Web-based Attacks on Host-Proof Encrypted Storage

Karthikeyan Bhargavan
INRIA

Antoine Delignat-Lavaud
ENS Cachan

Abstract

Cloud-based storage services, such as Wuala, and password managers, such as LastPass, are examples of so-called *host-proof* web applications that aim to protect users from attacks on the servers that host their data. To this end, user data is encrypted on the client and the server is used only as a backup data store. Authorized users may access their data through client-side software, but for ease of use, many commercial applications also offer browser-based interfaces that enable features such as remote access, form-filling, and secure sharing.

We describe a series of web-based attacks on popular host-proof applications that completely circumvent their cryptographic protections. Our attacks exploit standard web application vulnerabilities to expose flaws in the encryption mechanisms, authorization policies, and key management implemented by these applications. Our analysis suggests that host-proofing by itself is not enough to protect users from web attackers, who will simply shift their focus to flaws in client-side interfaces.

1 Host-Proof Web Applications

The remarkable increase in website attacks in recent years and the consequent loss of sensitive user data has motivated a security-focused redesign of web applications where data is now routinely stored in encrypted form on web servers and only decrypted when needed. This architecture protects users from malicious hackers who may steal a database from the server but will not be able to decrypt it. However, it does not prevent data theft by disgruntled employees, who may have access to the decryption keys. Moreover, since the server application has access to decrypted data and is itself accessible over the web, any vulnerability in its code risks leaking user data to a web-based attacker through standard attacks like cross-site request forgery (CSRF).

Server-side encryption may be adequate for casual websites, but users of cloud-based storage and privacy-sensitive applications such as password managers demand stronger security guarantees. For example, when the storage service Dropbox [5] revealed that some of its employees could read user files, it was widely criticized for violating user privacy [15]. Conversely, when the password manager LastPass [7] announced that its servers may have been compromised [16], public reaction was mitigated because of the *host-proof* [6] design that LastPass implements against this class of attacks.

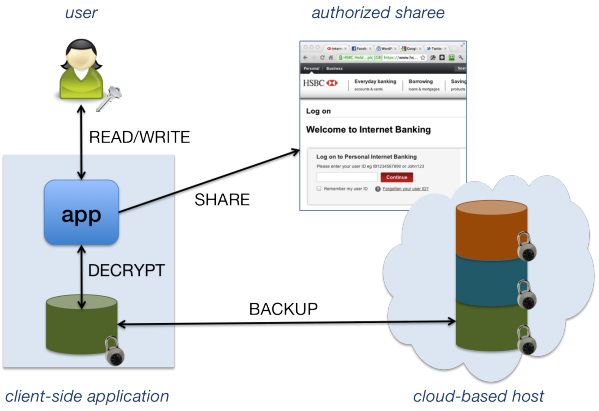

Figure 1: Host-proof web application architecture

A host-proof web application follows the architecture depicted in Figure 1. Personal data is encrypted on the client using a key or passphrase known by the user, while the web server only acts as an encrypted data store. The full functionality of the application is implemented in the client-side app, which performs all encryptions and decryptions, backs up the database to the server and, only when the user authorizes it, shares decrypted data with other users or websites. Since the server never sees unencrypted data (nor any decryption key, ideally), even if an attacker steals the database from the server, he cannot recover the plaintext without substantial computational effort to brute-force through every user's decryption key.

This design is sometimes called *cryptographic cloud storage*, and may use cryptographic mechanisms that enable some operations on encrypted data (such as search) [28]. The design is also sometimes misleadingly called *zero-knowledge* [3, 11]. We use the more neutral term *host-proof* to simply mean that the security of the application does not depend on trusting the server.

We consider two classes of host-proof web applications: cloud-based storage and password managers.

- Storage services, such as Wuala [12] and SpiderOak [11], offer a remote encrypted backup folder synchronized across all of the user's devices. The user may explicitly share specific sub-folders or files with other users, groups, or through a web link.

- Password managers, such as LastPass [7] and 1Password [1], offer to store users' confidential data, such as login credentials to different websites, or credit card numbers. When the user browses to a website, the password manager offers to automatically fill in the login form with a username and password retrieved from the encrypted database. The password database is backed up on a server and synchronized across the user's devices.

These applications differ from each other in their precise use of cryptography and in their choice of web interfaces. Tables 1 and 2 summarize the main features of a series of host-proof applications. In addition to those mentioned above, these tables include the cloud storage applications BoxCryptor [2] and CloudFogger [4] that add client-side encryption to non host-proof cloud services such as Dropbox. They also include the password managers RoboForm [10], PassPack [9], and Clipperz [3]. For each application, Table 1 notes the cryptographic algorithms and mechanisms used, while Table 2 summarizes the web interfaces offered.

Despite differences in their design and implementation, the common security goals of host-proof encrypted storage applications can be summarized as follows:

- *confidentiality*: unshared user data must be kept secret from all web-based adversaries (including the server application itself);

- *integrity*: encrypted user data cannot be tampered with without it being detected by the client;

- *authorized sharing*: data shared by the user may be read only by explicitly authorized principals.

In the rest of this paper, we describe five exemplary attacks on commercial host-proof applications that break these security goals by exploiting flaws in both their cryptographic design and their web interfaces.

2 Metadata Tampering Attacks on Client-side Encryption

Client-side encryption typically relies on the user either knowing an encryption key or knowing a secret passphrase from which a key may be derived. All the applications analyzed in this paper support the PBKDF2 password-based key derivation function [13] that takes a passphrase p, salt s, and iteration count c, and generates an encryption key k (of a given length):

$$k = \text{KDF}(p, s, c)$$

The salt ensures that different keys derived from the same passphrase are independent and a high iteration count protects against brute-force attacks by *stretching* the low-entropy password [29]. The choice of s and c varies across different applications; for example LastPass uses a username as s and $c = 1000$, whereas SpiderOak uses a random s and $c = 16384$. When c is too low or the passphrase p is used for other (cheaper) computations, the security of the application can be compromised [25]. The attacks in this paper do not rely on brute-force attacks against passwords. In the rest of this paper, we assume that all passphrases and keys derived from them are strong and unguessable.

Given an encryption key k and data d, each application uses an encryption algorithm to generate a ciphertext e:

$$e = \text{ENC}(k, d)$$

The applications in this paper all support AES encryption, either with 128-bit or 256-bit keys, and a variety of encryption modes (CTR, CBC, CFB). Some applications also support other algorithms, such as Blowfish, Twofish, 3DES, and RC6. In this paper, we assume that all these encryption schemes are correctly implemented and used. Instead, we focus on what is encrypted and how encrypted data is handled.

On storage services, such as SpiderOak and Wuala, each file is individually encrypted using AES and then integrity protected using HMAC (with another key derived from the passphrase)

$$h = \text{HMAC}(k', \text{ENC}(k, d))$$

To avoid storing multiple copies of the same file, some services, including Wuala, perform the encryption in two steps: first the file is encrypted using the hash of its contents as key, then the hash is encrypted with a passphrase-derived key.

$$e = \text{ENC}(\text{HASH}(d), d), \text{ENC}(k, \text{HASH}(d))$$

The first encryption doesn't depend on the user, enabling global deduplication: the server can identify and consolidate multiple copies of a file. Although the contents

Name	Data Format	Key Derivation	Encryption	Encrypted Data	Ciphertext Integrity	Metadata Protection
Wuala	Blobs	PBKDF2-SHA256	AES, RSA	Files, Folders	HMAC	✓
SpiderOak	Files	PBKDF2-SHA256	AES, RSA	Files	HMAC	✓
BoxCryptor	Files	PBKDF2	AES	Files, Filenames	None	✗
CloudFogger	Files	PBKDF2	AES, RSA	Files	None	✗
LastPass	XML	PBKDF2-SHA256	AES, RSA	Fields	None	✗
PassPack	JSON	SHA256	AES	Records	None	✓
RoboForm	PassCard	PBKDF2	AES, DES	Records	None	✗
1Password	Keychain	PBKDF2-SHA1	AES	Records	None	✗
Clipperz	JSON	SHA256	AES	Records	SHA-256	✓

Table 1: Example host-proof web applications and their cryptographic features

Name	Backup Location	Remote Access	Bookmarklet	Custom Client	Local Page	Browser Extension
Wuala	Application Server	Java Web Applet	✗	✓	✓	✗
SpiderOak	Application Server	JavaScript Website	✗	✓	✗	✗
BoxCryptor	Third-party (Dropbox)	None	✗	✓	✗	✗
CloudFogger	Third-party (Dropbox)	None	✓	✓	✗	✗
LastPass	Application Server	JavaScript Website	✓	✗	✗	✓
PassPack	Application Server	JavaScript Website	✓	✗	✗	✗
RoboForm	Application Server	None	✓	✓	✗	✓
1Password	Third-party (Dropbox)	None	✗	✓	✗	✓
Clipperz	Application Server	JavaScript Website	✓	✗	✓	✗

Table 2: Example host-proof web applications and their web interfaces

of each file is encrypted, metadata, such as the directory structure and filenames, may be left unecrypted to enable directory browsing.

Some password managers, such as LastPass, separately encrypt each data item: username, password, credit card number, etc. but leave the database structure unencrypted. Others, such as RoboForm and 1Password, encrypt each record as a separate file. Still others encrypt the full database atomically. In most of these cases, there is no integrity protection for the ciphertext. Moreover, some metadata, such as website URLs, may be left unencrypted to enable search and lookup.

When metadata is left unprotected and is not strongly linked to the encrypted user data using some integrity mechanism (such as HMAC), it becomes vulnerable to tampering attacks. We illustrate two such attacks.

RoboForm Passcard Tampering The RoboForm password manager stores each website login in a different file, called a passcard. For example, a Google username and password would be stored in a passcard Google .rfp of the form:

```
URL3:Encode('https://accounts.google.com')
+PROTECTED-2+
<ENC(k,(username,password))>
```

That is, it contains the plaintext URL (encoded in ASCII) and then an encrypted record containing all the login data for the URL. By opening this passcard in RoboForm, the user may directly login to Google using the decrypted login data. Notably, nothing protects the integrity of the URL. So, if an adversary can modify the URL to `bad.com`, RoboForm will still decrypt and verify the passcard and leak the Google username and password to the attacken when the user browses `bad.com`.

A web-based attacker can exploit this vulnerability in combination with RoboForm's passcard sharing feature. RoboForm users may send passcards over email to their friends. So if an adversary could intercept such a passcard and replace the URL with `bad.com`, the website can then steal the secret passcard data. Similar attacks apply when synchronizing RoboForm with a compromised backup server or when malware on the client has access to the RoboForm data folder.

1Password Keychain Tampering 1Password uses a different encryption format, but similarly fails to protect the integrity of the website URL. For example, a Google record in 1Password's Keychain format is of the form:

```
{"uuid":"37F3E65BA83C4AB58D8D47ED26BD330B",
 "title":"Google",
 "location":"https://accounts.google.com/",
 "encrypted":<ENC(k,(username,password))>}
```

Hence, an attacker who has write access to the keychain may similarly modify the `location` field to `bad.com` and obtain the user's Google password. Concretely, since 1Password keychains are typically shared over Dropbox, any attacker who has (temporary) access one of the user's Dropbox-connected devices will be able to

tamper with the keychain and cause it to leak secret data to malicious websites.

Similar vulnerabilities due to lack of integrity protection on filenames in BoxCryptor and CloudFogger enable an attacker to modify filenames of encrypted files, say from `a.pdf` to `a.exe`.

Towards Authenticated Encryption It is generally accepted among the cryptographic community that "encryption without integrity-checking is all but useless"[26]. A simple fix to tampering attacks would be to use an MAC to protect the integrity of both the metadata and the encrypted items, as in Wuala and SpiderOak. Alternately, the metadata could also be encrypted and the integrity of the plaintext could be protected by a cryptographic hash (before encryption).

More generally, many host-proof applications appear to use encryption algorithms as if they guaranteed ciphertext integrity. This assumption is false for many modes of AES and especially for hybrid encryption using a combination of RSA and AES. Instead, each password manager should seek to implement a scheme that provides authenticated encryption with associated data [30], where the associated data includes unencrypted metadata.

Vulnerability Response We notified both 1Password and RoboForm about these attacks on April 3, 2012.

The 1Password team responded within days with details of their new keychain format for their next version (4.0); this format includes integrity protections which potentially addresses our concerns, but a more detailed analysis of the new format remains to be done.

The RoboForm team proved more resistant to changing their design. They questioned our threat model ("if a malware can modify passcards, it can be just a keylogger instead"), but our attack works even on passcards transported over insecure email. Despite our demo, they refused to believe that we can tamper with passcards ("produce as many passcards as you want and then modify them. they all should be rejected"). We are continuing our discussions with RoboForm but do not anticipate any fixes in the near future.

Both vulnerabilities were publicly disclosed [19, 20].

3 Cross-Site Request Forgery on Remote Web Access

Some host-proof applications such as LastPass and SpiderOak offer fully-featured JavaScript interfaces to its roaming users. A user may login to the website with her passphrase and access her data. However, the passphrase itself should never be sent to the server; instead the JavaScript client should derive decryption keys within the browser. Ideally, all decryptions would also be run within the user's browser, but for efficiency, some decryptions may be executed server-side, with the promise that decryption keys are destroyed on logout.

SpiderOak JSONP CSRF Attack The SpiderOak website uses AJAX with JSONP to retrieve data about the user's devices, directory contents and share rooms. So, when a user is logged in, a GET request to `/storage/<u32>/?callback=f` on `https://spideroak.com` where `<u32>` is the base32-encoded username returns:

```
f({"stats":
   {"firstname": "Legit",
    "lastname": "User", "devices": 3, ...
    "devices": [["homepc", "homepc/"],
                ["laptop", "laptop/"],
                ["mobile","mobile/"]]})
```

Hence, by accessing the JSON for each device (e.g. `/storage/homepc/`), the JavaScript client retrieves and displays the entire directory structure for the user.

It is well known that JSONP web applications are subject to Cross-Site Request Forgery if they do not enforce an allowed origin policy [24]. SpiderOak enforces no such policy, hence if a user browsed to a malicious website while logged into SpiderOak, that website only needs to know or guess the user's SpiderOak username to retrieve JSON records for her full directory structure.

More worryingly, if the user has shared a private folder with her friends, accessing the JSON at `/storage/<u32>/shares` yields an array of shared "rooms" that includes access keys:

```
{"share_rooms":
   [{"url": "/browse/share/<id>/<key>",
     "room_key": "<key>",
     "room_description": "",
     "room_name":<room>}],
 "share_id": "<id>",
 "share_id_b32": "<u32>"}
```

So, the malicious website may now at leisure access the shared folders at `https://spideroak.com/browse/share/<id>/<key>` to steal all of a user's shared data.

Key Management for Shared Data Our specific attack can be prevented by simply adding standard CSRF protections to all the JSONP URLs offered by SpiderOak. However, a more general design flaw is the management of encryption keys for shared data. When a folder is shared by a user, it is decrypted and stored in plaintext on the server, protected only by a password that is also stored in plaintext on the server. This breaks the host-proof design completely since flaws in the SpiderOak website may now expose the contents of all shared folders (as indeed we found). A better design would be to use encrypted shared folders as in

Wuala [27], where decryption keys are temporarily provided to the website but not stored permanently.

Vulnerability Response We notified the SpiderOak team about the attack on May 21, 2012; they acknowledged the issue and disabled JSONP within one hour. However, no change was made to the management of share room keys, and no additional protections against CSRF attacks, such as `Referer` or token based checks, have been put in place. We fear that shared data on SpiderOak remains vulnerable to other website attacks; notably, many of the problems reported on the SpiderOak Security Response page relate to cross-site scripting.

4 Stealing Data from Client-side Websites

Wuala is a Java application that may be run directly as a desktop client or as a Java applet from the Wuala website. It maintains an encrypted directory tree where each file is encrypted with a different key and the hierarchy of keys is maintained by a sophisticated key management structure [27]. When started, Wuala asks for a username and password, uses them to derive a master key which is then used to decrypt the directory tree. On Windows systems, Wuala creates the following local directory structure:

```
%userprofile%/AppData
├─ Local
│  └─ Wuala
│     └─ Data (local cache)
└─ Roaming
   └─ Wuala
      └─ defaultUser (master key file)
```

The `defaultUser` file contains the master key for the current user. The `Data` folder contains the encrypted directory tree along with plaintext data for files that have been recently uploaded or downloaded from the server.
Wuala also runs a lightweight HTTP server on `localhost` at port 33333. This HTTP server is primarily meant to provide various status information, such as whether Wuala is running, whether backup is in progress, log error messages, etc. It may also be used to open the Wuala client at an given path from the browser. The user may enable other users on the LAN to access this HTTP server to monitor its status. The HTTP server cannot be disabled but is considered a mostly harmless feature.

Database recovery attack on Wuala We discovered a bug on the Wuala HTTP server, where files requested under the `/js/` path resolve first to the contents of the main Wuala JAR package (which has some JavaScript files) and then, if the file was not found, to the content of Wuala's starting directory.

If Wuala was launched as an applet, its starting directory will be `Roaming` in the above tree, meaning that browsing to `http://localhost:33333/js/defaultUser` will return the master key of the current active user. Using this master key file anyone can masquerade as the user and obtain the full directory tree from Wuala.
If Wuala was started from as a desktop client, its stating directory will be `Local` instead, allowing access to the local copy of the database, including some plaintext files. These flaws can be directly exploited by an attacker on the same LAN (if LAN access to the HTTP server is enabled; it isn't by default), or by any malware on the same desktop (even if the malware does not have permission to read or write to disk or to access the Internet). The attacker obtains the full database if Wuala was started as an applet, and some decrypted files otherwise.

Protecting Keys from Web Interfaces Our attack relies on a bug in the HTTP server, it simply should not allow access to arbitrary files under the `/js/` path.
More generally, the attack reveals a design weakness that the Wuala master key is available in plaintext when Wuala is running and is stored in plaintext on disk if the user asks Wuala to remember his password. This file is extremely sensitive since obtaining the file is adequate to reconstruct and decrypt a complete copy of the user's directory tree (on any machine). The software architecture of Wuala makes the file available to all parts of the application including the HTTP server. We advocate a more modular architecture that isolates sensitive key material and cryptographic operations in separate processes from (potentially buggy) web interfaces.

Vulnerability Response We notified the Wuala team about the vulnerability on May 21, 2012. They responded immediately and released an update (version 399) within 24 hours that disabled file access from the local web server. No other change was made to the HTTP server or master key cache file following our report. The vulnerability has been publicly disclosed [17].

5 Phishing Attacks on Browser Extensions

Password managers typically offer browser extensions that can be used to fill forms automatically on known websites. These extensions are written in JavaScript and either implement cryptography in JavaScript (e.g. LastPass) or call out to an external desktop application (e.g. 1Password and RoboForm).
When a user visits a website, say `gmail.com` with a password manager's browser extension installed, the extension examines the URL of the page to decide whether or not to automatically fill in the login form (using data re-

trieved and decrypted from the database). However, the code for parsing the URL is often flawed and does not account for maliciously crafted URLs.

1Password Phishing Attack For example, the URL parsing code in the 1Password extension (version 3.9.2) attempts to extract the top-level domain name from the URL of the current page:

```
var href = getBrowser().contentWindow.location.href
           + "/";
var domain = href.replace(/^http[s]*:\/\/(.*?)\/.*$/i,
                          "$1");
var middle = domain.replace(/^(www.)*(.*)/i, "$2");
return middle.substring(0,1).toUpperCase() +
       middle.substring(1,middle.length);
```

So given a URL `http://www.google.com`, this code returns the string `Google.com`. However, this code does not correctly account for URLs of the form `http://user:password@website`. So, suppose a malicious website redirected a user to the url `http://www.google.com:xxx@bad.com`. The browser would show a page from `http://bad.com` (after trying to login as the "user" `Google.com`), but the 1Password browser extension would incorrectly assume that it was on the domain `Google.com` and release the user's Google username and password. This amounts to a phishing attack on the browser extension, which is particularly serious since one of the advertised features of password managers like 1Password is that they attempts to protect naive users from password phishing.
Similar attacks can be found on other password managers, such as RoboForm's Chrome extension, that use URL parsing code that is not defensive enough.

URL Parsing Parsing URLs correctly with regular expressions is a surprisingly difficult task, despite URLs having a well understood syntax [14], and leading websites often get it wrong [31]. Perhaps the most widely used URL parsing library for JavaScript is parseUri [8] which uses the following regular expression (in "strict" standard-compliance mode):

```
strict: /^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)
    (?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?)
    )?((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*)
    )?(?:#(.*))?)/
```

This regular expression is also incomplete. For example, given the URL `http://bad.com/#@accounts.google.com`, it yields a domain `accounts.google.com` whereas the correct interpretation is `bad.com`.

Domain-based Authorization Password managers authorize websites based on their domain name. The basic flaw that enables our phishing attacks is that the interpretation of the domain of the URL by the browser extension is inconsistent with the interpretation of the browser. In the cases shown above, the extension was wrong and the browser was right. But even if the extension were right and the browser were wrong, a secret password may be leaked. An easy fix that prevents our attack is for the extension to directly use the parsed `window.location` object given by the browser. A different fix is to use a careful regular expression parser that mimics the browser.
A more general design question is whether domain-based authorization is appropriate for website login. On hosting websites such as WordPress and Google Sites, hundreds of different websites may share the same domain name, causing domain-based password managers to be very error-prone. Moreover, users may wish to only release their passwords over HTTPS, but domains do not include protocol information. So for example, if a user asked LastPass to remember her password to `https://facebook.com`, and later she was redirected to the HTTP login form on `http://facebook.com`, LastPass will happily fill in her username and password, revealing it to eavesdroppers on the network. We advocate that password managers implement site-specific authorization policies that include full origins (scheme, host, port) and enable users to choose their desired level of security.

Vulnerability Response We notified 1Password about the phishing vulnerability on April 3, 2012. The 1Password team responded immediately and released a new beta version of their browser extensions on April 5, 2012 (build 39304) that implements a new, more careful, URL parsing function. This function fixes the specific attack that we found but a full verification of their new URL parsing code and its consistency with different browsers remains an open question. The 1Password vulnerability has been publicly disclosed [18].

6 Rootkit attacks on bookmarklets

Bookmarklets are bookmarks that contain a fragment of Javascript code. When clicked, this code is injected into the current active page, a feature commonly used by password managers to fill login forms on the page using the user's password database. Bookmarklets can be considered lightweight substitutes for browser extensions and are particularly suited for mobile and roaming users. Unlike extensions, bookmarklets are evaluated inside the Javascript scope of the page they are being injected into, making them vulnerable to a variety of threats, collectively called *rootkit* attacks [21] that are very hard to protect against. Of particular concern are bookmarklets that handle sensitive data like passwords: they must ensure that they do not inadvertently leak the

data meant for one site to another. The countermeasure proposed in [21] addresses exactly this problem by verifying the origin of the website and has been adopted by a number of password managers, including LastPass and PassPack. However, they are still vulnerable to attack.

LastPass master key theft The LastPass Login bookmarklet loads code from `lastpass.com` that defines various libraries and then runs the following (stripped down) function:

```
function _LP_START() {
    _LP = new _LP_CONTAINER();
    var d = {<encrypted form data>};
    _LP.setVars(d,  '<user>',
     '<encrypted_key>', _LASTPASS_RAND, ...);
    _LP.bmMulti(null, null);
}
```

This code retrieves the encrypted username and encrypted password for the current website, it downloads a decryption key (encrypted with the secret key associated with the bookmarklet), and uses the decryption key to decrypt the username and password before filling in the login form. Even though the decryption key is itself encrypted, it is enough to know `<user>` and `_LASTPASS_RAND` to decrypt it. Hence, a malicious page can detect when the `_LP_CONTAINER` object becomes defined (i.e. when the user has clicked the LastPass bookmark), redefine this object and call `_LP_START` again to decrypt and leak the key, username, and password.
Since the username and password are meant for the current (malicious) page, this does not seem like a serious attack, until we note that the decryption key obtained by this attack is the permanent master key that is used to encrypt all the usernames and passwords in the user's LastPass database. Hence, the bookmarklet leaks the decryption key for the full database to a malicious website. A similar attack applies to the PassPack bookmarklet: a malicious website can steal a temporary encryption key that enables it to add a new record into the user's password database for any URL.

Per-record Key Derivation To protect host-proof applications against bookmarklet attacks, it is not enough to strongly authenticate the page that loads the content script. We also need to verify that the website is authorized to read any secret included in the content script. For example, our attacks would not be so serious if the keys revealed by the bookmarklet were specific to the website. Instead, they reveal a design flaw in the ways keys are used in LastPass; LastPass derives a master key from a username and a master password, without using any seed. This key remains constant for a long time (until the master password is changed). Moreover, it is used to individually encrypt each username and password field, and also used to re-encrypt the full database. To correctly implement data sharing with different websites, we advocate that different keys be generated for different records, by using per-record salts, or by including the URL (or its domain name) into the key derivation process.

Vulnerability Response We notified LastPass about the vulnerability on May 21, 2012. The LastPass team acknowledged the risk of leaking the master decryption key to malicious websites and changed their bookmarklet design within 24 hours. Decryption is now performed inside an iframe loaded from the `https://lastpass.com` origin, preventing the host page from stealing the key. However, they did not modify the overall design; hence, LastPass still uses a single master key for all encryptions.

7 Conclusions

The host-proof application design pattern provides one level of isolation between sensitive user data and website attackers, but this is not enough. Moving cryptography to the client means that special attention should be paid to enforcing strong isolation between code that is relevant to the user interface and code that performs security-sensitive cryptographic operations.
Current commercial host-proof client applications have critical flaws in the way they integrate browser-based interfaces with cryptographic code. We have presented a series of practical attacks that exploit these flaws. We have built demonstrations of these attacks and helped various vendors fix their software.
From the viewpoint of web application security, our attacks are not new; what is novel is their interaction with cryptographic mechanisms, and the way they reveal security design flaws. We found these attacks by a careful but manual study of selected host-proof applications over a few weeks. It is worrying that we were able to find attacks on most applications we looked at without the aid of any sophisticated tools.
To find more subtle attacks or to verify that an application is free from attack will require automated tools that can account for both web-specific threats and a precise model of cryptography but still scale up to realistic web applications. As ongoing and future work, our goal is to build such analysis tools based on sound formal foundations [22, 23] and apply them, for example, to the verification of the host-proof web applications studied here.

Acknowledgments Bhargavan is supported by the ERC Starting Grant CRYSP. This work was done during Delignat-Lavaud's internship at INRIA.

References

[1] 1Password. https://agilebits.com.

[2] BoxCryptor. http://boxcryptor.com.

[3] Clipperz. http://clipperz.com.

[4] CloudFogger. http://cloudfogger.com.

[5] Dropbox. http://dropbox.com.

[6] Host-proof hosting. http://ajaxpatterns.org/Host-Proof_Hosting.

[7] LastPass. http://lastpass.com.

[8] Parseuri 1.2: Split urls in javascript. http://stevenlevithan.com/demo/parseuri/js/.

[9] PassPack. http://passpack.com.

[10] RoboForm. http://www.roboform.com.

[11] SpiderOak. http://spideroak.com.

[12] Wuala. http://wuala.com.

[13] PKCS #5: Password-Based Cryptography Specification, Version 2.0. IETF, 2000.

[14] RFC3986: Uniform Resource Identifier (URI): Generic Syntax. IETF, 2005.

[15] Keys to the cloud castle. Economist, May 18th 2011. http://www.economist.com/blogs/babbage/2011/05/internet_security.

[16] LastPass Security Notification, May 4th 2011. http://blog.lastpass.com/2011/05/lastpass-security-notification.html.

[17] CVE-2012-3874: Wuala Status Page Leaks Plaintext Files, July 7 2012.

[18] CVE-2012-3879: Phishing attack on 1Password Browser Extensions, July 8 2012.

[19] CVE-2012-3882: RoboForm "Receive Passcard by E-mail" Feature Accepts Tampered Metadata, July 8 2012.

[20] CVE-2012-3883: 1Password Restore Feature Accepts Tampered Metadata, July 8 2012.

[21] Ben Adida, Adam Barth, and Collin Jackson. Rootkits for JavaScript environments. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, 2009.

[22] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE, 2010.

[23] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, Cambridge, MA, USA, June 2012. IEEE. To appear.

[24] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 75–88. ACM, 2008.

[25] Andrey Belenko and Dmitry Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Technical report, Elcomsoft Co. Ltd., 2012. http://www.elcomsoft.com/WP/BH-EU-2012-WP.pdf.

[26] Steven M. Bellovin. Cryptography and the internet. In *Advances in Cryptology: Proceedings of CRYPTO '98*, August 1998.

[27] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Rogert Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, SRDS '06, pages 189–198, 2006.

[28] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Proceedings of the 14th international conference on Financial cryptograpy and data security*, FC'10, pages 136–149, Berlin, Heidelberg, 2010. Springer-Verlag.

[29] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Proceedings of the First International Workshop on Information Security*, ISW '97, pages 121–134, London, UK, UK, 1998. Springer-Verlag.

[30] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 98–107, New York, NY, USA, 2002. ACM.

[31] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.