Bui, Thanh; Rao, Siddharth; Antikainen, Markku; Aura, Tuomas

XSS Vulnerabilities in cloud-application add-ons

Published in:
Proceedings of the 15th ACM Asia Conference on Computer and Communications Security

DOI:
10.1145/3320269.3384744

Published: 05/10/2020

Document Version
Peer reviewed version

Please cite the original version:
Bui, T., Rao, S., Antikainen, M., & Aura, T. (2020). XSS Vulnerabilities in cloud-application add-ons. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (pp. 610-621). ACM. https://doi.org/10.1145/3320269.3384744

XSS Vulnerabilities in Cloud-Application Add-Ons

Thanh Bui
thanh.bui@aalto.fi
Aalto University

Siddharth Rao
siddharth.rao@aalto.fi
Aalto University

Markku Antikainen
markku.antikainen@aalto.fi
Aalto University

Tuomas Aura
tuomas.aura@aalto.fi
Aalto University

ABSTRACT

Many cloud-application vendors open their APIs for third-party developers to easily extend the functionality of their applications. The features implemented with these APIs are called *add-ons* (also called add-ins or apps). This is a relatively new phenomenon, and its effects on the application security have not been widely studied. It seems likely that some of the add-ons have lower code quality than the core applications themselves and, thus, may bring in security vulnerabilities. In this work, we found that many of such add-ons are vulnerable to cross-site scripting (XSS). The attacker can take advantage of the document-sharing and messaging features of the cloud applications to send malicious input to them. The vulnerable add-ons then execute client-side JavaScript from the carefully crafted malicious input. In a major analysis effort, we systematically studied 300 add-ons for three popular application suites, namely Microsoft Office Online, G Suite and Shopify, and discovered a significant percentage of vulnerable add-ons among them. We present the results of this study, as well as analyze the add-on architectures to understand how the XSS vulnerabilities can be exploited and how the threat can be mitigated.

ACM Reference Format:
Thanh Bui, Siddharth Rao, Markku Antikainen, and Tuomas Aura. 2020. XSS Vulnerabilities in Cloud-Application Add-Ons. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), June 1–5, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3320269.3384744

1 INTRODUCTION

In modern web applications, user data is stored and processed mainly in the cloud, and the user interface is implemented with HTML and JavaScript on the web browser. This kind of architecture has several advantages. For example, the users do not need to install or update the applications, and sharing and synchronizing data between users and services becomes easier. A well-known example of a cloud application is Google Docs [2], an online document editor, which allows collaborative editing of office documents. The users need to trust the cloud applications to keep their data safe, and application developers have come a long way in securing the services.

Many of the cloud applications follow the microservice architecture where much of the functionality is implemented as independent services that are loosely coupled to the core service through APIs. The APIs can also be opened to external developers. The features implemented with these APIs are variably called *add-ons*, *add-ins*, or *apps*; we use the word add-on in this paper. For instance, the Translate [19] add-on for Google Docs allows the user to translate text to a chosen language — a feature that is not part of the core service. Successful cloud applications have created *marketplaces* for add-ons and aim to grow an ecosystem of innovative add-on services around their core platform.

The growing add-on market, however, creates new dangers. Many add-ons are quick hacks by inexperienced developers, and the users may not be aware of the difference between the add-on and the trusted host platform. Moreover, the host-application vendors have an incentive to attract new add-on developers to their ecosystems, which may lead to less stringent security controls for the add-ons than for the core service.

In this work, we study the security risks that arise from potential security vulnerabilities in cloud-application add-ons. In particular, we are interested in how add-on services process untrusted user input. This is a critical issue because the emphasis on collaboration and data sharing in the cloud applications makes it easy to exploit vulnerabilities in the handling of untrusted data.

The focus of our analysis is on JavaScript code injection, popularly known as *cross-site scripting (XSS)* [45]. While XSS has received much attention among web security researchers [24, 32, 33, 53, 54], to our best knowledge, the dangers of XSS in the context of cloud-application add-ons have not been extensively studied. In this paper, we aim to fill this gap with the following contributions:

- We explain in detail how XSS attacks against cloud-application users can occur through vulnerable add-ons.
- We analyze the architecture designs and the security mechanisms of three popular application suites, namely G Suite [1], Microsoft Office Online [3], and Shopify [4]. The goal is to find what the XSS attacker can gain in each case.
- We evaluate how widespread the problem is with an empirical study on the add-ons from the marketplaces of the selected application suites.
- For defensive solutions, we discuss good practices which the add-on developers can follow to secure their products. We also present the lessons that we learned from our analysis about design choices and their impact on the security of an

ASIA CCS '20, June 1–5, 2020, Taipei, Taiwan
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6750-9/20/06…$15.00
https://doi.org/10.1145/3320269.3384744

add-on system. We hope that the lessons would be useful for cloud-application vendors who are developing or improving their add-on systems.

The rest of the paper is structured as follows. Section 2 provides necessary background information. Section 3 explains in detail how XSS can occur in vulnerable add-ons. Section 4 describes our analysis on the add-on architectures of the selected cloud-application suites. Section 5 presents the results of the empirical study. Section 6 describes defensive solutions. Section 7 discusses the results. Finally, Section 8 summarizes related work, and Section 9 concludes the paper.

2 BACKGROUND

This section explains the concepts needed in the rest of the paper, i.e. cross-site scripting and cloud-application add-ons.

2.1 Cross-site scripting

Cross-site scripting (XSS) [45] is among the most common web application vulnerabilities. In an XSS attack, the attacker injects malicious client-side code, typically JavaScript, to a website that does not sufficiently filter client input. When a victim user accesses the target site, the injected code is executed in her web browser in the same context as legitimate scripts on the same page. Thus, the injected code may gain unauthorized access to resources on the target site, bypassing user authentication and the same-origin policy (SOP) [61].

XSS was first discussed in 2000 [12], and various variants of the attack have been discovered since then. In general, XSS attacks can be classified into four types:

(1) *Stored (persistent) XSS*: The injected script is permanently stored in a database on the target website. It is pushed to the victim's browser when the victim accesses the stored information.
(2) *Reflected (non-persistent) XSS*: The injected script is not stored on the server. Rather, the attacker tricks the victim's browser into sending the script to the server, which includes it in the immediate response page such as an error message or search results.
(3) *DOM-based XSS*: The injected script never reaches the server. Instead, it is injected to the Document Object Model (DOM) of the vulnerable web page, e.g. from a URL query string or fragment identifier, and executed directly on the client side.
(4) *Persistent client-side XSS*: This is relatively new type of XSS [53]. In the attack, the attacker injects malicious payloads into client-side storage (e.g. Web Storage, cookies) of the users that it targets. This way, the attack succeeds if the JavaScript code of the website executes the malicious data from the storage.

XSS variants can also be classified into *server-side XSS* and *client-side XSS* [46]. The former occurs when untrusted user data is included in an HTML response generated by the server, while the latter occurs when a JavaScript call uses untrusted user data to update the DOM of the vulnerable page in an unsafe way.

In all types of XSS, the attacker gains access to any information which the victim's browser stores or processes for the target website. Most commonly, the attacker steals cookies that enable it to impersonate the victim to the website. The attacker can also create a JavaScript key logger to record sensitive data entered by the victim, for example, passwords and credit card numbers. Moreover, the injected code can invoke HTML5 APIs, such as webcam or geolocation, although some of the APIs will only allow access if the victim has opted in to the features for the target site.

2.2 Cloud-application add-ons

Add-ons (also known as add-ins, plugins, extensions, or apps) add customized commands and features to a cloud application, called the *host application*. Each add-on is basically a separate web service with its own server and client components, but it has access to the user data and some core functionality of the host application through APIs defined by the application.

In addition to the add-on web service, each add-on has a web front-end, which is implemented with HTML and JavaScript. When the user starts an add-on, the host application typically loads the add-on UI into an `iframe` and displays it seamlessly as part of the user interface of the host application. There are two fundamentally different ways for the add-on UI to interact with the host application. It can either communicate locally with the host-application UI component or via the backend servers. In the latter case, the add-on UI usually connects to the add-on server in the cloud, which interacts with the host application server and accesses the user data through backend APIs that are not visible to the user.

Access control. Cloud application vendors typically implement *permission-based access control* for add-ons to limit their access to user data in the host application. Each add-on has a list of the permissions which it requires to operate. The host application usually asks the user to explicitly approve the permissions when the user runs the add-on for the first time or during its installation. In cases where the add-on is updated and needs new permissions, the host application will ask the user to review and approve them again. This access control tends to be rather coarse grained, i.e. the user has to grant all the requested permissions for either all user data or for a specific document. Furthermore, the add-on retains the permissions until the user uninstalls it.

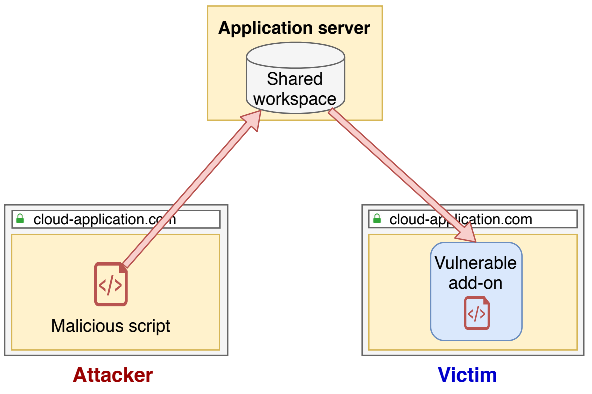
Marketplaces. Cloud application vendors often list their add-ons in an online *marketplace*, from which the users can choose and install (i.e. enable) any add-on for the applications. For instance, the G Suite marketplace [1] lists add-ons for Google applications such as Gmail and Google Docs. Usually, only a relatively small number of add-ons are provided by the application vendor itself, and the rest have been created by third-party developers.

3 XSS IN VULNERABLE ADD-ONS

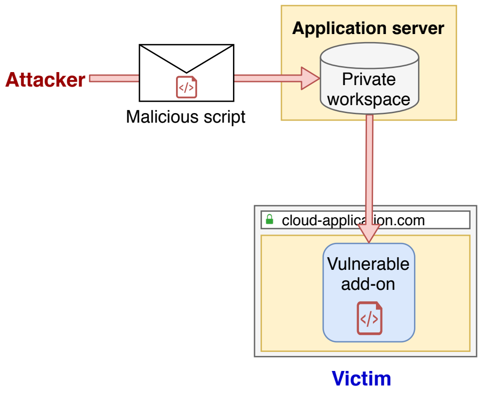
In this paper, we focus on *non-malicious add-ons*. These add-ons are written by well-meaning developers who do not intend to cause harm but might not be security experts. Nonetheless, such add-ons can be vulnerable to external attacks, including XSS.

We have identified two types of XSS attacks against vulnerable add-ons (see Figure 1):

(1) **Attack with shared workspace**: The attacker and the victim are colleagues, friends or remote collaborators, who use the same cloud application. The attacker shares a *workspace*



(a) With shared workspace



(b) With outside input

Figure 1: XSS attacks with vulnerable add-ons

with the victim. The workspace concept varies depending on the host application, but it basically is any environment through which changes made by one user are propagated to the others (e.g. a Google Docs document). The attacker injects malicious JavaScript code into the shared workspace. If the text would be visible to the user, it can be hidden with the usual techniques like a small font size or text color matching the background. When the victim enables the vulnerable add-on for the shared workspace and the add-on renders the attacker's input in an unsafe way, the injected script may become part of the web page in the add-on `iframe`, where it is executed by the victim's web browser. This way, the attacker has performed an XSS attack on the victim.

(2) **Attack with outside input**: Some host applications accept external input such as messages from non-users. This input may contain XSS code for an add-on that later processes the data. For example, if the host application is an email service (e.g. Gmail or Outlook), the attacker can hide the malicious script in an email and send it to the victim. If the victim has enabled a vulnerable add-on to process emails, the injected script may again find its way to the add-on's `iframe` and be executed there like any JavaScript in that frame.

The details of how the attacker injects the script into the shared workspace are naturally specific to the cloud application, to the add-on, and to the vulnerability that is being exploited. In any case, the root cause of the above attacks is that the vulnerable add-on routes the untrusted user input to JavaScript's execution sinks in the add-on UI without sanitizing it. In particular, malicious data from the attacker can be executed if the add-on renders it as HTML rather than as text. HTML element sinks include `document.write` or `document.body.innerHTML`. The attack would also succeed if the attacker's content is given as input to JavaScript methods such as `eval` and `setTimeout` which convert string input to code. The latter type of mistake is less likely, though, because developers are aware of the dangers of such functions.

Compared to the traditional variants of XSS (see Section 2.1), the attacks that we describe here are similar to *stored XSS* because the attacker's malicious input is stored in the host application's data storage. They can be either server-side XSS or client-side XSS depending on whether the malicious data is processed and rendered by the add-on server or by the add-on UI.

General consequences. With the ability to run arbitrary scripts in the context of the vulnerable add-on, the attacker can perform at least the following types of attacks:

- The attacker can access data on the add-on server through its APIs. In a microservice architecture, the add-on server is likely to have its own data storage.
- The attacker may be able to access data in the host application with cross-domain messaging, or indirectly through the add-on server. The ability to do this depends on the design of the host application and its add-on APIs.
- The attacker can spoof another user interface in the add-on `iframe` and trick the user into entering confidential data or credentials.
- As in all XSS attacks, the attacker can access HTML5 APIs and request access to local resources, such as geolocation, or authorization to access external resources owned by the victim user.

We will discuss the designs of different host applications and analyze what the attacker can gain in each case in Section 4. It is important to understand that the malicious script runs in the `iframe` with a different origin than the host application. Thus, it cannot access the DOM model of the host-application within the web browser or the cookies related to the host application. Instead, any access to host-application data has to be gained either through published APIs in the add-on server or with cross-origin access and messaging methods that are enabled by the host application.

4 HOST-APPLICATION ARCHITECTURES

To understand the consequences of malicious code execution, we analyzed the add-on architectures of three popular cloud application suites: Microsoft (MS) Office Online [3], G Suite [1], and Shopify [4]. This section presents our analysis in detail.

4.1 MS Office Online add-ons

MS Office Online is a cloud-based office suite, which includes popular office applications like Word, Excel, PowerPoint, and Outlook. All applications in the suite have the same architecture, which is illustrated in Figure 2. The add-on UI is displayed inside the host application UI, which allows the user to interact with the add-on seamlessly as part of the application. The add-on UI is contained in an `iframe` and has a different origin than the encapsulating application, which prevents it from directly accessing user data in the host application.

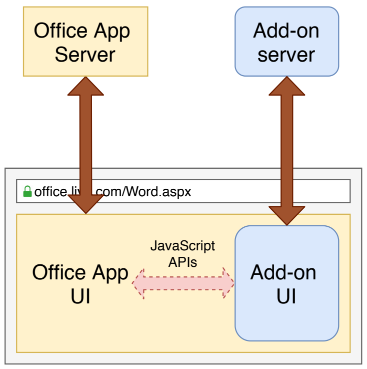


Figure 2: MS Office Online add-on architecture

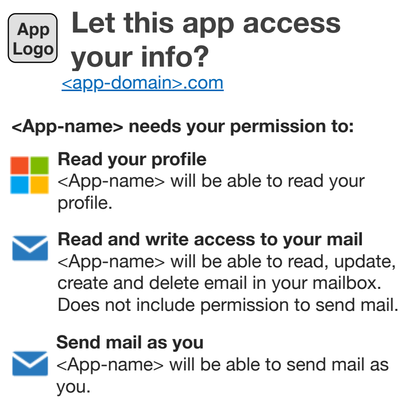
The add-on interacts with the host application on the client side via JavaScript APIs. Specifically, MS Office Online applications use `window.postMessage()`[43] for cross-origin messaging between the add-on `iframe` and the parent application window. The add-on can request different levels of access to the host-application data [38], shown in Table 1. If the host application is Word, Excel, PowerPoint or OneNote, the add-on can only request access to the current document on which the user is working. Outlook add-ons, on the other hand, can request access not only to the current item (i.e. email or compose form) but also to the user's whole mailbox. Furthermore, Outlook add-ons can call `getCallbackTokenAsync()`, a special API that returns an *access token* with the permission level of the add-on. The add-on UI running in the browser can send this token to the add-on server, which can use it to access the email server [40] on the add-on's behalf for a limited time.

4.1.1 XSS exploits. Let us consider how an attacker can exploit an MS Office Online add-on that is vulnerable to XSS. First, the victim needs to install i.e. enable the add-on. Then, depending on the add-on, the attacker can exploit the situation with either of the two attack vectors that we presented in Section 3: it can inject a malicious script into a document that is shared with the victim, or in the case of Outlook, it can send an email that contains the malicious script to the victim. When the attacker's script is running with the add-on's permissions, it can exploit the situation in the following ways.

Get the same level of access as the add-on. Because of the local messaging with the host application window, the attacker can access any resources which the add-on is permitted to access. Since the attacker's scripts run within the add-on's `iframe`, it is not possible for the host application to differentiate between malicious requests from the injected code and legitimate ones from the add-on UI code.

If the host application is Word, Excel, PowerPoint or OneNote, the attacker can access only the open document. This might not be immediately useful to the attacker because the document was originally shared with or by the attacker. However, the attacker can use the injected script as a backdoor to retain access even after the victim revokes his legitimate access. For example, the victim may make a personal copy of a form or template before filling it with confidential data, but the attacker retains access through the injected scripts in the form or template. In Outlook, on the other hand, the attacker will gain full access to the victim's mailbox if the vulnerable add-on has the *ReadWriteMailbox* permission. This means that the attacker can read all of the victim's emails and send emails on the victim's behalf.

Request an OAuth 2.0 token. As noted earlier, the ability to control the `iframe` enables the attacker to spoof parts of the application user interface, which makes it possible to trick the user in various ways, such as phishing for confidential data. We found one specific trick along these lines which the attacker can play on MS Office Online users. Many add-ons act as connectors between the MS Office Online applications and third-party web services built on the Azure platform [36]. Such an add-on implements the UI for a third-party backend server. The backend server communicates with the host application via APIs that are far more powerful than the client-side add-on APIs. To obtain such access, the third-party service provider must register an Azure application with the Microsoft identity platform [37], and the user must authorize the application to access the resources in the host application. The authorization is based on OAuth 2.0 [26] as follows. The add-on displays a popup that shows information about the application including its name, logo and domain, as well as the permissions which the application is requesting. This is illustrated in Figure 3. If the user agrees to authorize the application, the application will receive an *access token*, which it can use to access the requested resources from anywhere.



App Logo
Let this app access your info?
<app-domain>.com

<App-name> needs your permission to:

Read your profile
<App-name> will be able to read your profile.

Read and write access to your mail
<App-name> will be able to read, update, create and delete email in your mailbox. Does not include permission to send mail.

Send mail as you
<App-name> will be able to send mail as you.

Figure 3: MS Office Online add-on authorization prompt

Since the users of MS Office Online add-ons are already familiar with the OAuth authorization prompt, the attacker can exploit it

Table 1: MS Office Online add-on permission levels

Application	Permission	Description
Outlook	Restricted	Read phone numbers, addresses, and URLs from the current item
	ReadItem	Read all properties of the current item
	ReadWriteItem	Full access to the current item
	ReadWriteMailbox	Full access to the mailbox
Word, Excel, PowerPoint, OneNote	Restricted	Read/write settings of the add-in that are stored in the current document
	ReadDocument	Read only the text in the current document
	ReadAllDocument	Read everything in the current document, which includes text, formatting, links, graphics, etc.
	WriteDocument	Write to the user's selection in the current document.
	ReadWriteDocument	Full access to the current document

to phish for access rights. First, the attacker creates an Azure application with the exact same name as the vulnerable add-on. This is possible because Azure applications do not need to have unique names. With the injected script, the attacker requests authorization for some of the user's resources. If the victim authorizes the attacker's application, the attacker receives an access token with which it can access the victim's data from anywhere. The token is similar to the Outlook token discussed above but supported by all the MS Office Online applications. It would be difficult for the victim user to judge whether a particular add-on should be granted an OAuth 2.0 token or limited to the add-on APIs.

4.1.2 Case study: Translator for Outlook. We will use the Translator for Outlook [39], an add-on developed by Microsoft, to demonstrate the exploits. As the name suggests, it works with the email service Outlook and translates the user's emails into the selected language. The workflow for the add-on is as follows.

(1) The user opens an email and starts the add-on. The host application will display the add-on as a side panel in its UI.
(2) In the add-on UI, the user selects the language to which she wants to translate the open email.
(3) The add-on translates the email to the selected language and displays the result in its UI.

The vulnerability in the add-on is that it renders the translated text as HTML without escaping the text first. As the result, the attacker, who could be anyone on the Internet, can send an email with a malicious script to the victim. When the victim tries to translate it with the add-on, the scripts will be executed. The add-on, fortunately, has only the *ReadItem* permission. Thus, by exploiting the local messaging between the add-on UI and the host application, the attacker will not be able to do anything meaningful, such as read the victim's mailbox or send emails on behalf of the victim. To gain these access rights, the attacker's script must request an OAuth 2.0 token in the way described in the previous section.

4.2 G Suite add-ons

G Suite is another office suite and developed by Google. Some well-known applications in the suite are Google Docs, Google Sheets, and Gmail.

Before going into details of how G Suite add-ons work, we need to understand the concept of Google APIs [17]. They are a set of APIs that give programmatic access to many Google products, such as Google Maps and Google Drive. Before a client (e.g. a website) can access private user data with these APIs, it must be attached to a Google Cloud Platform (GCP) project [18]. The client then needs to obtain an *access token* with OAuth 2.0 as follows. First, the user is redirected to the Google Authorization website, where the user must sign in with her Google account. The website then displays an authorization prompt showing the name of the GCP project and the permissions which the client is requesting. If the user grants the permissions, the Google Authorization server sends the access token to the client.

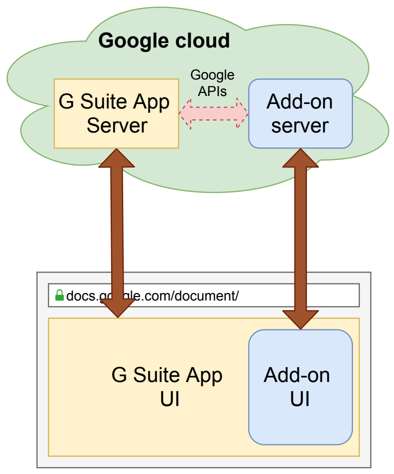
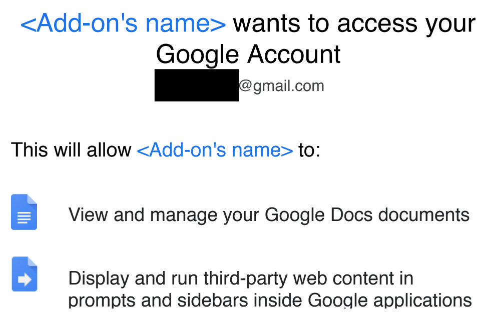


Figure 4: G Suite add-on architecture

Figure 4 shows the architecture for G Suite add-ons. The main difference to MS Office Online is that the G Suite add-ons are always hosted in the Google cloud. The add-on server is basically a Google APIs client that can directly interact with the user data. The add-on UI sends requests to interfaces defined by the add-on server, and the server implements the desired action on user data as well as returns responses. The add-on server interfaces can only be accessed by add-on code that originates from the same server. One example is Translate [19], a Google Docs add-on provided by Google. Its server has two main interfaces: one translates the user-selected text and returns the result, and the other replaces the text of the current selection with the translated text.

Since the add-on server is a Google APIs client, it must be authorized before it can access the user's private data. This occurs when the user starts the add-on for the first time. Figure 5 shows a typical authorization prompt. If the user approves, the add-on server obtains an access token with the requested permissions.



<Add-on's name> wants to access your Google Account

[redacted]@gmail.com

This will allow <Add-on's name> to:

View and manage your Google Docs documents

Display and run third-party web content in prompts and sidebars inside Google applications

Figure 5: G Suite add-on authorization prompt

G Suite add-ons can request permissions to access user data in any G Suite application. For example, a Google Docs add-on can request permission to send emails from the user's Gmail account. While this cross-application permissions model makes the add-ons flexible and powerful, malicious add-ons could exploit it to gain access to a wide range of user data by, for example, requesting permissions to "Read, compose, send, and permanently delete all your email from Gmail" and "See, edit, create, and delete all of your Google Drive files". To mitigate such threats, Google performs manual verification of add-ons that request sensitive permissions [23] to ensure that the add-ons comply with the Google API User Data Policy [22].

4.2.1 *XSS exploits.*

Next, we consider what kind of access a successful XSS attacker can get to user data.

Get the same access as the add-on. At first glance, since the host application window does not accept local messages from the add-on UI, it appears that the XSS attacker cannot access the victim's data. However, there is a very common Google API used by add-on UIs, which allows the attacker to bypass this limitation: the Picker API [20]. It is used to select a file or folder that is stored in Google servers. Like any other Google APIs, the Picker API requires an access token to operate. Add-on servers commonly create an interface by which the add-on code running in the browser can obtain a copy of the server's token; this is even a recommended practice [21]. The injected XSS code can request the access token from the same interface. Since the server's token is not limited to the Picker API, the attacker gains the same permissions to the user's data as the add-on server.

Request an OAuth 2.0 token. If the vulnerable add-on does not use the Picker API, the attacker can turn the injected script into a Google APIs JavaScript client and request an OAuth 2.0 token from the user. In that case, an authorization prompt is displayed to the victim in the same way as when an add-on requests permissions during its first run (see Figure 5). While the attacker must use his own GCP project as the malicious client, he can choose a project name that matches the add-on's name.

This is similar to the phishing exploit that we presented for Office Online add-ons in Section 4.1.1. We believe that this attack will have a high success rate because G Suite users are already familiar with the authorization prompt, and the victim might think that the add-on has been updated and needs new permissions.

4.2.2 *Case study: Form Ranger.*

Form Ranger [14] has the highest number of users for any Google Forms add-on. Google Forms is an online service that helps collect information from users via surveys and quizzes, and the add-on allows the user to populate such forms with data from a spreadsheet in the user's Google Drive. The workflow for the add-on is as follows.

(1) The user opens a form and enables the add-on. The host application displays the add-on as a side panel in its UI.
(2) The add-on shows the list of questions in the form. The user selects a question.
(3) The add-on displays a list of all the spreadsheets in the user's Google Drive. Note that the Picker API is used here.
(4) The user selects a spreadsheet document and a sheet and column in it for importing data. A preview of the data is displayed.
(5) The add-on populates the question with the data in the column.

At this point, the readers can probably guess the vulnerability. In Step 4, the add-on does not filter or escape the data from the selected spreadsheet document and renders it as HTML. Thus, if the attacker has access to the shared document, he can hide malicious JavaScript in the part of the document that will be used as form input. The code will be executed when the victim enables the add-on and it renders the data preview. Since the add-on uses Picker API (Step 3), the attacker can steal the access token and gain all the add-on's permissions, which include the following: "See, edit, create, and delete all of your Google Drive files", "See, edit, create, and delete your spreadsheets in Google Drive", "View and manage your forms in Google Drive", and "Send email as you".

In addition to having an XSS vulnerability, the add-on obviously asks for more permissions than it needs. For example, it does not need the ability to send email on the user's behalf. It also does not need full access to the user's Google Drive files. Read access to the specific spreadsheet would be sufficient and significantly reduce the damage caused by the XSS attack.

If the attacker wants to gain even more access rights, e.g. read the victim's emails, it can phish to obtain an OAuth 2.0 token in the same way as in the previous section.

4.3 Shopify add-ons

Shopify is an e-commerce platform with which small merchants can create online shops. It offers services such as payment, marketing, and customer engagement. Each shop is managed through a *web admin interface*, on which the owner can access the built-in services of the platform, for example, to list new products and to communicate with customers. Shopify add-ons integrate third-party services into this admin interface.

The add-on architecture is illustrated in Figure 6. The add-on server runs in the cloud and accesses the shop data with the Shopify REST APIs over HTTPS. Typically, the add-on server is authorized to access the Shopify server with an OAuth 2.0 *access token*, which it obtains as follows. When the user (i.e. the merchant) starts the

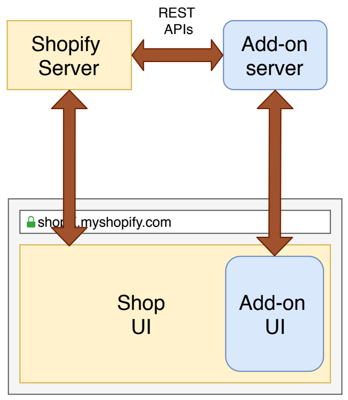


Figure 6: Shopify add-on architecture

add-on for the first time, the shop admin interface shows an authorization prompt with a list of the data to which the add-on requests to access (e.g. orders, products). If the user agrees, the Shopify server sends an access token to a pre-registered endpoint on the add-on server. The add-on server can then use the token to access the requested data. Later, the add-on can renew the access token without prompting the user.

The add-on UI is embedded into a menu in the shop admin interface. Shopify provides a set of local JavaScript APIs [50] for the add-on UI to perform resource-picking operations. If a workflow in the add-on requires the user to select a resource, such as a product, from the shop, the add-on UI communicates with the shop admin interface via `window.postMessage()`, and the admin interface asks the user to pick a resource of the requested type. The admin interface then returns the result data object to the add-on.

A shop on Shopify can have an owner and multiple staff members. The shop owner has full access to the shop and grants the other staff members access to some (or all) admin features including add-ons [51]. The view through an add-on UI is based on the intersection of the staff member's and the add-on's permissions.

4.3.1 XSS exploits. With Shopify, the attacker must be a staff member who is able to inject malicious scripts into the shop resources (the *shared workspace* attack vector). We found that Shopify prevents including HTML or scripts in customer information, but it is possible to inject scripts into the product and order descriptions. These scripts will be executed if other staff members use a vulnerable add-on that renders them in an unsafe way.

Some design choices in the Shopify add-on architecture make it harder to exploit XSS vulnerabilities in add-ons. First, the access token is sent directly to the add-on server and, thus, is not accessible to the add-on UI by default. Second, the local JavaScript APIs allow the add-on UI to perform resource-picking operations without exposing the shop data because it is processed in the admin interface. Nevertheless, there is a feature that can be exploited.

Install a malicious add-on. The attacker can trick the shop owner or an authorized staff member into installing a malicious add-on. Unlike MS Office Online and G Suite, where add-ons can only be installed from the respective marketplaces, Shopify allows users to initiate the installation process just by visiting a URL. Thus, the attacker can create a malicious add-on with a name similar to that of the vulnerable add-on and initiate its installation from the injected script. The victim might think that the add-on has been updated and it needs to be authorized again. Depending on the extent of the permissions granted, the malicious add-on may then be able to access any shop data.

4.3.2 Case study: Order Printer Pro. Order Printer Pro [15] is one of the most popular add-ons in the "Orders and shipping" category on the Shopify marketplace. The add-on allows its users to quickly create order-related documents (e.g. invoices, return forms) as well as printing or delivering them to the customers. The workflow for a staff member to print an order is as follows.

(1) The staff member starts the add-on from the admin interface. The host application displays the add-on as a part of its UI.
(2) The staff member selects the Orders menu from the add-on UI. The add-on shows a list of the orders of the shop.
(3) The staff member selects the order to print. The add-on displays the order as well as action buttons, which allow printing the order or exporting it into a PDF document. The displayed information includes the customer's shipping address, the items in the order, and any notes written by the staff members.
(4) The staff member clicks on the Print button, and the order is printed.

The vulnerability in the add-on is that, when it displays the order information, it does not render the staff members' notes in a safe way. Consequently, if the notes contain JavaScript snippets, they are executed. As we mentioned above, Shopify does not prevent including HTML or scripts in the field. Thus, any staff member who has write access to the orders can perform XSS on other staff members including on the shop owner.

5 EMPIRICAL ANALYSIS

To find out how common the XSS vulnerability is in cloud-application add-ons in the wild, we conducted an empirical analysis by looking for vulnerable add-ons in the marketplaces of the three selected cloud application suites. In this section, we explain how we chose the add-ons for the analysis, followed by the methodology that we used for systematically finding vulnerabilities in them. Finally, we show the analysis results.

5.1 Selecting add-ons for analysis

We consider only free add-ons from the marketplaces. Table 2 shows the number of such add-ons as of August 2019. Note that Microsoft and Shopify officially use the terms *add-in* and *app*, respectively.

Table 2: Add-on marketplaces (August 2019)

Marketplace	Terminology	Available free
MS Office Online	add-in	1150
G Suite	add-on	1180
Shopify	app	1265

We selected 100 free add-ons from each of the three marketplaces. 50 were popular ones (the available popularity metric varied between the marketplaces), and another 50 were selected randomly. In

Table 3: Vulnerable add-ons found in our empirical analysis

Marketplace	Selection criterion	Vulnerable add-ons	Attack vector	Status
MS Office Online	Popular	Translator for Outlook	Outside input	Fixing
		GIGRAPH - Network Visualization	Shared workspace	No response
	Random	Duplicate Remover	Shared workspace	Fixing
		Bubbles	Shared workspace	No response
		Radial Bar Chart	Shared workspace	No response
		Excel to JSON	Shared workspace	No response
		WritingAssistant	Shared workspace	No response
		Excel to SMS	Shared workspace	No response
G Suite	Popular	Form Ranger	Shared workspace	Fixing
		Flubaroo	Shared workspace	No response
		autoCrat	Shared workspace	Fixing
		formMule - Email Merge Utility	Shared workspace	Fixing
		docAppender	Shared workspace	Fixing
		Grackle Sheets	Shared workspace	No response
		Sheetgo	Shared workspace	Fixing
	Random	Form Duplicates	Shared workspace	No response
		Bulk Sheet Manager	Shared workspace	No response
		rosterSync - Teacher Edition	Shared workspace	Fixing
		Notifications for Forms	Shared workspace	No response
		Text gBlaster (SMS Texting)	Shared workspace	No response
		Mail Merge	Shared workspace	No response
		Response Editor	Shared workspace	No response
		Doc Variables	Shared workspace	Fixed
Shopify	Popular	Order Printer Pro	Shared workspace	No response
	Random	ShipHero Fulfillment	Shared workspace	Fixing
		Simple Admin	Shared workspace	No response
		ShipRelay Fulfillment	Shared workspace	No response
		Ship Systems 3D Box Packing	Shared workspace	No response

total, 300 add-ons were selected for the study. The logic behind this selection is that the popular add-ons represent the typical user exposure, while the randomly selected ones cover the entire spectrum of applications and code quality. The detailed criteria for selecting the add-ons were the following.

MS Office Online: We focused on add-ons for the following applications in the suite: Word, Excel, OneNote, Outlook, and PowerPoint. The remaining applications either have no add-ons or the add-ons are available only to domain users. Since the marketplace does not show the number of users for each add-on, we selected the top 50 add-ons based on the number of reviews.

G Suite: The applications available to individual users include Gmail, Docs, Sheets, Slides, Forms, Drive, and Calendar. We excluded Drive and Calendar because their add-ons only add new menus to the application UI, which means that they do not have any client-side code, thus leaving no room for the XSS attacker. For the remaining applications, we selected 50 add-ons with the highest number of users.

Shopify: We first sorted the add-ons in the marketplace by the "Most installed" option and selected the 50 top ones. Note that the marketplace does not show the number of installations of each add-on; instead, it shows only the rating and the number of reviews.

5.2 Analysis methodology

We manually analyzed the selected add-ons with *black-box testing* as follows. For each add-on, we first installed it and tried to understand its features. We then created a test item for each of the target applications:

- MS Office Online and G Suite: For document editing applications (Word, Google Sheets etc.), the test item was a document. To make sure that every source of data that the add-on was going to process contained JavaScript code, we added simple JavaScript snippets to every part of the document. For instance, if it was a spreadsheet, we included JavaScript code in the name of each sheet, the heading of each column, and some cells in each column. An example of the snippets is `<script> console.log("Pwned!!!") </script>`.
  For email applications (i.e. Outlook, Gmail), the test item was an email that we sent to ourselves. We inserted scripts to the subject and the body of the email.
- Shopify: We first created a test shop. As we mentioned in Section 4.3.1, it is possible to injects scripts only into the products and orders. Thus, we created several test products

and orders in the shop and added scripts into every possible location, such as the name and description of the items.

After creating the test item, we tried all functions of each add-on on the test item and looked for workflows that involve rendering the content of the item. We also added more customized JavaScript code in a number of cases because some add-ons only accept data in specific formats. For example, for the Doc Variables add-on [34], which allows defining and using variables in Google Docs documents, we had to insert our script into a variable definition, which has the format `${variable_name}`. If any of the injected script snippets were executed, we concluded that the add-on is vulnerable to XSS.

Challenges with automated verification. While our analysis was manual, we believe that it is sufficient to find most, if not all, XSS vulnerabilities in the selected add-ons. The main reason is that, compared to standalone web services, add-ons are usually quite simple with a relatively small number of features. The number of places where the malicious user can inject scripts is also limited. Thus, it is not difficult to understand and manually test all the workflows of an add-on with the black-box testing method. However, while our methodology is sufficient to produce results that, in our opinion, need wider attention, it would not be practical for a large-scale analysis of cloud-application add-ons. For example, it could not cover all the add-ons in the studies marketplaces. An automatic (or semi-automatic) solution is obviously needed for such purpose. We will discuss here the challenges that such an automated solution will have to overcome.

First, state-of-the-art automated techniques for detecting XSS vulnerabilities in the web typically work with client-side XSS [33, 35, 49, 53], whereas the XSS attacks against cloud-application add-ons may also exist on the server side. For example, the vulnerabilities of MS Office Online add-ons are always on the client side because the attacker's malicious input is propagated to the add-on via local messaging (i.e. `window.postMessage()`) and processed by the add-on UI. The vulnerabilities of G Suite and Shopify add-ons, on the other hand, can be either on the client side or on the server side depending on whether the add-on server returns the attacker's raw input to the client for further processing or renders the data beforehand.

Second, unlike traditional web applications where a significant portion of the workflows can be analyzed by loading every available URL in the websites with a browser, triggering add-on functionality usually requires user interaction. In addition, there is no single standard method to invoke an add-on. Some start by the user clicking and selecting the add-on from a host-application menu while the others require further user interaction to reach a point where the XSS code may be executed.

Third, most automated XSS detection approaches in the literature do not cope with user registration and logging in. Many add-ons act as the connectors between their host applications and third-party web services; thus, the users must create an account and log in to authorize the add-on. The registration process varies greatly, making them difficult to automate.

5.3 Results and responsible disclosure

We found 28 vulnerable add-ons among the 300 analyzed ones, which is around 9%. The result indicates that XSS vulnerabilities are common in cloud-application add-ons today. Table 3 shows the names of the add-ons, the attack vectors, and whether the add-ons have been fixed. Among the three marketplaces, only G Suite shows the number of users for each add-on. The vulnerable G Suite add-on with the greatest number of users, Form Ranger, had roughly 7.8 million users as of August 2019. The most popular vulnerable add-ons in the Microsoft Office Online and Shopify marketplaces are Translator for Outlook and Order Printer Pro, respectively. The former had 1772 reviews, and the latter had 371 reviews.

We observed that the vulnerability rate in the set of popular add-ons is lower in all the three marketplaces. This could be because the popular add-ons are more likely to be written by experienced developers. Also, it seems that add-ons that are vulnerable to outside input are rare. Specifically, only one add-on in our study is vulnerable. This could be because the add-on developers are more familiar with threats from external input (i.e. emails in this case) than those from shared workspace.

We have disclosed all the vulnerabilities to the corresponding developers (their contact information is available on the marketplaces). We also provided guidance on how the security bugs could be fixed (see the solutions for add-on developers in Section 6). We contacted all 28 vulnerable add-on teams or developers, and the current status of their response is tabulated in the Status column of Table 3. At the moment of writing, 1 has acknowledged and fixed their add-on, 9 have acknowledged the vulnerabilities but are still working on the fixes, and the rest have not responded. We also discussed the problem of the Picker API with Google. They confirmed the problem and said that they would take it into account in the next version of their add-on system.

6 DEFENSES

In this section, we discuss what the add-on developers and cloud application vendors can do to defend against the XSS attacks caused by vulnerable add-ons.

6.1 Solutions for add-on developers

To prevent XSS, add-on developers should not add untrusted data to the add-on UI as HTML because it can contain malicious JavaScript code. We discuss some practices that they can follow below.

Coding practices. Secure coding practices to prevent XSS are a common topic in web security literature [24, 41, 47]. A straightforward way to prevent XSS in cloud-application add-ons is to always render user input as text instead of HTML. Instead of the `innerHTML` property, the developer should use the `innerText` and `textContent` properties to insert text. With jQuery, the `.text()` method should be used instead of the `.html()` method.

In general, user data should not be interpreted as web application code. However, in the rare cases where it is necessary to render untrusted HTML as part of the add-on UI, the developer needs to properly validate and escape on the input first. On the server side, most web frameworks have built-in functions for such tasks. On the client side, JavaScript methods like `.toStaticHTML()` can be used to remove dynamic HTML elements and attributes in the user data before rendering it. We refer to [47] for a detailed guidance on how to escape characters to prevent XSS.

Security enforcement. Since add-ons are basically web services, add-on developers could implement a Content Security Policy (CSP) [52] to enforce some defenses on their add-ons. An extreme policy is to ban the execution of all inline scripts (e.g. `<script>` tags and inline event handlers). With that policy, even if the attacker managed to insert malicious scripts to the add-on UI, the scripts would not be executed. Only JavaScript in separate `.js` files and downloaded from whitelisted servers would be allowed to run.

Completely prohibiting inline scripts is not always an ideal solution because inline scripts are useful for various legitimate tasks. For example, event handlers are usually implemented in inline scripts. Moving inline event handlers to separate `.js` files cannot be done by simply copying and pasting the code; instead, they need to be rewritten with DOM APIs. Fortunately, CSP can be used in a less stringent way to avoid the hassle. Specifically, hash-based or nonce-based policies can be implemented so that inline scripts with pre-registered hash or nonce values are allowed to execute.

Add-on developers should minimize the permissions that their add-ons request. For example, unnecessary permissions in Form Ranger (Section 4.2.2) enable the XSS attacker to steal all of the victim's Google Drive files as well as sending emails on the victim's behalf.

There are other generic practices that help to defend against XSS [47]. The `HTTPOnly` flag should be set on session cookies and any custom cookies that do no need to be accessed from client-side JavaScript. Also, many web frameworks provide automatic escaping functionality [5, 16], which should be used whenever possible.

XSS detection. Add-on developers can also utilize the testing method from our empirical analysis (Section 5.2) to check their own add-ons. Specifically, they can create test items similar to ours and write test cases that continuously check for XSS vulnerabilities during their development cycle. While this method does not scale to very large bodies of add-ons, it is entirely feasible for testing individual add-ons by the developer.

6.2 Lessons for cloud-application vendors

This section presents some lessons for the developers of add-on systems in cloud applications. There are several platform-level design choices that can reduce the vulnerability of even poorly written add-ons to XSS.

Harden the add-on `iframe`. The add-on UI is contained in an `iframe`, and by default, the code in the `iframe` can call browser APIs to request access to features on the local device including geolocation, microphone and camera. The host application should restrict the browser features accessible to the add-on `iframe`. This can be done by setting the `allow` and `sandbox` properties of the `iframe` [42].

If the host application could set a CSP on add-ons to prevent execution of unwanted inline scripts, the XSS problem would be solved. At the moment, there is an experimental feature in the Chrome and Opera browsers that allows the host application to set such a policy. More specifically, the browsers added a new `csp` attribute to the `iframe` element. It can be used to specify a policy which the embedded page must enforce upon itself [60]. If this property becomes a standard feature in browsers, it gives cloud-application vendors control over the CSP policy in their add-ons. For example, they could prohibit inline scripts. On the negative side, deploying a strict CSP policy would break most add-ons currently in the marketplaces and require non-trivial updates to many of them. Thus, it might not be an attractive option for the marketplace owners.

Implement add-on logic in the add-on server, not in client-side JavaScript. Cross-origin messaging within the browser, as in MS Office Online, enables low-latency access from the add-on to user data that is available in the main application UI. On the negative side, this access is only controlled by the generic permissions shown in Table 1. These permissions may not be able to express all access policies that depend on the application-specific semantics of the requested operations. On the other hand, if the add-on logic is implemented on the server side, as in G Suite and Shopify, the add-on server will act as a layer of isolation between the client-side script and user data. It does this in two ways. First, the add-on server defines a limited, purpose-specific interface through which all access to user data has to go. Second, the add-on server implements business logic that further filters the kinds of read and write operations that are passed on to the cloud application. Both mechanisms act as filters between the user's data and any XSS code that may have taken over the add-on `iframe` in the browser.

Filter scripts in user input. The cloud-application vendors should think thoroughly about the types of user input that their applications need to receive in a shared workspace and filter out unwanted types. For example, Shopify deals with shop resources such as products and orders, which are relatively structured data. Thus, the developers know where HTML or scripts should not appear. The G Suite and MS Office Online applications are more problematic in this respect because the input to them is mostly documents, where legitimate HTML and scripts can appear in unexpected places.

Do not share access tokens to delegate all your permissions. OAuth 2.0 tokens are bearer tokens, and anyone in possession of the token can use it for resource access. This can lead to unsafe coding practices where too powerful tokens are delegated to unsafe places — as we saw in the case of the Picker API in G Suite. Instead of sharing its access token with the client side, the add-on server should mediate the access. Where tokens need to be shared, they should only convey the absolute minimum permissions needed by the client side. In the case of the Picker API, for example, the add-on server should delegate to the UI a restricted token that can only be used to list specific files or folders in the user's Google Drive.

Avoid asking user consent at runtime. Relying on user judgment when authorizing add-ons to access user data may not be as good an idea as it first seems. In particular, prompting the user for consent at runtime conditions the user to answering yes to every such prompt, including ones from injected malicious code. Thus, it may be better to ask for user consent only in a separate UI where the user installs or updates add-ons.

7 DISCUSSION AND FUTURE WORK

Since cross-site scripting is a well-known vulnerability, prudent engineering practices have been developed to prevent such mistakes [48]. On one hand, developers are aware of the need to filter untrusted input, and on the other, cloud application vendors have developed platforms and toolchains that make their products immune to most types of code injection. Nevertheless, the problem has not been completely solved. Attackers always find new ways of bypassing the defenses, and the speed of software development makes it difficult for threat analysis and defenses to stay up to date. This paper is doing its part to catch up with the development in one key area of modern software, i.e. add-on ecosystems.

We have confirmed by experiments that the vulnerabilities described in this paper are real and exploitable. There are, however, some additional practical considerations which a real-world attacker would face. The attacker needs to know which vulnerable add-on the victim is using, and the victim has to enable the add-on for the shared document. Thus, a successful attack probably requires a vulnerable add-on that users regularly invoke on large classes of documents which they are reading or editing. Translator and writing-assistant add-ons could meet these criteria. Add-ons that fix problems in data, such as duplicate removers, could even be installed by the victim when they receive a document with the corresponding problem.

In addition to the XSS vulnerabilities in the add-ons, another serious issue discovered in this paper is the way OAuth 2.0 tokens are used in the Picker API, and the powerful exploits that it enables for the XSS attacker. The Picker API documentation has educated developers to use a design pattern where an add-on server shares its OAuth 2.0 token with a client-side script. Even if the Picker API itself is replaced with a safer solution, this unsafe software pattern might persist among developers.

For these reasons, some further measures may be needed to prevent unsafe use of the access tokens. G Suite has the advantage that the add-on server runs in the Google cloud, and the access token is handled by the third-party code only in special cases, which could be monitored and blocked. Also, host applications could relatively easily reject tokens that come from somewhere other than the authorized add-on server. Such restrictions on the token usage would, however, take away the convenience and flexibility of bearer tokens that has made them popular with developers. Indeed, the designers of OAuth 2.0 have intentionally not built in support for channel binding, such as binding the token to a specific client address. In conclusion, there are ways of mitigating the threat caused by unsafe delegation of access tokens to client-side scripts, but it might take some time for the problem to go away entirely.

Overall, we hope that this paper will attract more attention to the area of cloud-application add-ons because further work is clearly needed. There might be other attack vectors that allow the attacker to exploit non-malicious add-ons. Analyzing threats from malicious add-ons could also be an interesting area for future work.

8 RELATED WORK

In this section, we survey the related literature about XSS attacks and security analysis of add-on ecosystems outside the domain of web applications.

XSS vulnerabilities. XSS has been one of the most common and harmful vulnerabilities in web applications. In spite of the availability of detection and defense mechanisms and changes in the architecture of web applications, XSS remains a prevalent problem [13, 53]. Security research literature on XSS includes a comprehensive overview [25], detection mechanisms [28, 30, 58], as well as preventive and defensive solutions [10, 31, 44, 55–57].

The rich literature on defenses against XSS includes both client and server-side solutions. Client-side solutions involve sanitizing user input before it is sent to the server. However, distinguishing between trusted and untrusted content and filtering out any malicious scripts are challenging tasks. This is why the sanitation of web pages is sometimes outsourced to the browsers [55] or to web firewalls that run on the client PC [28, 31].

Taint checking is a server-side protection mechanism where the input originating from untrusted sources is flagged as potentially malicious and subjected to further scrutiny (e.g. sanitizing the input). Similar techniques can be employed on the client side if combined with static analysis of the input processing [57, 58]. There are other server-side solutions that involve, for example, passive monitoring of the HTTP traffic [30] or dynamic comparison HTTP responses with a pre-defined response [10].

Add-ons outside web applications. Even though there are add-ons for almost any type of software, it is mostly the browser add-ons, which have undergone critical security scrutiny. For example, Google Chrome has a *browser extension* ecosystem, where the extensions themselves [29], their architecture [11] and protection mechanisms [27] have been targets of security research. Weissbacher et al. inspected Chrome extensions for privacy violations (e.g. leaking browsing history) by observing the network traffic patterns generated by the add-ons [59]. Firefox add-ons have also undergone similar vetting [7–9]. As more applications are moving to the cloud, we believe that cloud-application add-ons deserve the same attention from the security research community as browser add-ons.

Text editors also have add-on ecosystems (e.g. Sublime plugins) that have been recently criticized for security vulnerabilities. Azouri Dor analyzed several text editors and found that it is possible for a malicious add-on to achieve privilege escalation on the victim's computer [6]. The attack vector here involves crafting a malicious add-on and tricking the victim into installing and using it within the text editor. Our attacks, on the other hand, simply involves injecting a malicious script in an online document or item to be shared with the victim.

9 CONCLUSION

Add-ons in cloud applications are a relatively new phenomenon whose vulnerabilities have not been widely studied. In this work, we analyzed the security of such add-ons and found that flaws in the add-ons may introduce new security threats to their host applications. In particular, the add-ons do not always take care when processing untrusted input, which can make them vulnerable to XSS attacks. The attacker can inject malicious scripts into shared documents or emails, which are then processed by the vulnerable add-on. We found vulnerable add-ons in the wild and showed that exploiting them is not difficult. Moreover, we suggest that cloud-application vendors could do more to limit what the attacker can achieve once its XSS code is running in a vulnerable add-on.

REFERENCES

[1] G Suite marketplace. https://gsuite.google.com/marketplace.
[2] Google Docs. https://www.google.com/docs/about/.
[3] Microsoft AppSource. https://appsource.microsoft.com/en-us/marketplace/apps.
[4] Shopify App Store. https://apps.shopify.com/.
[5] AngularJS. AngularJS Strict Contextual Escaping. https://docs.angularjs.org/api/ng/service/$sce.
[6] Dor Azouri. Abusing text editors via third-party plugins. https://go.safebreach.com/rs/535-IXZ-934/images/Abusing_Text_Editors.pdf. [Accessed: 2019-08-20].
[7] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. 2010. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX Security Symposium.*
[8] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting browsers from extension vulnerabilities. In *The Network and Distributed System Security Symposium (NDSS).*
[9] Anton Barua, Mohammad Zulkernine, and Komminist Weldemariam. 2013. Protecting web browser extensions from JavaScript injection attacks. In *2013 18th International Conference on Engineering of Complex Computer Systems.* IEEE.
[10] Prithvi Bisht and VN Venkatakrishnan. 2008. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer.
[11] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An evaluation of the Google Chrome extension security architecture. In *21st USENIX Security Symposium.*
[12] CERT. 2000. Advisory CA-2000-02 malicious HTML tags embedded in client web requests. (2000).
[13] XiaoLong Chen, Mohan Li, Yu Jiang, and Yanbin Sun. 2019. A Comparison of Machine Learning Algorithms for Detecting XSS Attacks. In *International Conference on Artificial Intelligence and Security.* Springer, 214–224.
[14] New Visions Cloudlab. "Form Ranger" add-on. https://gsuite.google.com/marketplace/app/form_ranger/387838027286.
[15] Forsberg+two. "Order Printer Pro" add-on. https://apps.shopify.com/order-printer-pro.
[16] Google. Go Templates. https://golang.org/pkg/html/template/.
[17] Google. Google APIs Explorer. https://developers.google.com/apis-explore.
[18] Google. Google Cloud Platform Projects. https://developers.google.com/apps-script/guides/cloud-platform-projects#standard_cloud_platform_projects.
[19] Google. Quickstart: Add-on for Google Docs. https://developers.google.com/gsuite/add-ons/editors/docs/quickstart/translate.
[20] Google. What is Google Picker? https://developers.google.com/picker/.
[21] Google. Dialogs and Sidebars in G Suite Documents. https://developers.google.com/apps-script/guides/dialogs#file-open_dialogs.
[22] Google. Google API Services: User Data Policy. https://developers.google.com/terms/api-services-user-data-policy.
[23] Google. OAuth client verification. https://developers.google.com/apps-script/guides/client-verification.
[24] Jeremiah Grossman, Seth Fogie, Robert Hansen, Anton Rager, and Petko D Petkov. 2007. *XSS attacks: cross site scripting exploits and defense.* Syngress.
[25] Shashank Gupta and Brij Bhooshan Gupta. 2017. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management* (2017).
[26] Dick Hardt. 2012. *The OAuth 2.0 Authorization Framework.* RFC 6749.
[27] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. 2015. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems.*
[28] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. 2004. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *18th International Conference on Advanced Information Networking and Applications.* IEEE.
[29] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. 2015. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium.*
[30] Martin Johns, Björn Engelmann, and Joachim Posegga. 2008. XSSDS: Server-side detection of cross-site scripting attacks. In *2008 Annual Computer Security Applications Conference (ACSAC).* IEEE.
[31] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. 2006. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing.* ACM.
[32] Amit Klein. 2005. DOM Based Cross Site Scripting or XSS of the Third Kind. *Web Application Security Consortium* (2005).
[33] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-scale Detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM.
[34] Jesse Mccabe. "Doc Variables" add-on. https://gsuite.google.com/marketplace/app/doc_variables/232821636920.
[35] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting. In *NDSS.*
[36] Microsoft. Microsoft Azure. https://azure.microsoft.com.
[37] Microsoft. Microsoft Identity Platform (v2.0) Overview. https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-overview.
[38] Microsoft. Office Online Permissions. https://docs.microsoft.com/en-us/office/dev/add-ins/reference/manifest/permissions.
[39] Microsoft. "Translator for Outlook" add-on. https://appsource.microsoft.com/en-us/product/office/WA104380627.
[40] Microsoft. Use the Outlook REST APIs from an Outlook add-in. https://docs.microsoft.com/en-us/outlook/add-ins/use-rest-api.
[41] Microsoft. Privacy and security for Office Add-ins. https://docs.microsoft.com/en-us/office/dev/add-ins/concepts/privacy-and-security.
[42] Mozilla. The Inline Frame element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.
[43] Mozilla. window.postMessage() method. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.
[44] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*, Vol. 20.
[45] OWASP. Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).
[46] OWASP. Types of Cross-Site Scripting. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting#Types_of_Cross-Site_Scripting).
[47] OWASP. Cross-site scripting prevention. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.
[48] OWASP. XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.
[49] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS.*
[50] Shopify. App Bridge library. https://help.shopify.com/en/api/embedded-apps/app-bridge.
[51] Shopify. Staff account permissions. https://help.shopify.com/en/manual/your-account/staff-accounts/staff-permissions.
[52] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the Web with Content Security Policy. In *Proceedings of the 19th international conference on World wide web.* ACM.
[53] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS.*
[54] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. 2015. From Facepalm to Brain Bender: Exploring Client-side Cross-Site Scripting. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security.* ACM.
[55] Mike Ter Louw and VN Venkatakrishnan. 2009. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *30th IEEE symposium on security and privacy.* IEEE.
[56] Matthew Van Gundy and Hao Chen. 2009. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *NDSS.*
[57] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS.*
[58] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering.* IEEE.
[59] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. 2017. Ex-ray: Detection of history-leaking browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference.* ACM, 590–602.
[60] Mike West. Content Security Policy: Embedded Enforcement. https://w3c.github.io/webappsec-cspee/#dom-htmliframeelement-csp.
[61] Michal Zalewski. 2012. *The Tangled Web: A Guide to Securing Modern Web Applications.* No Starch Press.