounts, our attack essentially checks each password on *all* user websites at once. Further, our attack can be easily tweaked to find all matching passwords, instead of just one, by recursing into all branches that increase $\mathsf{dup}(\mathcal{V})$. In addition, our attack confirms to $\mathcal{A}$ if no password in P is in $\mathcal{V}$, which would be reflected by the fact that $\mathsf{dup}(\mathcal{V})$ does not increase after all candidate items have been injected.

5.1 Vulnerable Password Managers

In summary, the pre-conditions that an application must satisfy to be vulnerable to our generic attack are: (1) have support for vault-health metrics that contain the number of duplicate passwords; (2) the number of duplicate passwords must be computed across all vault entries; and (3) these metrics must be logged somewhere outside the victim's device (for example, in the application servers or in some organization administrator portal). We performed an analysis of all ten applications and found that LastPass, Dashlane, Zoho Vault, Enpass, Roboform, Keeper, and NordPass all satisfy these pre-conditions, and are thus vulnerable to our attack. We show in Figure 3 the breakdown of pre-conditions across all ten applications.

To experimentally validate our attack, we deployed proof-of-concept implementations of it against three of these vulnerable applications—LastPass, Dashlane, and Zoho Vault. We discuss these below, and refer readers to Appendix B for additional details of these.

In all three applications, updates to shared credentials are automatically synchronized across all clients with access to these, i.e., adversarial injections are automatically incorporated into $\mathcal{V}$. The main difference between each application is then the frequency of when clients log $\mathsf{dup}(\mathcal{V})$ to the applications servers. In LastPass, logs of $\mathsf{dup}(\mathcal{V})$ are triggered automatically after a client imports the updates to the shared credentials. In Dashlane and Zoho Vault, they are triggered once per day. A second notable difference is that LastPass supports shared folders, but Dashlane does not. Thus, $\mathcal{A}$ can use the binary-search and sequential injection strategies for the former and the latter, respectively. For Zoho Vault, the generic binary-search injection strategy is also feasible. However, in fact, application-specific details allow $\mathcal{A}$ to significantly speed up the attack, allowing them to recover *all* matching passwords with a *single* injection. Zoho Vault's vault-health logs consist of an array O of JSON objects, such that there is one such object for each entry $e \in \mathcal{V}$, containing metadata about the corresponding entry. In particular, each object contains a "`reused`" field, which maps to a boolean indicating whether e_{pw} is a duplicate or not. So, $\mathcal{A}$ can inject all candidate passwords in a single batch, scan O next time the logs are triggered, and identify whether the objects in O corresponding to the injected credentials are marked as a duplicate or not, thus revealing if that candidate password was already present in the user's vault. We discuss this optimized attack in more detail in more detail in Appendix B.

In all three applications, no interaction is required from U, and the attack only requires U to be logged into their account (for example, in their Chrome extension or web vault). Importantly, the victim need not have their vault open while the attack runs: if a user is logged in, background scripts check for updates, sync them to the vault, and trigger the requests to $\mathsf{dup}(\mathcal{V})$. As such, the attack can operate silently in the background while U is engaged in other tasks. Further, the attacks do not require the victim client to be quiet while the attack is running, and thus fall under the noisy device setting. So, $\mathcal{A}$ can just silently run an extended attack if need be, irrespective of the activity of the victim, until the passwords are found.

Attack implementation. We used an open-source list of common passwords [40] to gather our testing data, and confirmed that our attacks successfully recover the target string every time. Since our attacks are false-positive free—in particular, their correctness does not depend on the distribution nor

number of candidate passwords—we only tested our attacks on small, proof-of-concept workloads, and thus confirmed the correctness of the attack without overloading the applications' servers.

For each application, our experimental setup consisted of two test accounts on two different devices, representing the victim and the adversary, with a shared folder between them. The list of candidate passwords consisted of a subset of the aforementioned list of common passwords. We sampled uniformly at random a password from the list of candidate passwords, added it to the victim's vault from their machine, and ran the attacks from the adversary's machine. We set up a proxy server in the victim's machine to capture all outgoing network requests, and inspect the traffic from the victim to the server during the attacks. This simulates the information that would be learned by a malicious server. We explain additional details of our testing methodology in Appendix A.

Extension to a network adversary. For some applications, our attack may be able to be modified to assume a weaker network adversary instead of an eavesdropper. The key challenge is that a network adversary does not have access to the plaintext value of $\mathsf{dup}(\mathcal{V})$, as this is transmitted through encrypted channels. As such, we require an additional property from applications: that changes in the number of duplicate passwords result in changes in the size of the payload of the HTTPS requests that transmits the vault-health logs.[4]

From the applications vulnerable to an eavesdropping adversary, only Zoho Vault can be extended to a network adversary. Since there is a separate boolean string in O for each vault entry, $\mathcal{A}$ can leverage the subtle fact that the string `false` has one additional character than the string `true`, and thus the payload of the HTTPS request with O as its payload will fluctuate by 1 byte depending on whether each password is a duplicate or not. As such, our binary-search injection strategy can be used once again, by determining whether a batch of injections contains the target password or not based on this 1-byte difference.

6 Attacks from URL Icon Fetching

A feature of some password managers is to display a small identifying icon, such as a company logo, next to each vault entry. To do so, clients send a request to the server with the URL for the website, in plaintext, and receive back an image file with the icon. This clearly leaks the victim's URLs to an eavesdropping adversary (prior work [43] has pointed this out already for a few applications). However, as we show in this section, these requests can also be leveraged by a weaker *network* adversary to leak information about a victim's URLs.

This leakage arises from the fact that URL icons are fetched only *once*: any new entries for websites for which there is already a credential simply reuse the icon that is already on the client, and no new fetch request is sent to the server. These requests thus serve as oracle to determine whether a candidate URL w is in the victim's vault or not: $\mathcal{A}$ can inject an entry e such that $e_{url} = w$, wait for U's client to (automatically) synchronize this new credential, and observe if an icon fetch request is triggered or not; the latter indicates that w is already present in U's vault, since e is reusing the URL icon that is already downloaded. Importantly, note that this attack only requires the adversary to know whether a request to fetch the icon is *triggered or not*, and that, unlike our first attack, the actual (plaintext) payload of the request is not relevant. This information is also visible to a network adversary, who simply monitors whether a fetch request is triggered or not.

Analogous to our attack from Section 5, this basic oracle can be used to perform a dictionary attack on the websites stored on the victim's vault. In particular, the same sequential and binary-search injection strategies can be used, depending on whether the application supports shared folders or not.

6.1 Vulnerable Password Managers

In summary, the pre-conditions that an application must satisfy to be vulnerable to our generic attack are: (1) have support for URL icons; (2) fetch these URL icons from the application servers (instead of, e.g., storing them all in the client to begin with); and (3) re-use the stored icons across all vault entries. Our analysis of all ten applications revealed that Dashlane, 1Password, Enpass, Roboform, NordPass, and Proton Pass all satisfy these pre-conditions. We show in Figure 3 the breakdown of pre-conditions across all ten applications.

We experimentally validated our attack against Dashlane, using the Majestic ranking of top websites [4] to gather our testing data, using the same experimental setup as in Section 5, and confirmed that our attacks successfully recovers the target URL. Note that, as before, the correctness of the attack does not depend on the distribution nor number of candidate URLs, since icon fetch requests are deterministic, and as such it was sufficient to test on small workloads that did not overload their application servers.

7 Attacks from Storage-Saving Mechanisms

As discussed in Section 3, some password managers employ a variety of storage-saving techniques to reduce the size of their encrypted vault files. In this section, we show how two such mechanisms—file compression and attachment deduplication—lead to injection attacks against password managers. These have both been studied in other domains before (see Section 2), and here we show that they also lead to exploits in the context of password managers.

From the ten applications of our study, only KeePassXC supported storage-saving mechanisms. This is due to the fact

[4] We also technically require that request bodies are not compressed, but this is almost always true [41] (including for Zoho Vault).

that deduplication and compression are part of the specification of its underlying file format, KDBX 4, which is used by a variety of other password managers. Even though we implement our attacks on KeePassXC, we stress that our vulnerabilities target KDBX 4 itself, and thus any application that uses it (and that supports credential sharing) can be vulnerable to them.

Looking ahead, our attacks rely on the adversary being able to observe fluctuations in the *size* of the encrypted database file. Note that, if an application relies on third-party cloud providers to synchronize devices, a network adversary is sufficient: since each update re-uploads the entire encrypted file, the size of the payload of the HTTPS request can be used to detect fluctuations in the underlying file size. Further, since the attacks rely on very precise measurements, they operate in the quiet device setting, i.e., we assume the victim does not interact with their vault while the attack is running. However, our attacks benefit from the fact that modifications to shared entries are propagated automatically to the victim's vault, and thus our attacks can be run in a few minutes.

Background on KDBX 4. Broadly speaking, a database $\mathcal{V}$ serialized in KDBX 4 file format (the latest version of KDBX) consists of three main parts: an outer header H_{out}, an inner header H_{in}, and an XML payload I.

The file starts with H_{out}, which is unencrypted, and contains metadata about the database, such as cipher information, KDF parameters, whether the XML payload is compressed or not, etc. This is followed immediately by H_{in} and I (explained below), which are both under a single layer of encryption—using one of AES-CBC, Twofish-CBC, or ChaCha20—with a bespoke authentication mechanism; the details of the latter are unimportant for our attacks, so we omit them hereafter. Users can select their preferred encryption cipher in the database settings, with the default being ChaCha20. As usual, the encryption key K used is derived from PW, using either AES-KDF or Argon2. Then, H_{in} and I may optionally be compressed with gzip [17] before being encrypted, which is also a tunable parameter that is on by default.

The inner header H_{in} consists of two main parts: the encryption key K' used in the second encryption layer (more on this later), and an array A storing the binary content of all attachments across all entries in $\mathcal{V}$ concatenated together. Importantly, only one copy of each binary file is stored in A, even if the attachment is added multiple times to the database.

The XML payload I contains the user's data itself (excluding attachments, which are in H_{in}). For every entry $e \in \mathcal{V}$, there is a corresponding XML element in the payload, where each field of e is saved as an XLM subelement. There are two important types of subelements: (1) for every attachment in the list e_{attach}, if any, there will be a subelement with an integer denoting the index within A corresponding to the binary content of this attachment; (2) the subelement for e_{pw}, which stores an encrypted copy of e_{pw} using ChaCha20 and K'. There is no authentication on this inner encryption layer, since it is already under the authentication mechanism of the outer one. Further, the key K' is rotated after every modification to the database, and all passwords are re-encrypted. Other fields of e, such as e_{user} and e_{url}, are stored as unencrypted subelements; we denote by $\hat{e}$ all subelements of e, excluding e_{pw}, concatenated together.

Putting it all together, the serialization of $\mathcal{V} := (e_1, ..., e_m)$ has the following structure:

$$\mathcal{B} := H_{out} \,\|\, \mathsf{Enc}_K\Big(\mathsf{gzip}\big(H_{in} \,\|\, I\big)\Big)$$

where H_{in} and I have the following form:

$$H_{in} \,\|\, I = K' \,\|\, A \,\|\, \mathsf{Enc}_{K'}(e_{1,pw}) \,\|\, \hat{e_1} \,\|\, ... \,\|\, \mathsf{Enc}_{K'}(e_{m,pw}) \,\|\, \hat{e_m}$$

For simplicity, this notation omits low-level details such as XML tags and metadata that are not relevant to our attacks. We refer readers to documentation such as [12] for a more complete treatment.

Our attacks target the underlying file format itself, and any password manager that uses it may be vulnerable to them. Our attacks, however, require two assumptions regarding the target application and its users: that the application has support for cross-user sharing, and that compression is left on (note that official documentation from KeePass, the designers of KDBX, explicitly state that "it is not recommended to save databases without compression" [11].) Further, for clarity of presentation, we will assume that ChaCha20 is indeed the cipher used to encrypt the outer encryption layer, since it is the only stream cipher out of the three supported ones, but note that our attacks can be modified to work with the other ciphers.

The application on which we implement our attacks is KeePassXC, which is a newer, feature-rich port of the original KeePass. KeePassXC supports cross-user sharing (both for individual entries and folders) which are added as new XML elements in I, just like personal entries. Updates to these shared credentials are synchronized automatically if the victim has their vault unlocked (e.g., open in the background), and there is just a short time delay of 1-2 seconds.

7.1 Leakage from Attachment Deduplication

As discussed earlier, the KDBX 4 file format stores only one copy of the binary content of each attachment in A, even if it is received multiple times: whenever U adds a new attachment to their vault, it is compared against all attachments in A (by computing its checksum using a hash function), and added if and only if there is no match. If there is a match, the attachment pointer in the XML element for this entry simply refers back to the index of the first copy in A. Notably, deduplication occurs across both personal *and* shared attachments, which leads to a cross-user interaction that an adversary can leverage to leak information about the user's saved attachments.

Attack description. Our attack consists of a dictionary attack on attachments. That is, for a list of candidate attachments $W := \{w_1, ..., w_n\}$, the adversary wants to determine which w_i, if any, is in $\mathcal{V}$. We first note that there is a straightforward (but inefficient) attack. Assume that U and $\mathcal{A}$ have a shared entry e. Then, $\mathcal{A}$ can simply add each $w \in W$ to e_{attach}, one at a time, until some attachment does not increase the size of $\mathcal{B}$, which implies that it got deduplicated and thus is present in $\mathcal{V}$. This naive injection strategy is false-positive free, since deduplication is deterministic.[5] However, the number of injections scales linearly with $|W|$. We now describe a more complex injection strategy that an adversary can use for larger $|W|$. This second strategy leads to more false positives, and as such there is a trade-off between success rate and number of required injections.

Our refined strategy consists of a binary-search attack, which requires only $\lceil \log_2 n \rceil$ injections, analogous to that of Section 5: instead of injecting each attachment, one at a time, $\mathcal{A}$ can split W into two sets W' and W'' of equal size, and inject *all* attachments in W' in a single batch, followed by all attachments in W'' in a second one. Then, one of these injections will increase $|\mathcal{B}|$ by less than $|W^i|$, which means that some attachment in it got deduplicated. We can then recurse into this set, and repeat this attack iteratively until the target attachment is found.

A challenge with this approach is that, recall, the list of attachments is gzip-compressed with the rest of H_{in} and I, which adds noise to our measurements. In particular, it could be the case that the target file is in W', but all files in W'' are similar to each other. So, if the decrease in size from compressing all files in W'' is smaller than the target file, we would recurse into the wrong set. To address this, $\mathcal{A}$ can measure the compressibility of each injected list of files, and "penalize" the measurement according to this. Concretely, on every iteration of the attack, $\mathcal{A}$ computes $z' = \left|\mathsf{gzip}(w_1 \,\|\, ... \,\|\, w_{n/2})\right|$ and $z'' = \left|\mathsf{gzip}(w_{n/2+1} \,\|\, ... \,\|\, w_n)\right|$ locally, and recurses into W' if

$$\mathcal{B}_2/(\mathcal{B}_1 + z') < \mathcal{B}_3/(\mathcal{B}_1 + z'')$$

and into W'' otherwise; $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ respectively denote the encrypted database before both injections, after the first injection, and after the second injection.

After the attack is over, $\mathcal{A}$ can confirm whether the attack was successful or not by making an additional injection with just the found attachment: if the guess is correct, $\mathcal{B}$ will have no increase in size (except, potentially, for a negligible number of bytes due to noise from the re-encryptions.)

Attack implementation. We implemented both the naive and the binary-search attacks against KeePassXC, and confirmed that the former successfully recovers the target attachment every time. For the latter, we ran a sequence of experiments to estimate its probability of success. Our experimental setup consisted of two testing environments: (1) using real KeePassXC clients (version 2.7.6), and (2) a local re-implementation of all client-side operations relevant to our attacks, to simulate the real setting.

[5] The only edge case in which this attack may fail is if the candidate files are very small, such as less than 50 bytes. This is due to the fact that minimal noise is generated by the re-encryption of passwords after every save. In this case, however, $\mathcal{A}$ can simply inject each w_i multiple times, and compute the average file size, which mitigates the noise.

	Enron	**$\lvert w\rvert$=10KB**	**500KB**	**1MB**
$\lvert W\rvert = 32$	92	74	44	47
128	81	53	34	38
512	52	29	18	8

Figure 4: Experimental success probability of our dictionary attack on attachments exploiting deduplication, for real-world files (first column), and for synthetic test files (each file is of some random size between between 1 and $|w|$, and contains sequences of repeated characters). Each row represents a number of candidate files. These probabilities can be amplified via repetition.

This dual-setup approach is a common strategy used to evaluate deployed systems [6, 20, 44]. The real environment allows us to confirm the overall correctness and implementation of our attacks, while the simulated environment allows us to compute an empirical estimate of the probability of success. For the latter, we can run a high volume of trials in a reasonable amount of time, and without overloading the cloud-service provider. In addition, we can ensure that the trials are independent, by using the same starting state across each. We discuss our experimental setup in more detail in Appendix A.2.

Equipped with this setup, we used three types of datasets to gauge the attack's success rate on different workloads. The first consisted of the Enron corpus [2], which is a public dataset of real-world emails. This dataset helped us evaluate our attack on practical targets. The other two datasets, generated locally by us, consisted purely of synthetic data for benchmarking purposes: a set of random files, all of the same size; and a set of files of varying sizes, and composed of substrings of repeated characters. The reasoning behind these datasets is that random bytes and equal file sizes minimize the effects of gzip noise (since random data is less compressible), while substrings of repeated characters and different file sizes maximize gzip noise.

For each data set, we ran 100 trials of the attack, and recorded the fraction of these in which the target attachment was successfully recovered. Each trial consisted of sampling a fresh set of candidate files of the appropriate type, choosing one file at random, adding it to the victim's vault in a personal entry, and running the attack. For the dataset consisting of files of the same size and all random bytes, our attacks succeed with 100% probability for all file sizes. The results for the other two datasets are displayed in Figure 4. As our

experiments show, our attack succeeds with high probability. Further, as explained earlier, the adversary can confirm whether the output is correct or not, and re-run the attack if needed until the correct attachment is found, which essentially gets rid of false positives.

7.2 Leakage from Compression

The second vulnerable mechanism we identified in KDBX 4 is the fact that $H_{in} \| I$ is gzip-compressed before it is encrypted. The (oversimplified) way in which gzip works is that repeated substrings are replaced with short pointers to their prior occurrences, if any, which gets rid of redundancy in the payload. The resulting sequence of pointers and (unmatched) characters is then serialized using Huffman coding to yield the compressed bytestring.

Since I (resp. H_{in}) contains both personal *and* shared entries (resp. attachments), the first step of the gzip algorithm leads to a cross-user interaction that an adversary can exploit: if fields in the injected credentials match some field in U's personal credentials, the length of the resulting encrypted database will be shorter than if they do not match, as in the former case gzip will replace the injected content with a short pointer to U's private content.

Attack description: dictionary attack. Our attack consists of a dictionary attack, i.e., for a list of candidate items $W := \{w_1, ..., w_n\}$, $\mathcal{A}$ wants to determine which w_i, if any, is in $\mathcal{V}$. The items of interest consist of login information, except for passwords (e.g., usernames or URLs), and contents of attachments. The reason why passwords are not recoverable is due to the fact that these are under an additional layer of encryption, which prevents matching passwords from being deduplicated by gzip.

One possible attack approach is to simply inject each candidate string, one at a time, and return the one that leads to the smallest increase in database size. This attack, however, suffers from low accuracy, since it compares compressibility across *different* candidate strings. Since each string contains different characters, the Huffman coding step of gzip may lead to false positives: it could be the case that an incorrect candidate string is composed of characters that appear very frequently in the database, which leads to shorter Huffman codes and the appearance of a more compressible string.

Instead, drawing inspiration from other compression-based attacks such as [26, 31, 49], we use *two* injections per candidate w—the first containing w itself and the second a *scrambled* version of w—and compare their relative compressibility. Thus, each pair of injections consist of the same characters, which reduces the effects of Huffman coding. (To our knowledge this implementation of the generic "two-tries" method is novel.) More concretely, for each w_i, $\mathcal{A}$ first injects w_i through a shared entry e (depending on the type of W, e.g., by setting either $e_{url} = w_i$ or $e_{user} = w_i$). Then, $\mathcal{A}$ records $n'_i = |\mathcal{B}|$. This is followed by a second injection, where $\mathcal{A}$ replaces w_i in the relevant field with a random permutation of its characters, and records the updated $n''_i = |\mathcal{B}|$. Finally, $\mathcal{A}$ returns whichever w_i yield the maximum value for $|n''_i - n'_i|/|w_i|$.

The intuition behind the strategy is that, in the first injection, the correct string gets compressed in its entirety, while incorrect strings only get partially compressed. Then, the second injection serves as a baseline measurement to gauge the "worst-case" compression of a string with the same characters as w_i, to control for the noise from the Huffman coding. So, it may be the case that some string leads to the smallest total increase in $\mathcal{B}$ but, in fact, this is also the case when the string is replaced by a scrambled version of it. This confirms that the decrease in size comes from the short Huffman codes and not from redundancy with the rest of the database (otherwise, the second string would "break" the effects of compression).

As in the prior attack, $\mathcal{A}$ can confirm whether the attack was successful or not by, for example, re-injecting the found string, prepended with a pad of 32K bytes (the length of zlib's search window), and confirming that output file increased by the size of the string.

We note that our dictionary attack can potentially be extended to a *byte-by-byte* recovery attack, where the list of candidate guesses is not known a priori. To do so, $\mathcal{A}$ can instead employ a CRIME-style attack [49]. In such attacks, $\mathcal{A}$ first has knowledge of a prefix p, of length at least three bytes,[6] that precedes the secret, which serves to "bootstrap" the attack. To guess the first character, the high-level idea is that $\mathcal{A}$ tries all possible values z_i for it by injecting $p \| z_1$ followed by $p \| z_2$, and so on. Then, all incorrect guesses will get compressed by only $|p|$, but the correct guess will match an *additional* character, leading to a slightly smaller ciphertext. The attacker proceeds in this fashion, one byte at a time, until eventually the entire secret is recovered. If the adversary is extracting information in attachments, the known prefix p can be part of a document's template; for example, if recovering the salary in a contract, p may be "Salary: ". Conversely, if the adversary is recovering URLs or usernames, we can take advantage of the structure of KDBX 4, and set p to be the *XML tags* of the field. For example, if recovering a username, $p =$ `<Key>UserName</Key><Value>`. Further, $\mathcal{A}$ could recover a field from a *specific* credential, by appending additional information to p. For example, to recover the username specifically for `target-site.com`, $\mathcal{A}$ can use $p =$ `<Key>URL</Key><Value>target-site.com</Value>` ... `UserName</Key><Value>`. As before, injections are performed by updating the URL, username, or attachment fields of a shared entry, this time with $p \| z_i$ as the payloads. Exploring this extension further is outside the scope of our work, as such attacks rely on lower-level details of compression algorithms.

[6] gzip does not compress matching substrings unless they are at least four bytes long.

	Websites	Usernames	$\|w\| = 5$	10	15	20
$\|W\| = 4$	99	80	89, 61	100, 74	100, 86	100, 88
10	97	66	81, 51	98, 59	100, 57	100, 63
25	92	54	64, 30	97, 33	100, 48	100, 44
50	89	46	50, 17	93, 28	100, 29	100, 38
100	84	24	39, 8	92, 13	100, 16	100, 14

Figure 5: Experimental success probability of our dictionary attack on usernames and URLs exploiting compression, for real-world files (first two columns), and for synthetic test files (last four columns); the left and right values denote, respectively, when the strings consist of random bytes and are all of length w, and when they have repeated substrings and vary in length between 1 and $|w|$. These probabilities can be amplified via repetition.

Attack implementations. We experimentally verified our dictionary attack against KeePassXC, using URLs and usernames as examples of items of interest. An additional complication about KeePassXC is that, as explained earlier, every database save rotates K', re-encrypts all passwords, and updates metadata (for example, a timestamp indicating the last modification time of the shared entry). This results in a small number of bytes of noise in the compression side-channel. Given that our target items are only a few bytes long, such as URLs and usernames, this noise is noticeable. So, we use a simple refinement from [61]: inject each candidate item t times instead of just once, compute the average value of the resulting $|\mathcal{B}|$, and use this in the measurements. Of course, the larger t is, the higher the probability of success will be. $\mathcal{A}$ can pick an appropriate value based on the context of the attack (for example, the size of W, how long the victim will be using their device, etc).

We tested our attack in the same dual-setup as in Section 7.1 and using the same three types of workloads; the datasets of real-world data consisted of a list of the Majestic ranking of top websites [4], and a corpus of common usernames compiled from various data breaches [40]. The results of our experiments on usernames are displayed in Figure 5, which correspond to the success rate across 100 independent trials of each workload, using $t = 10$ for each injection.

8 Mitigations and Responsible Disclosure

We discuss mitigations for our attacks in this section, as well as the results of our responsible disclosure with the vendors directly affected by our work. The patches adopted by these may serve as inspiration for other applications that are vulnerable to our attacks.

The most immediate mitigation to our attacks from Section 5 would be to remove shared credentials from the computation of vault-health metrics that are logged to the servers, in order to disable the injection side-channel. This, however, would result in a loss of information for users, as they would no longer be able to detect duplicate passwords present in (non-malicious) shared credentials. Other mitigation approaches would depend on the exact purpose of these metrics, which is opaque to us. For example, if the metrics are only used for client-side computation and stored on the server, clients can encrypt these locally before exporting them. Conversely, if the metrics are used to compute server-side statistics, applications could use privacy-preserving aggregate statistics frameworks such as [15].

Our attacks from Section 6 against a network adversary can be mitigated by fetching icons every time a new credential is added to the vault, even if the URL is a duplicate. Note that clients can potentially still store only one copy of each icon in the client side, and the requests can be repeated just for the sake of hiding fetch patterns. To also hide the URLs from an eavesdropping adversary, applications can use private-information retrieval (PIR) schemes [14] to retrieve icons without revealing the URL in question.

A direct mitigation to our attacks from Section 7 is for applications to, of course, disable storage-saving mechanisms. However, this could result in a prohibitive blowup in their storage footprint. By definition, disabling deduplication doubles the cost of storing repeated attachments. The cost of disabling compression depends on the file format and the underlying user data; for highly structured file formats like XML, the increase is particularly noticeable. Local tests on example KeePassXC databases resulted in 5-6x increases in size.

Another mitigation approach is to confine the storage-saving mechanisms to subsets of the application data that are within the same trust context, which would disable the injection channel. For deduplication, this translates to deduplicating files separately for every shared folder. More work is required to devise an analogous solution for compression. A different mitigation approach for compression would be to isolate sensitive fields within the database by encrypting them under a second layer of encryption, which prevents similar data from matching with these fields. This is analogous to how KeePassXC protects passwords, as described in Section 7, and indeed they could consider encrypting other fields in a similar fashion. Yet another approach could be to add padding or noise to the database before encrypting it, in order to obfuscate the real size of the database. Adversaries can use statistical techniques to adapt to such mitigations, however, as has been the case in other contexts such as network traffic fingerprinting (see, e.g., [18]).

Responsible disclosure. The first version of our study analyzed just four applications (LastPass, Dashlane, Zoho Vault, and KeePassXC), who proceeded to deploy mitigations, which we describe below. A second round of disclosures is still ongoing, since we recently expanded the scope of our study. We will update the paper once we complete these.

LastPass adopted our suggested mitigation of separating vault-health metrics between personal and shared credentials,

which disables the injection channel. They released an initial implementation of this fix in version 4.129.0, removing shared folders from the vault-health logs, as these lead to the most severe variant of our attack. Removing individually shared credentials from the logs is more technically challenging—and individual credentials lead to a less practical version of our attack—and thus has been deferred to later in their roadmap; their projection is to release this fix by the end of the year, which would complete a full mitigation to our attack.

Zoho Vault plans to adopt a similar fix, by implementing an option to separately compute vault-health metrics on personal passwords as of version 4.0. Dashlane opted for a partial mitigation instead, namely, increasing their rate limits on the sharing endpoints "as much as possible". Given the fact that their vault-health metrics are only logged once per day, their tight sharing limits significantly affect the practicality and runtime of the attack. In addition, the resource limits on their web application and extensions prevent an adversary from sharing an unlimited number of credentials with a victim, which increases the runtime of the attack even more. As part of the disclosure, they informed us that incorporating shared passwords is a core feature of their vault-health metrics, and thus removing shared passwords would represent a notable disruption to this feature.

Then, to mitigate our URL icon fetching attack, Dashlane implemented a new feature as of version 6.2415 that allows users to turn off fetching credential icons, which disables the side-channel for both an eavesdropping and network adversary, and is thus a full mitigation to our attack; details of this new feature can be found at [16]. In addition, they migrated their icon fetching tool to a new endpoint (`api.dashlane.com`), which is used by multiple parts of their application logic. As such, this would make it significantly more challenging for a network adversary to use traffic analysis techniques to identify whether an icon fetch request is included in the traffic sent to this top-level endpoint, due to the high amount of noise from the other requests sent to this endpoint.

To address our attack on attachment deduplication, KeePassXC adopted our suggested mitigation of deduplicating files separately for every shared folder, which disables the injection side channel. Then, to address our compression-based attacks, they modified their file format by, every time the database is saved, picking a random length between 64 and 512 bytes, generating a random array of this length, and including this in a "custom data" field of their file format. We note that this is only a partial mitigation, as an adversary can potentially use statistical techniques to bypass the noise; this, however, would require a significantly higher number of injections. Both fixes were promptly implemented by the KeePassXC team, and have since rolled out as part of version 2.7.

9 Conclusion

We introduce a new threat model for password managers in this work, which we exemplify via four general attacks, using ten applications as case studies. Our attacks suggest the need to rethink certain aspects of password manager design, and of E2EE applications more broadly. We highlight some takeaways in this section.

Our attacks from Section 5 are symptoms of the more general pattern of exporting application state that is a function of both personal *and* externally-received data, which can potentially open the door for injection attacks. Thus, a guiding principle for E2EE application designers is to *separate data according to the trust assumptions of their system*. Our attack from Section 6 is an example of the broader pattern of fetching content from external sources in a *state-dependent way*, which an adversary can potentially exploit by injecting payloads and seeing how this affects subsequent fetches. Thus, a guiding principle for application designers is for client-server communication to be *as resource-specific as possible, and to not depend on the results of prior operations*. Lastly, our attacks from Section 7 serve as an example of the tensions between security and performance. Since storage-saving mechanisms get rid of redundancy in application files, techniques of this form naturally pave the way for potential injection attacks. As such, more work is required to understand how to balance storage costs and security, particularly in the context of compression and file deduplication, and to devise frameworks that help explore these trade-offs in a principled way.

We disclosed our attacks to the vendors affected by our work, who deployed fixes to address these. These mitigations, however, are bespoke solutions for the attack vectors presented in this work; a central direction for future research is to design general-purpose detection and mitigation techniques against injection attacks, and to deepen our understanding of this threat model.

Acknowledgements

This work was supported in part by NSF grants CNS-1704296 and CNS-2120651.

References

[1] Charles Proxy. https://www.charlesproxy.com/.

[2] Enron Email Dataset. https://www.cs.cmu.edu/~enron/.

[3] KeePassXC. https://github.com/keepassxreboot/keepassxc.

[4] The Majestic Million. https://majestic.com/reports/majestic-million.

[5] 1Password. Zero-knowledge encryption. https://1password.com/features/zero-knowledge-encryption/.

[6] Matilda Backendal, Miro Haller, and Kenneth G Paterson. Mega: malleable encryption goes awry. In *IEEE S&P*, 2023.

[7] Bitwarden. How end-to-end encryption paves the way for zero knowledge - white paper. https://bitwarden.com/resources/zero-knowledge-encryption-white-paper/.

[8] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *ESORICS*, 2010.

[9] Michael Carr and Siamak F Shahandashti. Revisiting security vulnerabilities in commercial password managers. In *ICT Systems Security and Privacy Protection*, 2020.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[11] KeePass Help Center. Database Settings. https://keepass.info/help/v2/dbsettings.html.

[12] KeePass Help Center. KDBX 4. https://keepass.info/help/kb/kdbx_4.html.

[13] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. Cracking-resistant password vaults using natural language encoders. In *IEEE S&P*, 2015.

[14] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 1998.

[15] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[16] Dashlane. Hide the icons in your dashlane login list. https://support.dashlane.com/hc/en-us/articles/17909345422354-Hide-the-icons-in-your-Dashlane-login-list.

[17] Peter Deutsch. Gzip file format specification version 4.3. Technical report, 1996.

[18] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE S&P*, 2012.

[19] Enpass. Syncing and accessing Enpass data between devices. https://support.enpass.io/app/sync/sync_and_access_enpass_data_on_all_devices.htm.

[20] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. Injection Attacks Against End-to-End Encrypted Applications. In *IEEE S&P*, 2024.

[21] Michael Fagan, Yusuf Albayram, Mohammad Maifi Hasan Khan, and Ross Buck. An investigation into users' considerations towards using password managers. *Human-centric Computing and Information Sciences*, 2017.

[22] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. Hey, you, get off of my clipboard: On how usability trumps security in android password managers. In *Financial Cryptography and Data Security*, 2013.

[23] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple {iMessage}. In *USENIX Security*, 2016.

[24] Paolo Gasti and Kasper B Rasmussen. On the security of password manager database formats. In *ESORICS*, 2012.

[25] Conor Gilsenan, Fuzail Shakir, Noura Alomar, and Serge Egelman. Security and privacy failures in popular 2fa apps. In *USENIX Security*, 2023.

[26] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: reviving the CRIME attack. *Unpublished manuscript*, 2013.

[27] Maximilian Golla, Benedict Beuscher, and Markus Dürmuth. On the security of cracking-resistant password vaults. In *CCS*, 2016.

[28] Google and Harris Poll. The United States of P@ssw0rd$. https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/PasswordCheckup-HarrisPoll-InfographicFINAL.pdf, October 2019.

[29] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *CCS*, 2011.

[30] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE S&P*, 2010.

[31] Mathew Hogan, Yan Michalevsky, and Saba Eskandarian. DBREACH: Stealing from databases using compression side-channels. In *IEEE S&P*, 2023.

[32] Troy Hunt. Passwords evolved: Authentication guidance for the modern era. https://www.troyhunt.com/passwords-evolved-authentication-guidance-for-the-modern-era/, July 2017.

[33] KeePass. Keepass Password Safe. https://keepass.info/.

[34] John Kelsey. Compression and information leakage of plaintext. In *International Workshop on Fast Software Encryption*. Springer, 2002.

[35] LastPass. An encryption model that prioritizes your privacy. https://www.lastpass.com/security/zero-knowledge-security.

[36] LastPass. Manage shared folders. https://support.lastpass.com/s/document-item?bundleId=lastpass&topicId=LastPass/c_about_shared_folders.html.

[37] LastPass. Notice of Recent Security Incident. https://blog.lastpass.com/2022/12/notice-of-recent-security-incident/, December 2022.

[38] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The Emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security*, 2014.

[39] Sanam Ghorbani Lyastani, Michael Schilling, Sascha Fahl, Michael Backes, and Sven Bugiel. Better managed than memorized? Studying the impact of managers on password strength and reuse. In *27th USENIX Security*, 2018.

[40] Daniel Miessler, Jason Haddix, and g0tmi1k. SecLists. https://github.com/danielmiessler/SecLists.

[41] Avi Networks. Http compression. https://avinetworks.com/glossary/http-compression/.

[42] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. Sigstore: Software signing for everybody. In *CCS*, 2022.

[43] Sean Oesch and Scott Ruoti. That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers. In *USENIX Security*, 2020.

[44] Kenneth G Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a secure messenger, 2023.

[45] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *SOUPS*, 2019.

[46] Milen Petrov. Android password managers and vault applications: Data storage security issues identification. *Journal of Information Security and Applications*, 2022.

[47] Reddit. I made a comparison table to find the best password manager. https://www.reddit.com/r/Passwords/comments/17f73pa/i_made_a_comparison_table_to_find_the_best/.

[48] Reddit. What password manager do you use? https://www.reddit.com/r/cybersecurity/comments/12jd9ec/what_password_manager_do_you_use/.

[49] Juliano Rizzo and Thai Duong. The crime attack. In *ekoparty security conference*, 2012.

[50] CYFOR Secure. How long does it take to detect a cyber attack? https://cyforsecure.co.uk/how-long-does-it-take-to-detect-a-cyber-attack/, December 2022.

[51] Security.org. Password Manager Industry Report and Market Outlook (2023-2024),. https://security.org/digital-safety/password-manager-annual-report/, September 2023.

[52] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *USENIX Security*, 2014.

[53] Ben Stock and Martin Johns. Protecting users against XSS-based password manager abuse. In *CCS*, 2014.

[54] Dropbox Security Team. How we handled a recent phishing incident that targeted Dropbox. https://dropbox.tech/security/a-recent-phishing-campaign-targeting-dropbox, November 2022.

[55] KeePassXC Team. KeepassXC: User guide. https://keepassxc.org/docs/KeePassXC_UserGuide.

[56] Andreas Tuerk. To stay secure online, Password Checkup has your back. https://blog.google/technology/safety-security/password-checkup/, October 2019.

[57] https://github.com/lgg. Awesome KeePass Projects. https://github.com/lgg/awesome-keepass.

[58] Zack Whittaker. Norton LifeLock says thousands of customer accounts breached. https://techcrunch.com/2023/01/15/norton-lifelock-password-manager-data/, January 2023.

[59] Min Xu, Armin Namavari, David Cash, and Thomas Ristenpart. Searching encrypted data with size-locked indexes. In *USENIX Security*, 2021.

[60] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: the power of file-injection attacks on searchable encryption. In *USENIX Security*, 2016.

[61] Dionysios Zindros and Dimitris Karakostas. Practical new developments in the breach attack. *Black Hat Asia*, 2016.

A Testing Methodology and Analysis

We explain the testing methodology for our attacks in this section, where we include a summary of the steps shared with vendors that allowed them to reproduce the attacks.

General setup. For each analyzed application, we first created two separate accounts for the service. The devices used were laptops running macOS Monterrey. If relevant for the application, we additionally created an organization under which both accounts were registered. Then, we established a shared folder (or individual credentials, if the application does not support folders) between both accounts.

A.1 Attacks from Sections 5 and 6

To simulate the view of both a compromised server and network adversary, we used Charles Proxy [1] to establish a proxy server on the victim's device. This allowed us to intercept both the encrypted HTTPS traffic, and enable SSL proxying to intercept the unencrypted traffic that would reach the application servers.

Our testing data consisted of an open-source list of common passwords [40] and the Majestic ranking of top websites [4], for the attacks in Section 5 and 6, respectively. We explain the main steps for our attack from Section 5 below; the steps for the attack in Section 6 are analogous, except that we inject URLs instead of passwords.

To implement the attack, we first sampled uniformly at random a password from this list, and added it to the victim's vault as the target password. We then sampled at random various subsets of passwords of increasing sizes (1 through 512, in increasing powers of 2), which represented the list of candidate passwords. Lastly, we ran the attack from the adversary's account, following the steps described in Section 5. At a high-level, this consisted of creating a batch of shared passwords in the adversary's account, waiting for the next request from the victim's account reporting the updated number of duplicates (which was intercepted by our proxy server), and repeating these steps, until completion of the attack.

To determine the network delay for each log of duplicates, as reported in Section 5, we compared (1) the timestamp from the initial request for the upload of the shared credentials, against (2) the timestamp of the request that logged the number of duplicate passwords. We computed the average delay between each injection. Then, the total time to run each attack is simply $t \cdot n$, where t is the number of iterations the attack requires, and n is the experimental delay between logs of duplicates.

Both attacks succeed with 100% probability by design: computing the number of duplicates in the vault and fetching URL icons are both deterministic, noise-free operations. Thus, there was no need for an experimental estimate of the attack's correctness. Indeed, our attack implementations succeeded every time.

A.2 Attacks from Section 7

To simulate the view of a compromised encrypted vault, it sufficed to place the encrypted database file in a local folder in the victim's device, and measure its size directly.

We verified the general correctness of the attacks by implementing them using real KeePassXC clients, and then computed an empirical estimate of their probability of success on a local simulation (described below). We used various data sources as testing data for the attacks, as described in Sections 7.1 and 7.2. For each, we first added the target item to the victim's device. Then, we ran the attack from the adversary's device as explained in the relevant sections, by iteratively adding credentials to the shared folder with injected items and measuring the updated size of the victim's encrypted vault. In this way, we were able to confirm the presence of the deduplication and compression mechanisms.

Statistical analysis. We implemented a local simulation of the core operations of the KeePassXC client that are relevant to our attacks, which allowed us to compute the statistical error rates of our attacks. Concretely, our simulation received as input an initial KDBX 4 database, and provided an interface for adding shared credentials to the database, which would update the vault in-place, in a way that is consistent with updates from real clients. This consisted of two steps: (1) Modifying the plaintext database, by adding the shared credential as a new XML element to the payload, adding new attachments to the inner header (if any), and updating metadata; and (2) Re-encrypting the new plaintext database, by rotating the secret key in the inner header, and compressing-then-encrypting the inner header and XML payload with this key.

Our implementation of this simulated environment was guided by KeePassXC's open-source codebase, in addition to manual comparison of decrypted KDBX 4 databases before and after a shared credential was added, which revealed all modifications that resulted. In particular, this is derived from files `src/format/Kdbx4Writer.cpp` and `src/format/KdbxXmlWriter.cpp` in KeePassXC's codebase [3]. We tested the consistency of this simulation by adding shared credentials via both our local implementation and real KeePassXC clients,

and verifying that the resulting database states are equivalent in both cases.

Equipped with this simulation, we ran the attacks (following the steps described in Section 7) in a single device, to be able to compute a high number of trials without overloading the application servers. We used the interface provided by our simulation to add credentials to the vault (which mocks the injected credentials), followed by local measurements of the resulting file size (which mocks the adversary's view of the compromised encrypted vault). This allowed us to empirically estimate the noise present in our attacks, by repeating each experiment many times on the same starting state, and tallying the number of successful attempts (see Section 7 for more details, and the reported error rates).

B Details of Case Studies from Section 5

We explain additional details for the case studies of our attacks from Section 5 in this section.

Case study #1: LastPass. LastPass's Chrome extension and web vault log vault security metrics via POST requests to the endpoint `lastpass.com/lmiapi/users/me/security/score`.[7] The payload of this request, among other things, maps the string "`numallduppasswords`" to the number of duplicate passwords in the vault, i.e., $\mathsf{dup}(\mathcal{V})$. Importantly, this metric only quantifies how many passwords themselves are reused, and not across how many accounts. That is, if there are already two copies of a password in a vault, adding a third copy does not increase this metric. As such, our attack on LastPass requires the additional assumption that U only has one copy of the target password in their vault at the start of the attack.[8]

The POST requests to `lastpass.com/.../score` are triggered after every modification to any entries in $\mathcal{V}$. In particular, a request is sent as soon as U's client automatically imports the updates to the shared folder F. Synchronization of changes to shared folders is very frequent. Indeed, LastPass claims that changes are "synchronized automatically and propagate to everyone with whom the folder has been shared" [36]. In practice, there is a short time delay, which represents the time the adversary has to wait between each of the $\lceil \log_2 n \rceil$ injections. We measured this to be between 3 to 5 minutes on average.

Case study #2: Dashlane. Dashlane's web vault and Chrome extension log their vault security metrics via POST requests to `styx.data.dashlane.com/v1/event/user`. The payload of this request contains the string "`passwords_reused_count`", which corresponds to $2 \cdot \mathsf{dup}(\mathcal{V})$. Unlike LastPass, this metric quantifies the number of *accounts* with reused passwords, i.e., we do *not* need to assume that U's vault only contains one copy of the target password. More generally, the attack requires no assumptions about the (lack of) activity by U,[9] and thus falls under the noisy device setting.

Case study #3: Zoho Vault. Zoho Vault's web vault logs their vault security metrics via POST requests to the endpoint `vault.zoho.com/api/rest/json/v1/dashboard/sendAssessmentDetails`. The payload of this request contains an array O of JSON objects, with the format described in Section 5.1. By counting the number of objects such that "`reused`" maps to `true`, the adversary can identify $\mathsf{dup}(\mathcal{V})$, and thus run the attack using the binary-search injection strategy. However, Zoho Vault is revealing much more granular information: $\mathcal{A}$ learns whether the password of each *specific* entry is a duplicate or not, instead of just the total number of duplicate passwords. So, the adversary can simply inject all candidate passwords in a *single* batch, and look for the `reused` field of the objects in O that correspond to the injected credentials.

The challenge with this approach is identifying which specific objects correspond to which specific injected credentials. To do so, the adversary can leverage an additional fact: upon creation, each credential is assigned a unique string identifying it, which is also stored in each credential's object in O. This ID is preserved after a credential is shared. So, the adversary can (1) locally create a credential for each candidate password, (2) note the assigned ID for each, and (3) look for these IDs in O after sharing the credentials. This allows the adversary to map every injected credential to its corresponding object in O, as desired.

Unlike the prior attack on LastPass, the correctness of this attack does not depend on U having only one copy of the target password in their vault. More broadly, our attack is robust to any kind of external noise, and thus falls under the noisy device threat model. We note that our extension of this attack to a network adversary does require that the device is free of external noise at every recursive round of the binary-search injection strategy, in order for fluctuations in $|O|$ to correspond only to changes in the `reused` flags of injected passwords. Importantly, however, the device need not be quiet for the *entire* attack: if $\mathcal{A}$ notices that external changes also took place, they can simply repeat this round of the attack. As such, the attack as a whole falls under the noisy device setting.

[7] This is only the case for users with premium accounts (any one of Families, Teams, Premium, and Enterprise plans, and the trial versions of these).

[8] Similarly, the basic attack requires a trivial refinement: if $P_{found} = P'$ in step (5), $\mathcal{A}$ needs to first delete or overwrite the entries they just injected, as otherwise there would already be a copy of the target password in the vault (from their prior injection).

[9] Unlike LastPass, when the metric increases $\mathcal{A}$ can distinguish false positives from legitimate matches by unsending the candidate, and seeing if the metric decreases again.