Technical Report

System Security Lab, School of Computing

National University of Singapore

You Can't Be Me: Enabling Trusted Paths & User Sub-Origins in Web Browsers

Enrico Budianto, Yaoqi Jia,
Prateek Saxena and Zhenkai Liang

April 10, 2014

You Can't Be Me: Enabling Trusted Paths & User Sub-Origins in Web Browser

Enrico Budianto[1], Yaoqi Jia[1], Xinshu Dong[2], Prateek Saxena[1], Zhenkai Liang[1],

National University of Singapore[1], Advanced Digital Sciences Center[2]
{enricob,jiayaoqi,prateeks,liangzk}@nus.edu.sg, xinshu.dong@adsc.com.sg

Abstract. Once a web application authenticates a user, it loosely associates all resources owned by the user to the web session established. Consequently, any scripts injected into the victim web session attains unfettered access to user-owned resources, including scripts that are coming from malicious activities inside a web application. In this paper, we establish the first explicit notion of *user sub-origins* to defeat such scripts. Based on this notion, we propose a new solution called UserPath to establish an end-to-end trusted path between the web application users and web servers. To evaluate our solution, we implement a prototype in Chromium, and manually retrofitted 20 popular web applications to use UserPath. UserPath reduces the size of client-side code that has access to user-owned resources by 8x to 264x, with minimal developer effort.

1 Introduction

Many of the web applications today, such as DropBox, Gmail and Facebook, provide user-oriented services, where users need to create their own accounts to use the service tailored to them. User-oriented web applications strongly isolate data belonging to individual users and bind access control privileges to specific user accounts (e.g., owners or administrators). In today's web applications, the authority of a user is typically represented by a session, and the security mechanisms are centered on protecting the session state from being accessed by attackers. The gap between the notion of user authorities and the representation of web sessions makes it possible for an attacker to exploit script injection vulnerabilities in order to hijack user-owned resources or perform unauthorized request. Script injection can happen in a variety of ways, such as mixed content, browser extensions, third-party libraries, and XSS vulnerabilities. In this paper, we do not focus on mechanisms to prevent web application vulnerabilities from occurring. Rather, we propose mechanisms to defend against *post-attack* malicious behavior of an injected script, which we term as *post-injection script execution* (PISA). Our proposal serves as a second line of defense for post-injection script execution when existing mechanisms of script injection prevention, such as CSP [**?**], fail to achieve full coverage [**?**].

PISE attacks cover a set of attacks under the assumption that the attacker has already injected code or requests in the web session. This post-attack malicious behavior is a strong and challenging attack model: it runs under the same

origin of the web application. In this work, we consider PISA attacks that targets sensitive data owned by users and mimics normal user interactions within a web application. For instance, XSS worms on on Facebook profiles that use self-XSS attacks to befriend certain users [**?**] or malicious extensions that stealthily forge transactions to illicitly buy goods on Amazon [**?**] are some of the real examples.

A recent line of research has proposed piecemeal defenses to mitigate some classes of PISE attacks via client-side channels [**?**, **?**], server-side channels [**?**, **?**], self-exfiltration channels [**?**], or via mimicking user interactions to legitimize dangerous information flows [**?**]. However, none of them offer a comprehensive solution to prevent this class of attacks. We observe two fundamental limitations of the web platform. First, web browsers rely on *same-origin policy* to control access of DOM, which lacks the notion of a *user sub-authority* that controls access to sensitive user-owned resources. Second, there is no direct way for the server-side web application to be faithfully informed about what is happening at the client-side code. As a result, web servers cannot, for example, distinguish between client requests generated in response to legitimate user interaction versus requests generated by injected scripts, even in the presence of network-level protection mechanisms like HTTPS.

Our Solution. We propose a solution called UserPath, which augments the present web with security primitive that represents ***User*** authority and establishes an end-to-end *trusted* ***Path*** between the user and the server. We bring in the first explicit notion of *user sub-origins*[1] into web applications, which are primitives that run in the authority of web application users. Our mechanism enables user sub-origins to isolate user's data and privilege-separate the code operating on it from the rest of the web origin. To complete our end-to-end system, we introduce trusted path [**?**] as an abstraction between human users and the web application server. A trusted path is a privileged channel, which allows the server to tightly and faithfully control the communication of visible content and input with the user (via the standard DOM APIs), even in the presence of malicious code. Although this concept has recently been explored to develop new access control mechanisms on mobile operating systems [**?**, **?**], adapting it to the web has been an open problem.

Our solution is practically deployable – we keep high compatibility with existing browsers, users, and developers. To keep compatibility with existing browsers, we reuse the existing web isolation primitives and minimize new abstractions that developers need to use (e.g. familiar notion of iframes). We ensure UserPath-enabled browsers to be backward-compatible with non UserPath-enabled websites. From the user's perspective, using a UserPath-enabled website should be largely identical to the original site, except for verifying a colored login input box when authenticating with a password (see Section 3.4). As a result, UserPath

[1] Recently, browsers have added support for per-page sub-origins [**?**] that compartmentalize contents on a web page within several sub-authority under the same origin. Per-page sub-origin proposal offers no guarantee to defend against post-injection script execution, and we complement per-page sub-origins with the additional notion of user authority and trusted path.

has a much lower adoption cost as compared to another recent trusted-path proposal that requires generation and uploading of SSL keys for every website [**?**]. Furthermore, our solution can also be easily deployed without much development effort. Specifically, developers can easily retrofit UserPath to their web applications simply by privilege-separating sensitive data and JavaScript logic on a client-side user-suborigins called UFrame. UFrame is an `iframe`-like component that isolates code under different JavaScript context and has the ability to render tamper-proof HTML elements. Such privilege separation of JavaScript code is straightforward for developer to use, as argued in recent works [].

From a security standpoint, users no longer trust a website at the time of login if script injection vulnerabilities are present in the website. Then, how does a user login and setup an authenticated trusted path? We address this critical issue by introducing secure UI elements and PAKE protocol [**?**], a *zero knowledge proof* protocol that lets two parties to authenticate each other without revealing secret information through the communication channel. Having authenticated the user, UserPath maintains isolation of sensitive resources throughout sessions by resorting to user sub-origins and trusted path.

Summary of Results. We deploy UserPath on 20 popular open-source web applications. The evaluation demonstrates our solution can protect user-owned data from PISE attacks in these applications with modest adoption effort (in the order of days). For each application, we label a number of data fields as sensitive, and modify the application logic to adopt UserPath. We find that UserPath eliminates the threats to user data from 325 historical security vulnerabilities in these applications, and reduces the TCB size by 8x to 264x. Finally, the performance overhead incurred by our solution is negligible for real-world applications. All case studies and our implementation in Chromium browser is open source and available online [].

Contributions. In summary, we make the following contributions in the paper:

- *User Sub-Origins & Trusted Path.* We propose the first explicit notion of user sub-origins on the web. We further develop an end-to-end trusted path to eliminate post-injection script execution targeting user-owned data.
- *Compatible Solution.* We design a solution UserPath to bring in user sub-origins and the trusted path to today's web platform. This solution leverages mechanisms already available in the web infrastructure to a large extent, and can be smoothly integrated into existing web browsers, web servers, and web applications.
- *Real-world Evaluation.* We implement our solution in Chromium and modify 20 web applications to enable UserPath with a small adoption effort. We release our patch to Chromium and the modified web applications [**?**] for further scrutiny.

2 Problem Definition and Approach Overview

The missing notion of user sub-origins in today's web sessions give rise to various attacks threatening web applications. We summarize such attacks and elaborate how they can occur with an example web application.

2.1 PISE Attacks Targeting User-owned Data

Unlike in traditional OSes (e.g. UNIX), there is no built-in notion of user in the present web. Under this setting, users login into sites and authenticate themselves using custom password-based interfaces. Authentication of subsequent HTTP requests is performed via "bearer tokens", such as sessions IDs, CSRF tokens, or cookies. In the presence of script injection vulnerabilities, these tokens are prone to attacks, either via direct token stealing, phishing attempts [**?**], or session riding (e.g., fake HTTP request). In this paper, we term such illegitimate access from malicious scripts to resources owned by benign victim user as *post-injection script execution* (PISA).

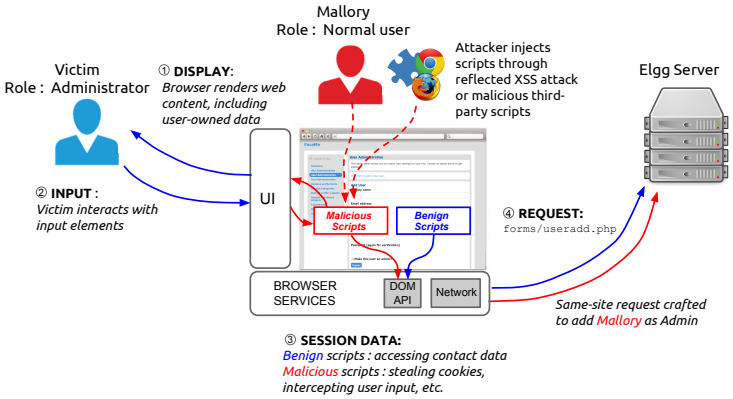

Fig. 1. Example Interactions in Elgg. Blue lines illustrate typical benign interactions between the user, UI elements and session data. Red lines illustrate examples of script-based cross-user attacks, where an attacker injects malicious scripts into victim's session, steals victim's CSRF token, and performs a same-site request forgery attack to Elgg server.

We illustrate various post-injection script execution by an example of open-source social networking application called Elgg[2]. It maintains user profiles, manages private message dispatch and blogging, as well as integrating other social networking sites. We consider the following features available to administrators:

[2] http://elgg.org/

- *Add New User*: This is a privileged feature that can only be accessed by administrator. The administrator specifies information belonging to a particular user that is going to be added to the system. The administrator can also mark the user as new administrator by identifying it on a checkbox element. Thereafter, this particular information is sent to the server using HTML Form submit mechanism.
- *Profile Management*: Elgg provides profile data management to maintain particular information for a user, just like what most social networking applications have. In addition, there is a feature to set other users as an administrator directly from their profile page. However, this feature is privileged to an administrator. The administrator could add another user as an administrator by clicking on "Make admin" link on a user's profile page.

In post-injection script execution, injected scripts can access user-owned resources located at the client side and the server side, as shown in Figure 1. At the client side, we categorize three variants of post-injection script execution depending on different channels that are exposed to an attacker.

- **Display Channel Attacks.** An attacker can tamper with display elements of the web to steal sensitive information from users. Two example of attacks that exploit this channel are UI defacing and phishing for user credentials. In UI defacing attacks, an attacker can alter the web content to mislead users. For instance, a malicious extension can change the appearance of a profile page in Facebook, as reported recently [**?**]. Besides that, malicious scripts could also introduce fake UI elements (such as fake login input) to steal users' credentials, and therefore allows them to impersonate as Alice on a site O. Unlike traditional phishing attacks where a malicious website mimics another benign website, in this example the malicious scripts are running within the victim origin O. Therefore, common UI indicators such as the SSL lock icons and URL bars do not help Alice in detecting a phishing attempt.
- **Input Channel Attacks.** In order to tamper with sensitive data, an attacker can exploit this channel by (1) intercepting or stealing user input; or (2) launching an attack that programmatically interacts with the interface element of the web [**?**, **?**]. In the second scenario, malicious scripts can impersonate a user by forging a user interaction with the DOM element on the web page (e.g. clicking the "add user" button) and mimic the user's action. Another popular attack that exploits this channel (and the display channel) is clickjacking, which typically runs in a different website than Elgg. It can, for instance, load Elgg in a transparent overlay. Then underneath Elgg, it renders another malicious web page to attract users to click on the "Make admin" button in invisible Elgg layer above. Clickjacking attacks sabotage a user's intention to interact with a UI element intended by an attacker.
- **Session Data Channel Attacks.** Malicious scripts injected into the web page have access to arbitrary data. It can exfiltrate sensitive data, including cookies, CSRF tokens, capability-bearing URLs, and passwords, through two

channels: directly to an attacker-controlled website [?] or via the victim website itself, which is recently discussed and termed as self-exfiltration attacks by Chen et.al. [?]. Such leakage happens due to the shortcomings of current browsers to protect sensitive data within a web element. Due to the lack of input sanitization on Elgg's "edit page" functionality [?], we demonstrate one XSS attack [?] to steal cookie data on Elgg and exfiltrate the cookie via public blog entry, which is visible to the attacker.

Besides those three attack variants, the injected scripts also have access to the network, allowing the attacker to access server-side resource of a user.

– **Network Request Channel Attacks.** Malicious scripts can craft and send HTTP requests to the server by invoking `XMLHttpRequest` API, or using HTML's resource tag attributes, such as a `src` attribute in an `<img>` tag. This crafted request can be used to perform specific operations on the server-side application. Some websites implement a CSRF token that is sent along with the HTTP request and the server-side application verifies whether incoming request carry expected CSRF tokens. However, secret CSRF tokens and other existing defenses for CSRF attacks, such as `Referer` and `Origin` header [?], do not suffice for preventing same site request forgery attacks.

2.2 Insufficiency of Existing Solutions

Many existing solutions provide piecemeal defenses against post-injection script execution. In Table 1, we highlight the ability of existing second line of defense techniques to mitigate post-injection script execution. We categorize the defenses based on four channels on the web that are exposed to the attackers due to PISE attacks (Section 2.1). As Table 1 summarizes, none of them provides full protection for all the four channels above against malicious scripts injected into victim web sessions. We refer readers to Section 6 for a detailed discussion on this issue. To the best of our knowledge, we are the first to propose a user-based end-to-end trusted path that comprehensively protects all the four channels.

Table 1. Various Techniques for Mitigating PISE Attacks

	I[1]	II[2]	III[3]	IV[4]		I[1]	II[2]	III[3]	IV[4]
HTML5 Privilege Separation [?]			√		WebWallet [?]		√		
HTML5 Data Confinement [?]			√	√	Secure UI Toolkit [?]	√	√	√	
Object-Capability Sec Model [?, ?]			√		Clickjacking Defenses [?, ?]	√	√		
PathCutter [?]			√	√	Cryptons [?]		√	√	
Request Triggering Attribution [?]		√		√	DOMinator [?]			√	
Adsentry [?]			√		Origin Bound Certificates [?]			√	
UserPath	√	√	√	√					

[1]Display Channel [2]Input Channel [3]Session Data Channel [4]Network Request Channel

2.3 Threat Model & Scope

We now briefly discuss the in-scope threats of our work. We consider the attacker to be a standard *web attacker* [**?**] that is able to exploit script injection vulnerabilities in a web application and browser's add-ons []. All attacker payloads are client-side scripts, and we assume no compromise on the web server and web browser, as well as the underlying OS. We assume that the user is *benign*, i.e., we do not aim to prevent an attack where an authenticated user attacks the web applications within its own user authority. An HTTP parameter tampering attack, wherein Alice might attack Elgg for profit (e.g., randomly add users to increase number of friends), is such an example [**?**]. We also assume the security of user passwords, i.e., the users should not disclose their passwords nor use the same password for different websites. Lastly, although user approach is applicable to non-JavaScript based in concept, our discussion here precludes malicious Flash scripts or Java Applets embedded in web pages.

3 UserPath Design & Security Properties

Our solution protects user-owned resources in the web application from post-injection script execution. We combine techniques for protecting different channels exploited by the attacks, compatible with today's web browsers and web applications.

3.1 Challenges & Key Ideas

Protection for sensitive user-owned resource should cover the entire life time of web sessions, starting from user authentication to the teardown of the web session. We explain the challenges in doing so below.

Protecting User Credentials. Malicious scripts can exploit display channels to launch in-application phishing attacks and stealing users' passwords. Note that although the browser's security indicators are perfectly valid, they do not help users to recognize such attacks as they occur within the victim website. To achieve secure authentication, the idea is to allow web browser to render secure login elements on the web applications (Section 3.4). Such elements are special UI controls rendered by the browser, which can be easily verified by the user and cannot be tampered with untrusted JavaScript code. Once users enter their credentials, leaking these credentials to an untrusted environment (a script or server) is not desirable. To address this critical problem, we employ a PAKE protocol that enables web browser to authenticate a user to a web origin without directly exchanging credential information with origin O.

Establishing Notion of Users. After successful authentication, another challenge is to securely establish a notion of users inside a web session. We term this step a *secure delegation* (Section X), in which the browser creates a user sub-authority in origin O. This step constitutes a form of authority delegation on the web. To achieve this goal, the key idea is to conceptually split the web

session into two partitions, one web session running under the authority of the web application origin O, the other one running under a user sub-origin O_{Alice}. UserPath ties all sensitive resources belonging to user Alice under sub-origin O_{Alice}, which represents the explicit notion of Alice's *sub-authority*[3]. Note that code running in O_{Alice} represents the authority of Alice in O, and is more privileged than the origin O's code.

End-to-End Trusted Path. During the live web session, UserPath needs to withstand post-injection script execution that exploit any of the four channels we discuss in Section 2.1. Fully protecting the four channels is challenging with any single mechanism. Instead, we carefully integrate various strong security mechanisms to secure all four channels and build an end-to-end trusted path connecting each secure channel to cover interactions between a user, browser and web server. We rigorously explain the detail of our solution in Section 3.4.

3.2 UserPath Design

The process starts by a user Alice visiting a web page with origin O. Alice interacts with the application with the authority of its web origin O (Figure 2 Step A) which invokes an API to draw a special "credential box" for Alice to enter her password. To initiate authentication process, UserPath leverages the standard authentication mechanism using user name and password. The origin O decides the placement and location of the credential box on the web page and Alice needs nothing more than her usual password for this step. Unlike prevailing password boxes where the input is directly accessible to the web page, the data that is entered by Alice in the credential box will stay in the memory of the browser and is not accessible by application code. Therefore, it prevents attackers' scripts from stealing the password. Subsequently, the browser locates the server's URL address by identifying `url` property of the element's HTML, and carry out PAKE protocol to securely authenticate Alice and the server (Step B). After a successful PAKE authentication, a session key K_s is derived for Alice and the server. Both parties mutually authenticate each other.

To allow users to distinguish a credential input element from any other similar-looking credential requesting elements rendered by application code, the browser displays a rectangle of color M in its chrome area and updates the color M simultaneously in its display region[4]. The user recognizes the original credential elements by a visual check. Therefore, this approach defeats any phishing attempts from malicious scripts.

After authentication is carried out using PAKE protocol, UserPath initiates secure delegation step to establish a user sub-authority O_{Alice}. UserPath creates a UFrame to run Alice's privileged code separated from the rest of the application code within a web origin O. Unlike the temporary origin (e.g., sandboxed `iframe` []) which runs in a distinct privileged environment, UFrame runs

[3] This secure delegation process is akin to executing an `su - alice` command in a UNIX-like system.

[4] The browser dynamically decides a foreground text color in the credential input element that has high contrast with the current background color M.

within the user Alice's authority with a higher privilege than any other part in the web page. As a privileged entity, it has *one-way access* by getting (1) full access to the main page's DOM via special DOM APIs that allows it to create secure UI elements; (2) a direct secure channel to UI states in the browser; and (3) the ability to create a secure resource access HTTP requests to the backend server. By implementing the UFrame, UserPath isolates user-owned data from being accessed by the less-privileged application code running in O's authority. All code that processes user events and the associated program data is privilege-separated in the UFrame.

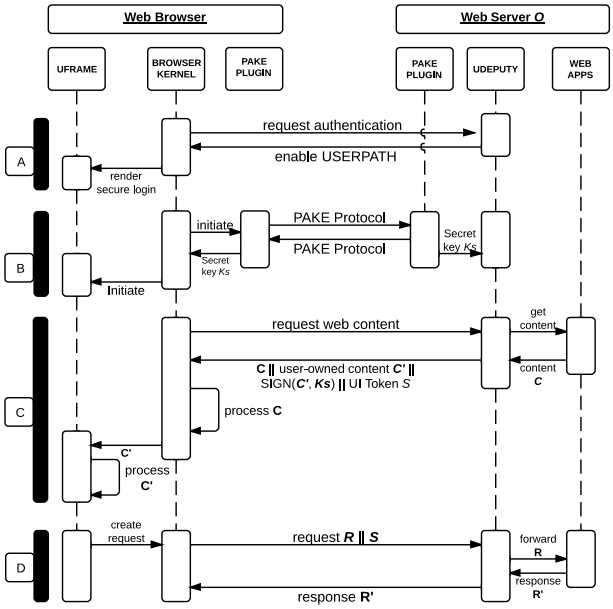

Fig. 2. Sequence of operations in a UserPath-enabled session.

So far, we ensure that no client-side attacks can compromise UFrame-protected code. But, how do we make sure that UFrame code itself is not initialized without any attacker payload when it is fetched from the backend server? To cope with this problem, we largely piggyback user-based isolation of the server on existing mechanisms such as CLAMP [], DIESEL [], and UserFS []. Once the code is received by the UFrame at the initialization step, the browser checks the integrity of the code with respect to K_s.

In order to delegate user-owned resource to the UFrame, backend server passes the code for the UFrame to the browser, which is signed by K_s. The browser needs to check the authenticity of UFrame code because malicious scripts could hijack the `XMLHttpRequest` object and tamper with UFrame code at the initialization step. The browser thus bootstraps the UFrame and provides a ded-

icated `XMLHttpRequest` channel to securely communicate with the UDeputy at the server side. Inside the UFrame code, the UDeputy embeds a set of nonces S called the *user interaction token* (Step C) that can be used in generation of resource access HTTP requests from client side. These tokens can only be attached by browser kernel, are not exposed to web application code, and are explicitly attached to HTTP requests initiated by code in UFrame. Such resource access requests are those that instruct the server to perform privileged operations under a certain user, such as adding a user as an admin in our running example. UFrame attaches user interaction tokens to such requests at the client side to enable web servers in discerning requests sanctioned by authentic user interactions (Step D).

3.3 Security Properties

We aim to ensure that web platform design that is UserPath-compliant will be resilient against PISE attacks. Our solution enforces the following semantics as security properties for UserPath.

- **P0**: **Safe Mutual Authentication & K_s Establishment.** A mutual authentication between the user and the server is required for web servers to securely delegate user Alice's authority O_{Alice} to client-side code within its web origin's authority O. This delegation is bootstrapped by Alice's user name and password. The secure delegation process must ensure that credential information does not leak outside Alice's authority, such as to attacker-controlled domains. After the successful authentication, a session key K_s is derived. The key K_s must remain unforgeable, unguessable, and unique during the sessions.
- **P1**: **Secure Delegation.** UFrame code that is passed from the backend server needs to be signed by K_s that is derived from mutual authentication between user and web server. Once web browser received the content of UFrame, it has to check the authenticity of the code with respect to K_s.
- **P2**: **Post-initialization Security of UFrame.** All sensitive data and code must be kept isolated inside a UFrame. The rest of the application code outside UFrame must not be able to access these data and code whatsoever.

The property P0 and P1 serve as the basis for subsequent security properties P2, P3, and P4. Beyond these two properties, each channel requires additional security properties described as follows.

- **P3**: **Secure Visual and Input Channels for Users**
  Visual channel. We reuse the standard secure visual channel that requires display integrity, intent integrity, spatio-temporal integrity, and pointer integrity to ensure *distinguishability* of secure UI elements from non-secure ones. Secure UI elements cannot be obstructed or tampered with by untrusted code. Its elements should be able to display confidential information to users and are not accessible to non-UFrame code. This has been explored

through a number of research work [?, ?, ?] and this is not part of our contributions.
Input Channel. All keyboard inputs to secure input elements go directly into the browser. The confidentiality and integrity of input action should not be violated by untrusted scripts. The browser should be able to distinguish genuine user interactions from those mimicked by JavaScript code.

- **P4**: **Secure Browser ↔ UFrame Channel.** Privileged UFrame can communicate to the browser directly in order to create secure UI elements or to read contents in DOM objects securely with no possibility of interception from untrusted code. The confidentiality, integrity and authenticity of such communications are maintained by the browser.
- **P5**: **Secure UFrame ↔ Server Channel.** Web server should be able to distinguish requests generated from the authentic user interaction, and those that are not. The communications between the UFrame and the server are protected in their confidentiality and integrity.

Due to space constraints, we give a more thorough example-by-example security analysis in our technical report [?].

3.4 Security Analysis

We explain how the property P0-P5 defeat attacks presented in Section 2.1. Following this section, we detail the implementation of UserPath in Chromium web browser (Section 4).

Attacks targeting credential information. Credential input element and PAKE-based authentication guarantee no sensitive data being leaked to the untrusted web application code. On one hand, code running under authority O has no access to the data as the sensitive data is privileged. On the other hand, any attempts to intercept the transmission of the password will fail, since PAKE protocol enables authentication without actually revealing the whole plaintext of the password to the server. Besides, web browser draws a special credential element and updates the color of the element simultaneously. This mechanism lets the user to distinguish the original credential element from similar-looking elements, thus defeats in-application phishing attacks. Therefore, our solution enforces P0.

Trusted Path through UFrames. After the secure delegation process finishes, the browser creates a UFrame for executing trusted JavaScript code. In this step, the browser already has a shared key K_s that can be used to secure the communication with the server. UDeputy then tags[5] the content of UFrame using the key K_s and sends it to the browser, embedded in a custom HTML tag. Whenever the browser indicates UFrame content during parsing, it checks the integrity of UFrame code, and creates an `iframe` with a random origin

[5] Tag refers to the process of creating an authentication code using a symmetric key. It frequently substitutes for the term MAC.

$O_R = PRG(K_s)$, where $PRG(K_S)$ is a pseudorandom generator function that takes the shared key K_s as the seed. As the sensitive code and data are hosted in distinct privileged environment, the rest of web application cannot access such resources, thus enforcing property P1.

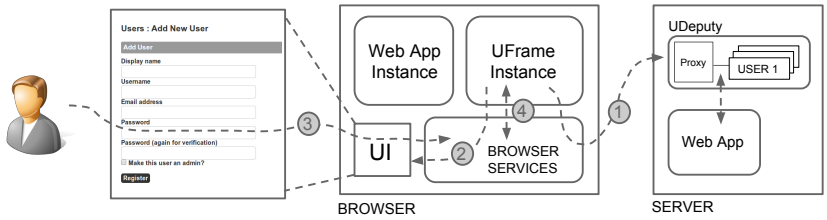

Fig. 3. Secure Channels in UserPath. There are four secure channels that constitute an end-to-end trusted path from the user to the web server. Each secure channel is resilient against scripted attacks generated by malicious scripts.

With the UFrame created at the client side running under a user sub-origin, e.g., O_{Alice}, we augment the present web browser and server with four secure channels to establish an end-to-end trusted path from the user to the server. We illustrate the four secure channels in Figure 3

– ① **UFrame ↔ UDeputy Channel.** This channel secures two-way communication between UFrame and UDeputy. UFrame also uses this channel to pass resource access requests to the server, as well as updating its content.

UFrame → UDeputy Communication. UFrame receives a set of user interaction tokens from the UDeputy, along with the content that is sent from the server. To prevent requests forged by injected scripts from accessing user-owned data, the UDeputy only accepts the request if it carries the expected user interaction token. In particular, for each resource access request, the server sets a unique user interaction token T to an element that can trigger this request (e.g., the "Add user" Button). The UFrame attaches the token T corresponding to the interaction with element to the header of the HTTP request sent through the dedicated `XMLHttpRequest` interface. Thus, the UDeputy in the server-side web application can decide whether the request is triggered by the expected user interaction by verifying the appended token, if any. Note that the token will be attached if and only if the elements triggered by trusted code in UFrame or real human user interaction. The authenticity and confidentiality of user interaction tokens are protected by the secure channel between the UFrame and the UDeputy. Therefore, the secure channel enforces property P4 in Section 3.3.

UDeputy → UFrame Communication. This communication happens when the UDeputy sends the content to the UFrame to update the code or user interaction tokens. The content is sent in JSON format that is tagged using

key K_s by the UDeputy. The content is later parsed and processed by the UFrame.

- ② **Visual Channel.** Using secure channel between UFrame and UDeputy, web servers can update UFrame's trusted code. Such trusted code might need to securely display sensitive information to users. To support such operations and functionality securely, we introduce secure UI elements for display.

  Secure Display Element. The secure display element allows UFrame code to securely render a region of UI content, such as images, styled-texts, and so on, to the user. Such a region is tamper-resistant from the JS code running in the web page, thus establishing a secure visual channel between user and the browser. In consequence, it also enforces visual channel property in P2. This channel ensures critical content, such as the warning messages of suspicious activities with Alice's account, is conveyed to Alice correctly. This element can also be used to render confidential information that is invisible and not accessible to attacker's scripts.
- ③ **Input Channel.** Besides the visual channel, our design provides a secure channel for user input interaction. It ensures that user interactions with secure UI elements are directly received by the browser without passing through the untrusted code. We reuse existing solutions to ensure visual, spatio-temporal, intent, and pointer integrity to prevent clickjacking attacks [**?**, **?**].

  Secure Input Element. To support such functionality, we introduce secure input element as part of secure UI elements. A secure input element renders an input element for the UFrame to receive users' keyboard inputs, or a button to receive users' mouse clicks. The functionality is equivalent to HTML `input` elements. This secure element is rendered by using a special API call available to the UFrame code, as we discuss next. It is not accessible to malicious script running on the web page.

  Security Analysis. In this step, any input into the secure input element goes directly to the browser and is passed onto the UFrame. Any event that invokes JavaScript function call is directed into functions inside the UFrame, and any access to secure element's properties is confined to JavaScript code inside the UFrame. Thus, web page application has no way to access the state of the UI element. This enforces property P1 and P3 in Section 3.3.
- ④ **UFrame ↔ Browser Channel.** The UFrame needs a way to securely access the DOM elements of the web page. To support this operation, the browser provides a one-way isolation mechanism with *upcalls* and *downcalls*. An upcall is an API access that comes from the browser to the UFrame (Browser → UFrame). On the other hand, a downcall is an API access that comes from the UFrame targeted at the browser (UFrame → Browser). For each access type, we introduce new DOM APIs between the UFrame and the browser, as listed in Table 2. The APIs are accessible only for code running on UFrame and the browser kernel. We design the APIs by mirroring on standard DOM API, and we expect the web developers to experience easy learning curve when implementing the APIs.

Table 2. Secure DOM APIs for UFrame

Type	API Name	Description	Type	API Name	Description
D	`createSecElement`	Create secure UI element	U	`storeSecretKey`	Store key K_s that is derived from PAKE protocol
D	`getSecElementById`	Get secure UI element's object based on ID	U	`updateUFrameCont`	Update UFrame code or data content
D	`setSecElmAttr`	Set the property of an object with the corresponding value	U	`createContext`	Create a UFrame context that runs with user privilege. It lets UFrame to access privileged APIs
D	`getSecElmAttrVal`	Get the property's value of an object	U	`removeSecretKey`	Remove secret key K_s during teardown process
D	`deletePAKESesKey`	Delete session key K_s on the browser kernel	U	`removeUIToken`	Remove user interaction token T. This is done in teardown process

D: Downcall API U: Upcall API

– **Teardown.** As the user Alice logs out of O, the UDeputy invalidates the session key K_s, and sets a custom HTTP header `X-USERPATH:Session-destroy` in the HTTP response for the log out request. After getting this response, the browser destroys all user interaction tokens for the session and the session key K_s. To allow session reconnection, similar to cookies, the browser caches the user interaction tokens and K_s until the user logs out. The UDeputy redirects the request to the login page if the key and tokens expire.

3.5 Compatibility & Usability Implications

We discuss the usability implications that may arise from our design. First, we assume that web application users will always check the background color of any credential-seeking elements, and only enter their passwords if the color matches that of a rectangle displayed in the browser's chrome area. Second, we rely on prior research [**?**, **?**, **?**] to ensure the visual, temporal and pointer integrity of a secure visual channel. The usability of such a scheme has not been fully evaluated; a thorough user study on its usability merits separate research, similar to prior work evaluating other security schemes [**?**, **?**].

Compatibility with SSO-based Applications. Our mechanism can easily be extended to handle authentication via Single-Sign On (SSO). If the server O delegates the authentication to an SSO provider S, a separate HTTPS connection is established from the browser to S. Thereafter, the credential input element uses the user name and password to initiate the PAKE authentication with S. Upon successful completion, the browser obtains a shared key K_s with S, which is also communicated by S to O in a separate channel. O can create a UDeputy for Alice using K_s. The browser thus creates a UFrame with the authority of *Alice*@S, which can isolate *Alice*@S from another user.

4 UserPath Implementation

We build an end-to-end implementation of UserPath by modifying the client side browser and server side web application. On the client side, we implement the UFrame and trusted path components by modifying Chromium[6], the open source version of Google Chrome. We patch Chromium version 12 by adding roughly 475 lines of codes spreading over 26 files inside Chromium codebase. The browser modifications are required to implement the UFrame component and secure channels for the trusted path.

In our implementation, we integrate TLS-SRP [**?**] — a PAKE-based web authentication that operates at the transport layer — into our system. In addition, we have installed a browser level TLS-SRP plug-in for performing PAKE protocol between browser and the web server. The plug-in consists of 381 C++ lines of code, which is 2.6 MB in size. On the server side, we implement UDeputy and modify several web application components according to our design. We also apply a patch to the Apache web server for supporting TLS-SRP [**?**].

We have released our patch to Chromium and the modified web applications on our public repository [**?**]. We also release a demo video showing how UserPath integrates with our running example Elgg [**?**].

4.1 User Sub-Origin Creation

To integrate TLS-SRP into Chromium, we develop an NPAPI plug-in for this browser to support client-side TLS-SRP authentication protocol. In deploying our solution to real world applications, developers just need to install the plug-in into the web application's login page. Deploying TLS-SRP to the client-side web application is straightforward with several lines of code to initiate and handle TLS-SRP authentication.

4.2 Trusted Path

We leverage existing mechanisms in the Chromium web browser to establish trusted paths. We modify *isolated worlds* [**?**], a feature provided by Chromium to separate execution context between two JavaScript codes. For ease of implementation, we use isolated worlds to simulate `iframe`-based isolation. However, our solution can also be implemented using `iframe`-based isolation with temporary origin (see Section 3.4 for detail). By utilizing isolated worlds, we let the user run a web application code and a piece of JavaScript code under two separate contexts, isolated each other. In this section, we further call the execution context of web application code as *main world* context.

In our implementation, we make a small number of changes in the classes of `ScriptController`, `V8IsolatedContext` and `V8NodeCustom`. Essentially, we separate UFrame's code from the main world's code. Then, we run UFrame code under a separate JavaScript execution context using isolated worlds. We

[6] http://www.chromium.org/

modify isolated worlds by adding a data structure called `IsolatedContextMap` that maintains the relation between code running on the web page with its context, represented by context identifier. Therefore, the system could recognize the context where a JavaScript code running by checking the data structure. Finally, we modify Chromium to mediate access from a JavaScript object to a DOM Node. In this mediation, only a JavaScript code that is running on a UFrame context should have access to all DOM objects on the web page. On the contrary, the code that is running under main world's context should not be able to access any DOM object that is created under a UFrame context. This way, we preserve one way isolation of UFrame code.

We use our running example in Section 2 to illustrate how we implement trusted path execution. For example, we label contact information as a sensitive element to prevent them from being leaked to malicious code running on a web page. In Listing 1.1 line 6, a secure DOM element is created by invoking a downcall API `createSecElement()`. This API receives a JSON object `jsonData` as an input, and creates a secure UI element based on information given by `jsonData`. The object `jsonData` is user-owned contact information, which is a content set by UDeputy.

```
1  //-- Add secure element
2  // phone number content obtained from UDeputy
3  var jsonData = {"type":"display","elm":"div","id","phone-info
       ","value":"88880000","parentNode":"div-container"};
4
5  // create secure div element to display phone number and
       append it into existing DOM element
6  var phoneElm = createSecElement(jsonData);
7
8  ...
9
10 //-- Add a user as admin
11 var xhr = new XMLHttpRequest();
12 xhr.open('POST', 'http://'+URL+'/elgg/action/useradd', true);
13 xhr.setRequestHeader('Content-type', 'application/x-www-form-
       urlencoded');
14
15 var username = getSecElementById('username').getSecElmAttrVal
       ('value');
16 var data = "username="+username;
17 ...
18 // secure resource access to Elgg server
19 xhr.send(data);
```

Listing 1.1. Trusted Code Running in a UFrame. This piece of code executes under the user's authority O_{Alice} to create a secure `div` element into the web page and secure HTTP request to add a user as an admin.

Besides adding secure UI elements to the web page, we also need to protect client-side request to Elgg's server. We modify form submission process

to create an admin in Elgg to be generated only from the UFrame code. In Listing 1.1 line 10-20, we create a POST request directly from UFrame using `XMLHttpRequest`. The data that is sent through POST request (e.g., username, password) is obtained from user input on secure input elements. The UFrame code can access those input elements by invoking `getSecElementById()`, and `getSecElmAttrVal()` to get the attribute value of those secure objects (Listing 1.1 line 16). As `XMLHttpRequest` object is being called from the UFrame, the browser treats the request as secure resource access to the server.

5 Evaluation

We deploy UserPath on 20 open source web applications (as Table 4 shows) from 8 different categories (as Table 5 presents) including 3 frameworks (WordPress, Joomla, and Drupal). These web applications are statistically popular, built using PHP, and cover a wide range of functionalities, shown in Figure 4. We evaluate our solution from the following four aspects.

- **Scope of Vulnerabilities**. We analyze historical vulnerabilities in web applications that can potentially lead to post-injection script execution. UserPath eliminates these 325 vulnerabilities from threatening the security of user-owned resources.
- **Applicability to Web Applications**. We evaluate UserPath on 20 open-source web applications from 8 different categories, ranging from social networking to customer relationship management. We show that the abstractions and primitives proposed in UserPath can be applied to these applications to protect typical sensitive fields owned by users in the applications.
- **Adoption Effort & TCB Reduction**. We demonstrate UserPath can be integrated with the web applications with a small adoption effort (1-2 days). In addition, UserPath significantly reduces the size of TCB that has access to use-owned resources in these applications from 8x to 264x.
- **Performance**. We measure performance overhead incurred by UserPath, which is around 1% - 3% in our experiment.

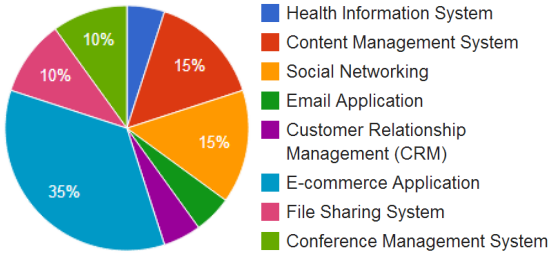

Fig. 4. Categories of Web Applications Studied in UserPath Evaluation

5.1 Scope of Vulnerabilities

We study the set of vulnerabilities in the web applications that could potentially leads to post-injection script execution. Among the 20 open source web applications listed above, there are 325 vulnerabilities on those web applications that can be exploited to launch post-injection script execution. Most of them have been patched and recorded in the vulnerability database, but some of them are still unpatched.

Table 3 shows the summary of the web applications, including its popularity, their sensitive user data, and the number and representative samples of vulnerabilities that can result in post-injection script execution. Among those 20 web applications that we study, all of them have at least one vulnerability to a subset of post-injection script execution, namely XSS or CSRF attacks. Some of them even have more than ten vulnerabilities of the same attack vector. To name one of them, PrestaShop has two critical vulnerabilities. One type of vulnerability (marked by ID CVE-2008-6503) allows an attacker to inject arbitrary web script or HTML to login page. The other vulnerability, marked by ID CVE-2011-4544 lets the attacker to exploit file management process of an administrator to launch XSS attack. This confirms our hypothesis that existing applications are vulnerable to post-injection script execution, thus motivating the need for second line of defense solutions.

5.2 Case Study : Elgg Social Networking Engine

We first show a detailed case study to illustrate the steps in enabling UserPath in Elgg. In this application, we modify three features that manage sensitive user-owned resource: (1) user's profile data management, (2) the "add new user" feature, which is a privileged feature that can only be accessed by administrator, and (3) "user settings" in administrator portal to set a particular user as an administrator. We mainly focus on the "add new user" feature in our description.
Code Changes. The first thing to do is by creating a Credential Input Element, in which an admin could use to perform TLS-SRP-based authentication. We let the browser renders credential element by making a small changes in file `actions/login.php`. A piece of trusted JavaScript code is passed from the server to the browser subsequently after the administrator logs in to the system. It contains a set of instructions to create secure elements to support this feature.

The feature "add new user" is a functionality that accessible only to user with administrator privilege. In "add new user" page, we label display name, username, email address, password, admin flag, and a button to initiate form request as sensitive elements. Essentially, those elements need to be dynamically created from within a UFrame code to let them rendered as secure elements. We do so by modifying a PHP file `forms/useradd.php` to handle this operation. In total, there are six input elements and one button element created by the UFrame code to add new user. Finally, we add a click event listener to the button in order to launch POST requests. In the event listener function, We invoke the `XMLHttpRequest` API from the UFrame code. As the request is generated from

within the UFrame, the browser treats this request as secure resource access to the Elgg server, and appends to it the user interaction token accordingly.

In the first feature, we locate 16 sensitive UI elements and label them as sensitive. In total, there are four files that we modify: `useradd.php`, `details.php`, `user.php`, and `makeadmin.php`. Besides, we redesign authentication method to enable TLS-SRP based authentication. Inside a UFrame code that we embed in a web page, we dynamically generate secure UI elements for 16 labelled sensitive elements. As for "add new user" feature, we take a different approach by modifying HTML Form-based user-data submission in `useradd.php` into a dynamic `XMLHttpRequest` that is triggered from the UFrame. Therefore, we enforce secure resource access to Elgg's server for this particular HTTP request.

Result & Challenges. We successfully retrofit UserPath to Elgg by adding 270 lines of JavaScript and PHP code in its application code. The TCB size of the UFrame in the modified Elgg is 46x smaller than the size of TCB in vanilla Elgg. Specifically for "add new user" feature, we successfully protect the labeled elements from script injection attacks. We have tested the secure elements by running a script to obtain these sensitive objects. UserPath successfully eliminates the threats by returning `null` each time the untrusted script tries to access each object. Besides, we launch another script to forge a fake request to add a user, and monitor the HTTP request using Chromium's network monitor. As a result, our modified browser embeds user interaction token only to request coming from the UFrame, which is as we might expect.

The main challenge of using UserPath to retrofit Elgg is the difficulty in locating the functionality we need to modify, because Elgg is built using their own toolkit. After understanding the toolkit, the modification effort is straightforward.

5.3 Applicability to Web Applications

We successfully retrofit all 20 web applications to adopt UserPath. Among these applications, we locate several data and operations that are sensitive to users and modify the PHP file where these data and operations are processed. Generally, there is no rule to choose the correct data or operations to be protected by UserPath. However, data that is frequently targeted by the attacker to be stolen may be a good indicator to be marked as sensitive. We manually choose several pieces of sensitive data and operations for each web applications, and summarize them in Table 3.

5.4 Adoption Effort & TCB Reduction

We demonstrate practicality of UserPath by summarizing adoption effort and TCB reduction of 20 retrofitted web applications in Table 4. We measure the adoption effort by the following benchmarks: number of additional code, number of modified files, and number of days spent in modifying web application. Besides, we also measure TCB reduction by comparing initial TCB size (i.e., the web page size) and final TCB size after implementing UserPath.

We find that UserPath requires small changes to the existing web application code. Given the set of sensitive user-owned data and functionality that we want to protect from post-injection script execution, we only need to add at most 270 lines of PHP and JavaScript code into the web application, with 167 lines of code added for each web application on the average. Moreover, we empirically show that we achieve reduction of 8x to 264x in TCB for our case studies. We measure this reduction by comparing the size of final TCB (e.g., the UFrame code) with the entire web page size (see column IV in Table 4). We treat the web page size as initial TCB size as we need to trust the entire web page in order to protect our sensitive data and operation.

We also find that modifying web application according to UserPath incurs relatively small burden on the developer side. On the average, given a set of sensitive user-owned resource to protect in Table 3, a developer only needs to modify 6 files within 1.3 days for one web application to make it comply with UserPath.

5.5 Performance

The main performance impacts our solution brings to the web platform includes: PAKE-based secure delegation, the UFrame creation, and new secure elements introduced into DOM. As our demo video [**?**] shows, in our experiments with the 20 web applications, we do not observe any slowdown in user interactions with the applications. Since the login phase contains all the three factors, we measure the overhead of the login time for 8 applications from 8 different categories. Table 5 summarizes the results of the login time (averaged on 5 runs) between the click on the Login button and the next page finishes loading . We can see that our solution UserPath introduces negligible performance overhead to these applications. This confirms our speculation that the minimal performance overhead that might incur from UserPath would be largely masked by the timing variances in network requests.

6 Related Work

In this section, we discuss recent research works that are related to our solution.
Privilege Separation Privilege separation is a fundamental security primitive that reduces the potential damages of compromised software components by partitioning software into different compartments. It has been widely adopted in traditional applications [**?**, **?**], web browsers [**?**, **?**, **?**, **?**], and web applications [**?**, **?**, **?**, **?**]. View isolation implemented by PathCutter [**?**] separates code running in different `iframe`s (views) as well as requests coming out of different views. It thus prevents unwanted access to data between views, either directly or indirectly via sending requests to the server.

However, one key question in applying privilege separation is how to partition data and code and control sharing between partitions, which is difficult

to determine in today's highly dynamic web applications. For example, web applications commonly include additional external scripts at run time, which may break pre-determined partitions. Instead, our solution in this paper takes a user-centric approach to bring in user sub-origins to the present web, and confine user data only to code delegated by the user sub-origin.

Data Confinement Confining data in web applications has recently received attention in the research community. For instance, Roesner et al. propose user-driven access control gadgets, which allow users to directly grant access to user-owned resources by their UI interactions with such gadgets [**?**]. Our solution shares the similar insight as to confine user data back to user-sanctioned operations, although we face different challenges in protecting user data on the web. Unlike resource on operating systems, the distributed nature of the web and decoupled server-client architecture requires additional secure channels to confine user data on the web. We address such challenges by integrating TLS-SRP into web authentication to build an end-to-end trusted path from the client-side application code to the web server.

Several other works have been proposed to confine sensitive data on the web [**?**] or cloud platform [**?**]. Compared to these proposals, our solution does not confine user data according to any application-specific configuration or data propagation policies; instead, it ensures that user data only flows within user sub-origin, both at the client and the server side.

Trusted Paths Building trusted paths across untrusted software components [**?**] has practical significance today. Trusted paths enable software components to safeguard sensitive data or system resources in the presence of security vulnerabilities that are prevalent in today's systems and applications [**?**]. Various prior works examine potential solutions for trusted paths between user-interaction elements and software applications [**?**, **?**, **?**, **?**]. Similarly, Web Wallet redesigns browser's user interfaces to protect user credentials against phishing attacks [**?**]. As an increasing concern, the usability of trusted path proposals has been evaluated in real-world usage [**?**, **?**]. Zhou et al. propose a hypervisor-based general-purpose trusted path design on commodity x86 computers, and present a case study on a user-oriented trusted path [**?**]. Under a different scenario, our solution builds an end-to-end trusted path by leveraging and augmenting the existing web browser and server. This trusted path connects the user at the client side to a trusted component (UDeputy) at the server side, ensuring only user-delegated sub-origins can access protected data.

Such a trusted path differs from a recent proposal on a trusted path between user keyboard inputs and the web server, where no explicit notion of users is established [**?**]. Moreover, compared to it, our solution requires much smaller changes to web browsers; by piggybacking on passwords for authentication, we avoid the usability challenges in requiring users to generate and upload SSL keys as in [**?**].

Dong et al. propose a solution to identify requests crafted by injected scripts from requests triggered by user interactions [**?**]. We apply a similar mechanism in our solution as part of input channel protection. However, their work focuses

on monitoring and diagnosing web application behaviors, and does not yield a solution for protecting data in web applications.

Injection Attack Prevention As we discuss in this paper, injected scripts pose major threats to web applications. Previous endeavors of security researchers have devised numerous solutions to prevent or mitigate script injection [**?**,**?**,**?**,**?**]. Nevertheless, in practice, it is difficult to eliminate all script injection vectors [**?**]. Our solution as a second line of defense, can complement these solutions on script injection prevention, and improve the overall security for web applications.

Clickjacking Defenses Various clickjacking defenses focus on preserving the integrity of user interfaces [**?**, **?**, **?**]. On the other hand, post-injection script execution can potentially exploit all visual, temporal or perceptual interpretation of the application's user interfaces to redirect the user's action to unintended elements. Such attacks go beyond the scope of existing clickjacking defenses, and require more comprehensive solutions to tightly bind privileged data access to legitimate users. We thus design our solution in this paper, and it can be deployed in parallel to existing clickjacking defenses to prevent a broader category of attacks.

7 Conclusion

Post-injection script execution are a major threat to user-owned resources in web applications. The present web lacks the explicit notion of user sub-origins, while providing any scripts injected into web sessions unfettered access to resources owned by the victim user. In this paper, we propose new abstractions to bring in the explicit notion of user sub-origins into the present web. Our solution establishes an end-to-end trusted path between the user and the web server, which allows the server to confine resources belonging to one user and code delegated with the user's authority. We show that our solution eliminates a large amount of post-injection script execution in real-world applications, and can be integrated with today's web browsers and applications with minimal adoption cost.

References

1. W3C: Content security policy 1.0. `http://www.w3.org/TR/CSP/`
2. Johns, M.: Preparedjs: Secure script-templates for javascript. In: Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA '13 (2013)
3. Akhawe, D., Li, F., He, W., Saxena, P., Song, D.: Data-confined html5 applications. In: Proceedings of the 18th European Conference on Research in Computer Security. ESORICS '13 (2013)
4. Dong, X., Chen, Z., Siadati, H., Tople, S., Saxena, P., Liang, Z.: Protecting sensitive web content from client-side vulnerabilities with cryptons. In: Proceedings of the 20th ACM Conference on Computer and Communications Security. CCS '13 (2013)

5. Parno, B., McCune, J.M., Wendlandt, D., Andersen, D.G., Perrig, A.: Clamp: Practical prevention of large-scale data leaks. In: Proceedings of the 2009 IEEE Symposium on Security and Privacy. (2009)
6. Felt, A.P., Finifter, M., Weinberger, J., Wagner, D.: Diesel: applying privilege separation to database access. In: Proceedings of the 6th ACM Symposium on Information, Computer and C ommunications Security. ASIACCS '11 (2011)
7. Chen, E.Y., Gorbaty, S., Singhal, A., Jackson, C.: Self-exfiltration: The dangers of browser-enforced information flow control. In: Proceedings of the Workshop of Web 2.0 Security & Privacy 2012. (2012)
8. Dong, X., Patil, K., Mao, J., Liang, Z.: A comprehensive client-side behavior model for diagnosing attacks in ajax applications. In: Proceedings of the 18th International Conference on Engineering of Complex Computer Systems. ICECCS '13 (2013)
9. Projects, T.C.: Per-page suborigins. `http://www.chromium.org/developers/design-documents/per-page-suborigins`
10. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C.: User-driven access control: Rethinking permission granting in modern operating systems. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. (2012)
11. Engler, J., Karlof, C., Shi, E., Song, D.: Is it too late for pake? In: Proceedings of Web 2.0 Security and Privacy Workshop 2009
12. Anonymous: Url for patches to chromium and web apps. `https://github.com/anonymous-repo/userpath`
13. Group, E.: Facebook profiles can be hijacked by chrome extensions malware. `http://underurhat.com/hacking/facebook-profiles-can-be-hijacked-by-chrome-extensions-malware`
14. Roesner, F., Fogarty, J., Kohno, T.: User interface toolkit mechanisms for securing interface elements. In: Proceedings of the 25th annual ACM symposium on User interface software and technology. UIST '12 (2012)
15. YGN Ethical Hacker Group: Elgg 1.7.9 xss vulnerability. `http://yehg.net/lab/pr0js/advisories/%5Belgg_179%5D_cross_site_scripting/`
16. CVE: Cve-2012-6561 xss vulnerability in elgg. `http://www.cvedetails.com/cve/CVE-2012-6561/`
17. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Proceedings of the 15th ACM conference on Computer and communications security. CCS '08, New York, NY, USA, ACM (2008) 75–88
18. Akhawe, D., Saxena, P., Song, D.: Privilege separation in html5 applications. In: Proceedings of the 21st USENIX Security Symposium. (2012)
19. Wu, M., Miller, R.C., Little, G.: Web wallet: Preventing phishing attacks by revealing user intentions. In: Proceedings of the 2006 Symposium On Usable Privacy and Security. (2006)
20. Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Language-based defenses against untrusted browser origins. In: Proceedings of the 22nd USENIX Security Symposium. (2013)
21. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web application. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. (2010)
22. Huang, L.S., Moshchuk, A., Wang, H.J., Schechter, S., Jackson, C.: Clickjacking: attacks and defenses. In: Proceedings of the 21st USENIX Security Symposium. (2012)

23. Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. ASIACCS '10 (2010)
24. Cao, Y., Yegneswaran, V., Porras, P., Chen, Y.: Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium. NDSS '12 (2012)
25. Zhou, Y., Evans, D.: Protecting private web content from embedded scripts. In: Proceedings of the 16th European Conference on Research in Computer Security. ESORICS '11 (2011)
26. Dong, X., Tran, M., Liang, Z., Jiang, X.: Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In: Proceedings of the 27th Annual Computer Security Applications Conference. ACSAC '11 (2011)
27. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium. CSF '10 (2010)
28. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N.: Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS '10 (2010)
29. Ye, Z.E., Smith, S.: Trusted paths for browsers. In: Proceedings of the 11th USENIX Security Symposium. (2002)
30. Libonati, A., McCune, J.M., Reiter, M.K.: Usability testing a malware-resistant input mechanism. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium. NDSS '11 (2011)
31. Slack, Q.: Tls-srp in apache mod_ssl. `http://sqs.me/security/tls-srp-in-apache-mod_ssl.html`
32. Anonymous: Url for UserPath demo video. `https://github.com/anonymous-repo/userpath/wiki/Demo-Video-URL`
33. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS). NDSS '10 (2010)
34. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: Proceedings of the 12th USENIX Security Symposium. (2003)
35. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: Proceedings of the 13th USENIX Security Symposium. (2004)
36. Grier, C., Tang, S., King, S.: Designing and implementing the op and op2 web browsers. ACM Transactions on the Web (2011)
37. Tang, S., Mai, H., King, S.T.: Trust and protection in the illinois browser operating system. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. OSDI '10 (2010)
38. Wang, H.J., Grier, C., Moshchuk, A., King, S.T., Choudhury, P., Venter, H.: The multi-principal os construction of the gazelle web browser. In: Proceedings of the 18th USENIX Security Symposium. (2009)
39. Barth, A., Jackson, C., Reis, C., Team, T.G.C.: The security architecture of the chromium browser. `http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf`
40. Louw, M.T., Ganesh, K.T., Venkatakrishnan, V.N.: Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In: Proceedings of the 19th USENIX Security Symposium. (2010)

41. Ingram, L., Walfish, M.: Treehouse: Javascript sandboxes to help web developers help themselves. In: Proceedings of the 2012 USENIX conference on Annual Technical Conference. USENIX ATC'12 (2012)
42. Papagiannis, I., Pietzuch, P.: Cloudfilter: practical control of sensitive data propagation to the cloud. In: Proceedings of the 2012 ACM Workshop on Cloud Computing Security. CCSW '12 (2012)
43. of Defense Standard, D.: Department of defense trusted computer system evaluation criteria, 5200.28-std (1985)
44. Symantec: Vulnerability trends, internet security threat report. `http://www.symantec.com/threatreport/topic.jsp?id=vulnerability_trends&aid=vulnerability_trends_intro`
45. Tong, T., Evans, D.: Guardroid: A trusted path for password entry. In: Proceedings of the 2013 Workshop on Mobile Security Technologies. MoST '13 (2013)
46. McCune, J.M., Perrig, A., Reiter, M.K.: Safe passage for passwords and other sensitive data. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium. NDSS '09 (2009)
47. Cheng, Y., Ding, X.: Virtualization based password protection against malware in untrusted operating systems. In: Proceedings of the 5th International Conference on Trust and Trustworthy Computing. TRUST '12 (2012)
48. Zhou, Z., Gligor, V.D., Newsome, J., McCune, J.M.: Building verifiable trusted path on commodity x86 computers. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. (2012)
49. Ter Louw, M., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: Proceedings of the 2009 IEEE Symposium on Security and Privacy. (2009)
50. Gundy, M.V., Chen, H.: Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: Proceedings of the 16th Annual Network & Distributed System Security Symposium. NDSS '09 (2009)
51. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium. NDSS '09 (2009)
52. Roesner, F., Kohno, T.: Securing embedded user interfaces: Android and beyond. In: Proceedings of the 22nd USENIX Security Symposium. (2013)

Table 3. List of Vulnerabilities that Can Lead to Post-Injection Script Execution in 20 Open-Source Web Applications

App Name & Version	Popularity	Sensitive User Data	# of Relevant Vulnerabilities
Elgg v1.8.16	>2,800,000 downloads	Private profile data and admin options (set user as admin and add new user)	**3** (CVE-2012-6561: XSS, EDB-ID 17685 & 8993: XSS)
Friendica v3.2.1744	Forbes's Top 3 social network application	Private contact, friend list, and message data	**1** (Bug ID 0000535: Reflected XSS)
Roundcube v0.9.4	>2,400,000 downloads	Address book, settings and private emails	**12** (CVE-2013-5646: XSS & CVE-2009-4077: CSRF)
OpenEMR v4.1.2	Serving >30,000,000 patients	Personal information, medical records, and payment information	**2** (ZSL-2013-5129 & 103810: XSS)
ownCloud v5.0.13	>350,000 users	Contacts, export files and user share options	**15** (CVE-2013-1942: XSS & CVE-2012-4753: CSRF)
HotCRP v2.61	Used by USENIX, SIGCOMM, etc.	Contact information, review and privilege settings	**3** (Bug ID 3f143d2: XSS)
OpenConf v5.30	Used by ACSAC, IEEE, W3C, ACM, etc.	Contact information, review, edit submission and role setting	**1** (CVE-2005-0407: XSS & CVE-2012-1002: XSS)
PrestaShop v1.5.6.0	Powering >150,000 online stores	Personal infomation, credit slips, addresses and checkout information	**2** (CVE-2008-6503 & CVE-2011-4544: XSS)
OpenMRS	Has been used in at least 123 developing coutries	Medical info, user's ID number, role setting	TRUNK-3935 & TRUNK-3937: Stored XSS
OpenCart v1.5.6	>250,000 downloads	Account, address book and checkout information	**1** (CVE-2010-1610: CSRF)
AstroSpaces v1.1.1	DZineBlog's Top 10 open social network platform	Profile information, private message and admin settings	**1** (Bug ID 001: XSS)
Magento v1.8.0.0	Used by >200,000 business	Account information, address information and checkout information	**1** (CVE-2009-0541: XSS)
Zen Cart v1.5.1	>3,000,000 downloads	Account, profile and checkout information	**4** (CVE-2011-4567 & CVE-2012-1413: XSS)
osCommerce v2.3.3.4	>12,000 registered sites with >270,000 members	Account, profile and checkout information	**10** (CVE-2012-1792 & CVE-2012-2935: XSS)
StoreSprite v7.24.4.13	With 14 payment gateways	Account, profile and checkout information	**1** (CVE-2012-5798: XSS)
CubeCart v5.2.4	Powering thousands of online stores	Account, profile and checkout information	**1** (CVE-2008-1550: XSS)
WordPress v3.6	Used by >60,000,000 websites	Account, contact and setting information	**91** (CVE-2013-5738: XSS & CVE-2013-2205: XSS)
Joomla v3.2.0	>35,000,000 downloads	Account, contact and setting information	**45** (CVE-2013-3059 & CVE-2013-3267: XSS)
Drupal v7.23	>1,000,000 downloads	Account, contact and setting information	**126** (CVE-2012-0826: CSRF & CVE-2012-2339: XSS)
Piwigo v2.5.3	Translated into 50 languages	User's management, permission, sensitive profile	**4** (CVE-2013-1468: CSRF & CVE-2012-2209: XSS)
X2CRM v3.5.6	>4,500 installations across 135 countries	Account, opportunity management and contact information	**1** (CVE-2013-5693: XSS)
Summary	-	-	**325** for all 20 applications

Table 4. Adoption Effort and TCB Reduction after Implementing UserPath in 20 Open-Source Web Applications

App Name	# of LOC	Original TCB (KB)	TCB after implementing UserPath (KB)	TCB Reduction Factor	# of Modified Files	# of Days Spent
Elgg	270	414.6	9.1	46x	4	2
Friendica	176	1053.8	5.3	199x	13	1
Roundcube	96	946.0	8.0	118x	4	2
OpenEMR	141	53.6	6.6	8x	7	1.5
ownCloud	106	555.2	2.9	191x	4	1.5
HotCRP	139	184.5	4.6	40x	5	1
OpenConf	151	55.9	2.4	23x	5	1
PrestaShop	111	580.3	5.8	100x	5	1
OpenCart	266	754.8	11.5	66x	6	2
AstroSpaces	119	67.3	3.5	19x	5	1
Magento	227	987.0	11.2	88x	4	1.5
Zencart	130	241.8	6.5	37x	6	1
osCommerce	122	425.8	5.9	72x	5	1
StoreSprite	133	513.8	4.6	112x	4	1
CubeCart	118	469.2	6.2	76x	5	1
WordPress	102	308.7	3.9	79x	4	1
Joomla	87	819.3	3.1	264x	3	1
Drupal	72	199.6	2.6	77x	3	1.5
Piwigo	216	673.5	7.8	86x	6	1
X2CRM	217	1380.4	6.1	226x	10	2

Table 5. Time Taken for Login without & with UserPath (in seconds).

Category	Application Name	Time without UserPath	Time with UserPath	Overhead
Social Networking	Elgg	3.38	3.45	2.07%
Email Application	Roundcube	7.28	7.49	2.88%
Health Information System	OpenEMR	3.238	3.338	3.09%
Conference Management System	HotCRP	1.037	1.065	2.70%
E-commerce Application	OpenCart	4.26	4.40	3.29%
Content Management System	WordPress	3.708	3.777	1.86%
File Sharing System	Piwigo	1.558	1.575	1.09%
Customer Management System	X2CRM	9.105	9.364	2.84%