Michael Freyberger*, Warren He, Devdatta Akhawe, Michelle L. Mazurek, and Prateek Mittal

Cracking ShadowCrypt: Exploring the Limitations of Secure I/O Systems in Internet Browsers

Abstract: An important line of privacy research is investigating the design of systems for secure input and output (I/O) within Internet browsers. These systems would allow for users' information to be encrypted and decrypted by the browser, and the specific web applications will *only have access to the users' information in encrypted form*. The state-of-the-art approach for a secure I/O system within Internet browsers is a system called ShadowCrypt created by UC Berkeley researchers [23]. This paper will explore the limitations of ShadowCrypt in order to provide a foundation for the general principles that must be followed when designing a secure I/O system within Internet browsers. First, we developed a comprehensive UI attack that cannot be mitigated with popular UI defenses, and tested the efficacy of the attack through a user study administered on Amazon Mechanical Turk. Only 1 of the 59 participants who were under attack successfully noticed the UI attack, which validates the stealthiness of the attack. Second, we present multiple attack vectors against ShadowCrypt that do not rely upon UI deception. These attack vectors expose the privacy weaknesses of Shadow DOM — the key browser primitive leveraged by ShadowCrypt. Finally, we present a sketch of potential countermeasures that can enable the design of future secure I/O systems within Internet browsers.

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

***Corresponding Author: Michael Freyberger:** Princeton University, E-mail: mdf3@alumni.princeton.edu
Warren He: UC Berkeley, E-mail: _w@eecs.berkeley.edu
Devdatta Akhawe: Dropbox, E-mail: dev.akhawe@gmail.com
Michelle L. Mazurek: University of Maryland, E-mail: mmazurek@cs.umd.edu
Prateek Mittal: Princeton University, E-mail: pmittal@princeton.edu

1 Introduction

Privacy Concerns in Web Applications. Smartphone and web applications have transformed peoples' lives. In just over a decade, Facebook now has 1.23 billion daily active users [1]. Gmail, YouTube, and Google Maps similarly all have more than 1 billion daily active users [27]. Even smaller companies, like Wunderlist, which is a to-do list manager, has over 13 million users who have created more than 1 billion to-dos [34]. While all of these applications are incredibly useful, they all rely upon the user trusting them with sensitive data.

We must trust Google will not leak our sensitive information regarding our meetings, tasks, and emails. Recently, Google was found to be collecting users' mobile device locations even when the users opted out of location services [11]. We must trust Facebook will not allow employees to spy on our conversations with our friends. However, just a few years ago, a bug at Facebook led to a leak of 6+ million users' data [22]. This is just one of the many headlines regarding information leaks.

Some popular applications require access to users' sensitive information. For instance, Google Maps needs to know the user's home address in order to navigate him home. However, many applications do not actually need to know users' sensitive information in order for their product to work. Wunderlist does not need to know any of the contents of the 1+ billion to-dos on their platform. All of this content can be encrypted, and ought to be encrypted. However, in many cases, the business incentives of corporations may not be aligned with customers' expectation of privacy. Users need to be in control of their information in order for them to be confident their data is secure.

Browser Based Secure I/O. One important step toward putting users back in control of their information is creating a framework that allows for users' information to be encrypted before being accessed by the web application. This framework would allow a user to enter text on any web application, and the untrusted web application would only be able to access an encrypted ver-

sion of the user's information. The user would have exclusive access to his plaintext information. This framework requires Internet browsers to support secure input and output (I/O). The state-of-the-art approach for a secure I/O system within Internet browsers is a system called ShadowCrypt created by UC Berkeley researchers [23]. ShadowCrypt transforms all input fields into secure input fields. Secure input fields are modified in two important ways. First of all, ShadowCrypt adds a lock icon, a border color, and a passphrase to all secure input fields, which signal to the user that the input has been made secure. Secondly, ShadowCrypt modifies the HTML and JS of the page so that the web application can only access the users' information in an encrypted format.

1.1 Contributions

We first demonstrate practical attacks against ShadowCrypt that bypass its security features. We validated our attacks using a real-world implementation of ShadowCrypt and via a user study administered on Mechanical Turk. Finally, we present a sketch of potential countermeasures against our attacks. By using ShadowCrypt as a case study, our work highlights important guiding principles for designing secure I/O within Internet browsers.

User Interface Attack. Addressing user interface (UI) attacks — attacks that rely upon deceiving the user through a manipulation of UI elements — is an important part of the threat model for a secure I/O system. We demonstrate, through a combination of attacks, that ShadowCrypt is vulnerable to a serious user interface attack. ShadowCrypt provides in-content security indicators for all *secure input nodes*, which are input nodes that do not reveal the user's information to the web application. All secure input nodes have a lock placed at the end of the input, a user specified border color, and a user specified passphrase. Our attack mimics these in-content security indicators in order to trick the user into thinking an insecure input node is secure. Our user interface attack is significant because it cannot be mitigated using existing UI defenses. In Section 7, we discuss why InContext [24] does not prevent our attack. The detailed steps of the UI attack provide insight into how a secure I/O system can be designed such that it is robust against UI attacks. The attack is comprised of the following aspects:

- **Render an unencrypted input field** onto the web page by bypassing the ShadowCrypt scheme of converting all input fields into secure input fields.
- **Position the unencrypted input field** directly on top of a secure input field so that the unencrypted input field looks and feels secure based on the positioning of the lock and the border color.
- **Guess the user's selected border color** used by ShadowCrypt when the input is focused and the border has become thicker.
- **Prevent the user from detecting** that his information was intercepted by triggering ShadowCrypt to encrypt the plaintext after it has been read by the attacker.

The crux of our user interface attack is that it tricks users into interacting with an insecure element by redrawing the insecure element as a secure element. This is conceptually different than typical UI attacks, such as clickjacking; rather than trying to lure the user into interacting with a sensitive element without them realizing it (as previous attacks do), our attack is making the user interact with a non-sensitive element, hoping the user believes the element is sensitive.

Attack Evaluation. We tested the UI attack through a user study administered on Amazon Mechanical Turk. Of the 105 users who completed the study successfully, 46 users were not under attack and 59 were under attack. Of the 59 users who were under attack, only 1 user identified the attack. This user study validates the stealthiness of the user interface attack. We discuss the design of the user study and our results in Section 5 and 6 respectively.

Shadow DOM Analysis. In addition to the user interface attack, this paper presents a deep study of Shadow DOM, the browser primitive that empowers ShadowCrypt. We present additional attacks directly against ShadowCrypt's use of Shadow DOM. These attacks allow us to directly access the plaintext information via client side JavaScript code, which is the most important attack vector to prevent in a secure I/O system in the browser. Therefore, these attacks demonstrate the privacy guarantees of Shadow DOM, which ShadowCrypt relies upon, are insufficient.

Countermeasures Sketch. In addition to demonstrating the various types of attacks and validating their effectiveness, we discuss a sketch of potential countermeasures against these attacks. An important takeaway from our analysis is that secure I/O systems must use browser primitives that are explicitly designed for security and privacy. Our proposal is to put users' text into

a separate *origin*, which would be isolated via the same origin policy. We present how `iframes` can be used to host all of the users' plaintext information, rather than Shadow DOM elements. The same origin policy that is enforced by modern web browsers will allow for all of the user events that take place and the data that is stored within the `iframe` to never leak information to the parent web application. Additionally, our countermeasure analysis presents how systems can be developed such that they are robust against UI attacks. The most important aspect of the UI defense is that it must be able to analyze the web page and alert the user of a UI attack, rather than simply rely upon in-page UI indicators.

Impact and Significance. Our work highlights significant UI vulnerabilities in ShadowCrypt. Even though ShadowCrypt has not witnessed widespread deployment, we believe it is critical to understand the security of such mechanisms, which could influence the design of ongoing research systems [7, 13, 17, 19, 28–30, 36] and future technologies. Our work demonstrates that it is important for any system that aims to secure browser-based I/O (even beyond ShadowCrypt) to explicitly consider UI attacks in its threat model. Given the vulnerabilities that we have identified, and the corresponding lessons learned, we hope that our findings motivate the design and deployment of the next generation of secure browser-based I/O systems that are resilient to UI based attacks.

2 Background and System Model

Creating a secure I/O system within web browsers requires an understanding of the different components of a web application and how they interact. Next, we discuss fundamental implementation details of web applications, our design goals, and our system/threat model.

2.1 Client Server Model

A web application comprises many different components. The server is typically comprised of a database and a front-end layer. The front-end layer is responsible for handling calls made to the server, querying the database, and delivering the results to the client. When the client requests to view the overall website, the server responds to the client with HTML, CSS, and JS files that are used to create the web application on the client. The client is the web browser, which is responsible for creating the Document Object Model (DOM) given HTML files and executing the JavaScript files. The DOM is a tree structured representation of the user interface. JavaScript files can be used to add animation to the site and listen to user events such as clicks or keypresses. JavaScript files have APIs that allow them to directly manipulate the DOM.

2.2 Chokepoints

When considering a user's privacy, it is important to consider which of these components have access to the user's data in plaintext. Oftentimes, web applications are built such that every component has access to the user's information. If the application has a stronger concern for security and privacy, it will encrypt data at one of the chokepoints that are labeled in Figure 1 by utilizing a particular framework or application that encrypts the application's information.

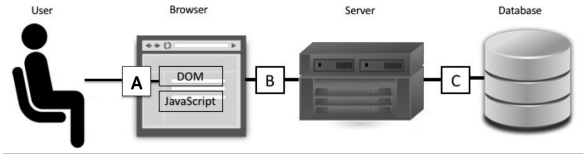

Fig. 1. The client server model. A web application can encrypt data at one of the above chokepoints (A, B, or C).

Server-Database Interface. Systems in this category encrypt the data before it reaches the database server, but allow server-side and client-side application code access to plaintext data. For instance, CryptDB [32] encrypts all of the web application's information before it is stored in the database. Web applications that use CryptDB or other forms of encrypted databases operate at chokepoint "C".

When executing at chokepoint "C", despite storing all of the information securely, all of the server code has access to the plaintext information. In other words, when the front-end logic makes a request to the database, the user's data will be returned to the front-end component in plaintext. Therefore, the user is forced to trust all of the components to the left of the chokepoint to not handle their data maliciously or inappropriately. Therefore, the trusted computing base (TCB) includes all components except for the database. Applications that only employ database encryption are protected against database attacks, but do not provide

strong privacy guarantees since any employee with the database keys can access the user's private information.

Client-Server Interface. Systems in this category only allow for their client side application code (including Javascript/HTML) to have access to the users' plaintext information. The encyption takes place at chokepoint "B". All components to the left of chokepoint "B" are part of the TCB and all components to the right of chokepoint "B" only access the encrypted information. An example of an application that executes at chokepoint "B" is Mylar [33].

It is important to understand there is a distinction between chokepoint "B" and network communication that uses HTTPS. An application that operates at chokepoint "B" encrypts all of the users information prior to sending the data to their backend server. Therefore, the backend server only has access to the encrypted data. With HTTPS, the users' information is encrypted prior to sending the data over the network to the backend server, but upon arrival the data is decrypted. Therefore, with HTTPS the backend server has access to the plaintext data.

Client-Web Application Code Interface. The final chokepoint in Figure 1, chokepoint "A", is located between the client (browser) and untrusted web application code (JavaScript/HTML). In this case, the client/browser is fully in control of the information provided to the web application. All web application components (including JavaScript/HTML) only have access to the encrypted forms of the information, and the client has exclusive access to the plaintext information. This is the chokepoint at which a secure I/O system must execute. While chokepoints "C" and "B" allow some components to be removed from the TCB, by executing at chokepoint "A" the TCB is as small as possible.

2.3 Design Goals

Executing at chokepoint "A" is not the only requirement for a secure I/O system. These are the following goals that must be addressed in order to build a secure input and output system.

Client Side Component Isolation. All client side code that belongs to the parent site must be prevented from accessing the plaintext information as it is being entered by the user, or when it is being rendered as output by the browser. The browser must be able to read the plaintext information, as it must be able to render the plaintext, but all client side code must be prevented from accessing the information. Therefore, there must be a strong boundary that exists between the client side components of the parent site and the client side components that represent the secure input and output fields.

Key Management. The information must be encrypted by one of the user's secret keys. This secret key must only be known by the user and anyone with whom the user shares this key. The keys will be managed by the browser, and it is critical the keys cannot be accessed by any other application on the user's device or web application running inside of the browser.

UI Defenses. The system must be robust against user interface attacks. The system must detect a user interface attack and notify the user before the user enters private information into an insecure field.

Usability. The system must not interfere with the usability of the site. The performance of web applications must not be significantly impacted and the user interface must not be drastically changed by the secure input and output fields on the site. Furthermore, the design of the system must be intuitive so that registering a web application with the system is simple. The usability of the secure I/O system must not be a barrier that inhibits users from adopting the feature.

2.4 Threat & Trust Model

Our primary threat is that of a malicious web application attempting to exploit the user's personal information. Therefore, the attacker in this case is the web application itself. The aim of the attacker is to bypass the secure I/O system that is implemented by the browser.

The system must be designed in order to prevent all confidentiality attacks. As long as the client side web application cannot access the plaintext information, network attackers and server side attackers are not of any concern in regards to confidentiality because the information will be encrypted and the keys will be stored securely in the user's browser.

The system must also be robust against integrity attacks. If the encrypted information is tampered with by an attacker on the client side, network, or server side, the system should identify the attack and notify the user that his information has been corrupted.

The system does not consider attacks against availability. As with many systems, preventing availability attacks is near impossible. For example, a malicious web application can always drop entries with encrypted data (specially since encrypted data is explicitly marked as such in ShadowCrypt like systems).

This system only considers applications that can function with access to only ciphertext. A todo list man-

ager is an example of such an application. A banking application on the other hand needs access to the plaintext data, and is therefore outside of the scope of this paper.

The TCB for this system includes the browser, because the browser is ultimately responsible for rendering the plaintext information. Furthermore, given we must trust the browser, everything beneath the browser must be trusted, this includes the OS and any I/O peripherals. This system does not need to trust any of the client side code or the sites that it is executing on.

We assume that the browser reliably isolates web applications from each other (following the same-origin policy) and that the browser's internals are isolated from all sites. Google Chrome implements a multi-process system [35], based on earlier research [21], that provides strong isolation between untrusted web content of different origins and the browser itself. In Chrome, extensions are modeled as origins that have additional permissions over web content, but, like web content, are still isolated from one another and from the browser itself. Extensions' code runs separately from web applications' code, and we assume that they are securely isolated. However, extensions that alter pages, such as ShadowCrypt, share the same document with web content; we do not assume that elements added to the page by an extension are isolated from the web application (although ShadowCrypt attempts to do this).

The attacker (such as a malicious or compromised web application) is capable of tampering with client side JavaScript code in order to bypass or block the secure I/O mechanism; thus client side JavaScript code is considered untrusted. However, note that client side JavaScript code cannot access the user's sensitive information that is stored inside the browser such as the user's keys and configuration settings, due to access control primitives deployed in modern browsers.

2.5 Shadow DOM

The most important primitive leveraged by ShadowCrypt is the Shadow DOM. The Shadow DOM provides the ability to create a separate encapsulated DOM attached to an existing element. This helps programmers avoid breaking sites due to conflicting CSS selectors or JavaScript variables [12]. In Figure 2, the input and span element within the document tree are both shadow hosts, which host their own shadow tree. Inside of the shadow tree on the left is an input node encapsulated from the rest of the document tree. When this page is

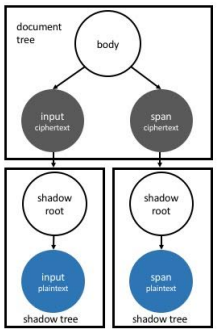

Fig. 2. Shadow DOM creates encapsulated DOMs hosted by an element in the normal DOM.

rendered, the styling and placeholder value for the input node in the Shadow DOM will be rendered rather than the input in the document tree, which is acting as the shadow host. This is the key idea behind ShadowCrypt. If the shadow tree can host the plaintext data, the user will be able to interact with the site as normal. If there can be a barrier placed between the shadow host and its corresponding shadow tree, the plaintext information will be encapsulated away from the client side code entirely, and only the user of the application will be able to see the plaintext information.

Creating Shadow Trees. A shadow tree is created via JavaScript, and cannot be created directly via the HTML of the page. Here is some sample code demonstrating how to create a shadow tree.

```
<button>Hello, world!</button>
<script>
var host = doc.querySelector('button');
var treeRoot = host.attachShadow();
treeRoot.textContent = 'Hello from Shadow!';
</script>
```

The main API call is `attachShadow`, which is applied to a DOM element. The DOM element that calls `attachShadow` becomes the shadow host, as indicated by the variable names. In this above example, the button will render "Hello from Shadow" because the content inside of the shadow tree is rendered rather than the content in the shadow host.

3 ShadowCrypt

Overview. ShadowCrypt is a Chrome extension that runs before each web page is loaded. ShadowCrypt makes input text fields and corresponding output fields secure by using the Shadow DOM to guard the DOM boundary, ensuring client-side JavaScript cannot access the user's private information. In addition to guard-

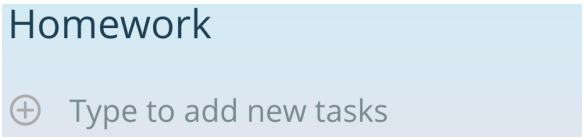

Fig. 3. An example site with a standard input field.

Fig. 4. The same site with the input field made secure by ShadowCrypt. Notice the border around the input field, and the lock on the right side of the input field.

ing the DOM boundary, ShadowCrypt modifies the UI of the input field to demonstrate that the input field has been made secure. Figure 3 and 4 demonstrate how ShadowCrypt transforms the user interface of input fields. ShadowCrypt adds a colored border to the input field, and places a lock icon on the right side of the input field. These UI modifications are used to signal to the user that ShadowCrypt has secured the input field.

For each site registered with ShadowCrypt, there are different configurable settings, demonstrated by Figure 5. The parameter that is most noticeable from the user's perspective is the key color, which is the color of the lock used in the secure input fields. While this may seem like an aesthetic choice, it is used a means of authentication for the user.

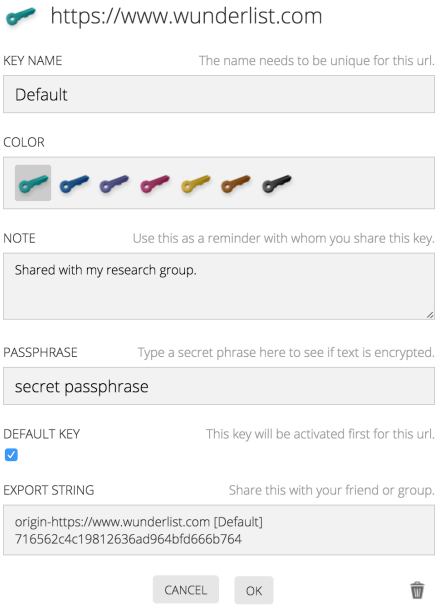

Fig. 5. The ShadowCrypt options for each key. The configurable settings include the key name, key color, note, and passphrase.

Authentication. ShadowCrypt uses the key color and an associated passphrase, which hovers as a tooltip over the lock, to signal to the user that a secure text field is genuine. The user sets a color and passphrase, known only to the ShadowCrypt extension, for each domain with which he plans to use ShadowCrypt. When he sees his correct color and passphrase, the user can be confident there is no attempt to spoof ShadowCrypt's secure text field.

In addition to the passphrase being displayed to the user upon hovering over the lock, upon hitting the keyboard shortcut CTRL-`, ShadowCrypt displays the passphrase in a pop up input window. The pop up window along with the passphrase, which is set to "secret passphrase", is provided in Figure 6.

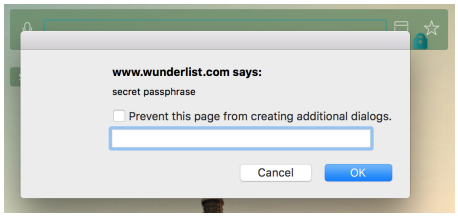

Fig. 6. ShadowCrypt provides an alternate means of entering information into the input field through a pop up window. The pop up window includes the user's passphrase to prove to the user the pop window was produced by ShadowCrypt.

Threat Model. The authentication mechanisms that are put into place by ShadowCrypt may provide a false sense of security against UI attacks. Even though UI attacks were not included in the original ShadowCrypt threat model, we argue that UI attacks are important attack vectors that must be considered for any similar secure I/O framework. *Furthermore, given ShadowCrypt makes an effort to prevent UI attacks through the use of a lock icon, key color, and passphrase, we believe it is important to determine to what extent these measures are actually protecting the user.* This paper will highlight in detail how ShadowCrypt is currently vulnerable to a UI attack. We have developed a detailed UI attack that utilizes many different attack vectors, and the final discussion will present different ways to design a system that is protected against these attack vectors.

4 User Interface Attack

4.1 Rendering Unencrypted Nodes

The first step in the user interface attack is being able to render an unencrypted input node onto the page. ShadowCrypt attempts to detect all input fields on the page, and make them secure; however, we uncovered multiple methods that can be used to get past ShadowCrypt and render unencrypted input nodes on the page.

Undetected Input Types. The first method focuses on how ShadowCrypt selects which input nodes to make secure. ShadowCrypt protects against text inputs, textareas, iframes, and nodes that have editable content. This covers many of the possible ways client applications can ask for user input; however, it does not cover all of the input types. HTML5 added a few new input types, and some of these types can be used to directly receive user input. The input type *search* behaves just like a regular text input but is intended to be used for search fields [3]. Therefore, a client application can simply change all inputs with type "text" to type "search" in order to bypass ShadowCrypt. While it is conceivable for ShadowCrypt to also scan the page for search entries, this attack vector can similarly be executed using input elements of type *email*. In this case, it is important for ShadowCrypt, and any future secure I/O system, to not manipulate email fields, because a user's email is typically required for logging into an application, and therefore, must be entered in plaintext. Given email input types must not be protected in order for the application to be usable, this method can be applied in order to render an unencrypted input field on the page.

Undetected DOM Events. Rather than relying upon these different input types, we also found a DOM manipulation that ShadowCrypt is not currently listening to, which can lead to an input node being left unencrypted. If an input form is set to be the following:

```
1 <form class="new-task">
2   <input class="main-input" type="checkbox" name="text"
        placeholder="New task" />
3   <input class="false-input" type="text" name="false-text"
        placeholder="New task" />
4   <input type="submit" value="Submit">
5 </form>
```

ShadowCrypt will transform the input with class `false-input` (line 3) into a secure input node, and ignore the input with class `main-input` (line 2) because it is of type "checkbox". Now, once the page finishes rendering, the user interface attack can run a very simple line of JavaScript:

```
1 document.querySelector('.main-input').type = "text";
```

This will transform the input node which was originally a checkbox into an input with type text. This modification of the DOM can be listened to using a Mutation Observer, and ShadowCrypt catches this modification, but does not correctly transform this new input node into a secure input node. Rather, the new input with type text is left unencrypted. This demonstrates the difficulty of comprehensively securing all input nodes.

Input-Like Elements. Lastly, in addition to these above methods, a client application could simply create an input-like `div` element, that will not be noticed by ShadowCrypt. A user could create a `div`, and register all of the necessary click and keyboard events to make the `div` feel like an input field. A good example of this is a Google Document. Google Documents do not use input fields, rather it is a `div` that has all of the necessary listeners to take the user's input. Similarly, a client application can create a `div` that feels like an input, and this input-like `div` would not be made secure by ShadowCrypt. *Therefore, no amount of patching to ShadowCrypt will prevent a client application from rendering an unencrypted input field on the page.* This makes it clear that a secure I/O system must operate under the threat model that includes an attacker rendering an unencrypted input field onto the page. Given this attack vector, it is important for a secure I/O system to have in-content security indicators that cannot be mocked or a means of alerting the user that there is an unencrypted input field on the page.

These above methods can lead to certain input nodes being left visibly unencrypted; however, the attack is not complete. A user that insists on having encrypted data will not use the application, reducing the attack to a denial of service. The attack must appear to have ShadowCrypt working properly in order to intercept sensitive data. Next, we discuss approaches to enhance the stealthiness of the attack.

4.2 Positioning

In order to make the user feel that he is interacting with a secure input node, the unencrypted input node can be placed directly on top of a secure input node. Therefore, the secure input field's border color will be present, and the ShadowCrypt lock will be on the right side of the input. Even though there is an unencrypted input node being rendered directly on top of the secure input node, it will still look exactly like a secure input field. Figure

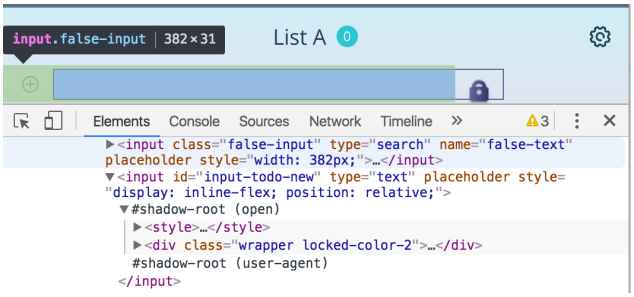

Fig. 7. This site is being attacked. The attack includes an input field with type search, which makes this input field unencrypted. It is positioned directly on top of the secure input field. The unencrypted input field is slightly less wide in order to not conflict with the the lock at the end of the secure input.

7 shows the extent of the false input overlaid on the secure input, along with the DOM trees.

In order to position the insecure input field on top of the secure input field we had to set the insecure field's CSS properties to be:

```
.false-input { position: absolute; z-index: 10; }
```

Setting the `position` to be absolute allows for the position of the secure input field to not be affected by the presence of the insecure input field. The `z-index` property allows for the insecure input field to be rendered on top of the secure input field. The width of the insecure input field is set to be exactly 40px less than the width of the secure input field, because the width of the lock is 40px. Since the lock is not covered by anything, all of the functionality of the lock is retained. With this type of styling on the insecure input field, the user interface attacks become impossible to notice. An advanced user can explore the HTML source code to identify the attack, but based on the UI alone, a user cannot tell there are two input fields being rendered on top of each other.

4.3 Border

So far the look of the site has been perfectly masked, and the user cannot identify the insecure input field. When the user clicks into the input field, the user expects to see the border become slightly thicker. ShadowCrypt sets the border width to be 2px when the input field is focused (the user clicks into the input field). However, now when the user clicks into the input field, the ShadowCrypt input field is not focused because the insecure input field has been focused. There is no way to force the secure input field into a focused state, because there is no API for letting two disjoint elements be focused at the same time, so this portion of ShadowCrypt cannot be mocked perfectly.

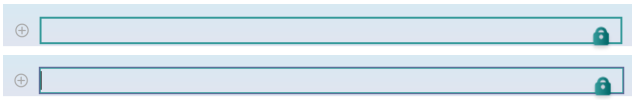

Fig. 8. Above: Benign, a 2px green border rendered by the ShadowCrypt. Below: Under attack, a 1px green border and a 1px purple border around that.

In order to attempt to mock the thicker border, we apply a border with a fixed color around the insecure input field. In order to apply this border we added a `div` before the insecure input field that is set to not be displayed unless the insecure input field is focused. The fixed color that we use as the extra border is the key color that is set by default by ShadowCrypt. This is because even though the key color is a configurable parameter, we believe that a typical user will not actually change the default key color. Furthermore, even if the user does change the key color, *it is very difficult to notice that the extra thickness on the border is a different color*. The key color was originally included in the ShadowCrypt design in order to make user interface attacks more difficult; however, it seems that this feature is not enough for the user to notice when an attacker is getting in the way of the key color working as it is supposed to. We present empirical validation for our observations in Section 6.

Figure 8 demonstrates how similar the focused inputs are even when the extra thickness is a different color. Note that text-based input type in our examples is the dominant use case for a system like ShadowCrypt. The user study, which is discussed in the next two sections, empirically demonstrates that users have a hard time noticing the extra thickness is a fixed purple color. A further discussion on how to best include UI markers in a secure I/O system will be explored in Section 7.

4.4 Keyboard Shortcuts

ShadowCrypt comes with two keyboard shortcuts. CTRL-` opens a new window in which the user can enter text. This new window includes the user's passphrase as means of authentication. CTRL-SPACE is used to toggle the input field between a locked and unlocked state. Both of these keyboard shortcuts cannot be mocked without access to the internals of the shadow tree. The pop-up is a separate window where the attacker has no control, so an overlay attack would not be possible. We do not know the passphrase, so creating a mock pop-up window with the correct passphrase would be very difficult. In a similar manner to how we mocked the thicker border, we could use the default passphrase

in the pop-up window; however, in this case if the user changes the passphrase, it will be very obvious something is broken/attacked.

CTRL-SPACE changes the state of the lock into a grayed out lock and the default border is removed. Even though the style of the unlocked input is static, it cannot be mimicked because the functionality of the grayed out lock cannot be replicated. We originally attempted to listen for an input of CTRL-SPACE in the insecure input field, and then simulate a CTRL-SPACE event in the secure input field, which would change the state of the lock without degrading the functionality at all. However, creating key events with a specific key is blocked by JavaScript APIs because the `keyCode` field is read-only.

Since the keyboard shortcuts cannot be mocked, whenever a user hits CTRL, we turn off the attack until the user has submitted that particular entry. Here is the code demonstrating the attack being turned off upon CTRL being pressed:

```
'keydown .false-input': function(e) {
 if (e.keyCode === 17) {
   targetNode = document.querySelector('#input-todo-new');
   falseNode = document.querySelector('.false-input');
   targetNode.value = falseNode.value;
   ev = new Event("input");
   targetNode.dispatchEvent(ev);
   $('.false-input').emulateTab();
   $('.false-input').remove();
   $('.div-border').remove();
} }
```

If CTRL is hit, we set the value of the secure input host node to be the value that is currently in the insecure input node. We then trigger an input event on the host node. This event will be captured by ShadowCrypt and interpreted as if it came from the internal input node of the shadow tree because of event retargetting. This event will trigger ShadowCrypt to encrypt this new value. By setting the value of the host node, and then triggering the encryption process to take place, it is as if the user has been entering the text into the secure input field all along. We then simulate hitting TAB so that the secure input field is focused, and we finish the attack by removing the insecure input field and the border that is used to mock the extra thickness. Once the user has submitted the entry, the attack is turned back on by adding the insecure input field and the mock border `div`. With the insecure input field in the DOM, we simulate hitting SHIFT-TAB, which will return the focus to the insecure input field. These steps allow for the attack to be turned off in a smooth manner upon the user attempting to use the keyboard shortcuts, which allows for the attack to not be noticed even when the user uses one of the keyboard shortcuts.

4.5 Man in the Middle

With the attack as it stands the user will most likely be tricked into entering his information into an insecure input field because he thinks it is secure. Therefore, masking the secure input is complete. The final step requires mocking the secure output. Each output `<span>` is highlighted with the user's key color. Once again, we do not have access to the user's key color, so this step is difficult. However, rather than attempting to highlight each secure output, for each entry we store both the plaintext and the ciphertext (which we obtain by triggering the ShadowCrypt encryption scheme) in the database. Then we only return the ciphertext to the client so that ShadowCrypt's secure output works as expected and we do not have to mock anything for secure output to work as expected. Therefore, this attack acts as a man-in-the-middle attack as we gain access to the plaintext information, and then allow for ShadowCrypt to work as expected when rendering the output.

4.6 Summary of UI Attack

The key to the UI attack is making the user believe he is interacting with a secure input node, when in fact the input node is insecure. First, we managed to bypass ShadowCrypt's input detection schemes in order to render an insecure input node onto the page. Next, we rendered a secure input node directly beneath the insecure input node, which allowed for the insecure input node to appear to have a lock indicator, which would trick the user into believing the insecure input node was actually secure. We were unable to perfectly imitate the secure input node; however, we were able to mock the important security indicators allowing for our UI attack to be essentially undetectable. In order to measure the stealthiness of the attack we administered a user study on Amazon Mechanical Turk. The design of the study will be covered in Section 5 and the results will be covered in Section 6.

5 User Study Design

Our user study was designed with two parallel goals: to evaluate the usability of ShadowCrypt (which was not evaluated in the original paper) and to examine the effectiveness of the UI attacks identified in Section 4.

To do this, we implemented a sample application to work with ShadowCrypt. In an online study with 105

participants, we asked participants to first use ShadowCrypt within this sample application and then attempt to determine whether an experimental version of the application was compromised. Overall, we aim to evaluate the stealthiness of our user interface attack.

We recruited 105 participants via the Mechanical Turk platform. Following best practices, we required participants to have above 95% approval rating, and to have completed 100 approved human intelligence tasks (HITs). The survey took less than 30 minutes to complete, and we paid the participants $3.50. We sought permission from our Institutional Review Board (IRB) to conduct the study; however, this study was determined to not be human subjects research as defined by DHHS regulations.

5.1 Sample Application

Our sample application is a to-do list manager, strongly based on the sample Meteor application [5]. This application allows users to create an account, and then once logged in to create *to-do lists* and add *tasks* to each list. We selected a to-do list manager as our sample application since it focuses on text-based input types, which is the dominant use case for a system like ShadowCrypt [23].

5.2 Survey Design

Using the sample application. To begin, the participant joins the sample application.[1] Upon registering, the participant is instructed to create a few lists and a few tasks per list in order to gain a deeper understanding of the application. In order to measure the user experience of ShadowCrypt, we first measure the user experience of the to-do list application without ShadowCrypt. We ask participants to agree or disagree with five statements on a 5-point Likert scale, including "Agree or Disagree: Using the to-do list application was fun" and "Agree or Disagree: adding a task on the to-do list application was difficult" (full list of questions in Table 2).

Using ShadowCrypt. The next section walks the participant through installing ShadowCrypt. In order for the participant to understand how to use ShadowCrypt, and why it is useful, we created a tutorial video. The tutorial video can be found at http://bit.ly/shadowcrypt-tutorial-video. We hoped the video would be more engaging than text-based training. We carefully designed the video training to cover all of the relevant information from the original ShadowCrypt paper, in order to ensure participants had access to all necessary security knowledge. At the very end of the survey, after affirming to the participant that they would be paid, we ask for an honest answer to whether or not the user watched the full video. Out of the 105 participants, 100% of participants claimed to have watched the video completely.

With ShadowCrypt installed and the tutorial complete, the participant creates a few lists and tasks with ShadowCrypt enabled. This section also walks the user through navigating to the options menu and updating the key color and passphrase. Furthermore, this section walks the user through the aspects of ShadowCrypt that must be checked in order to make sure ShadowCrypt is working correctly. Therefore, this explains to the user how to create a new task using the CTRL-` keyboard shortcut, as well as how to toggle the input field between a locked and unlocked state using CTRL-SPACE keyboard shortcut. There is also an explanation on passphrase, and it discusses how to hover over the lock in order to display the passphrase. To understand the user experience with ShadowCrypt, we next ask the same user experience Likert questions as in the case of the sample application without ShadowCrypt (full list of questions in Table 2) and two direct usability questions (Table 3).

Experimental Application. We tell the user that he is about to go to an experimental version of the to-do list application. This experimental application has a 50% chance of being compromised with the UI attack that was discussed in Section 4. The user is instructed to create a few lists and tasks, and explore modifying the key color and passphrase to determine if the site is compromised. Once the user is finished using the experimental application, he reports whether or not he believes the application was compromised, along with an explanation of his answer.

5.3 User Study Limitations

While the user study was an efficient means of acquiring information on the usability of ShadowCrypt and the effectiveness of the UI attack, it does have a few limitations. First, it is difficult to determine if the usability rankings were honest responses. It is possible the users claimed ShadowCrypt to be usable because that

1 The full user study can be found at https://docs.google.com/document/d/10FSaJfjPd4zjozT__2__-947qdv2Y74Aljguvs64__6lFA

is what they believed we wanted to hear. Second, this test represents a best-case scenario for identifying an attack, since the user is explicitly told to look for an attack. It is possible the percentage of users who identified the attack would be even less in real life, when they would not be focused on identifying the attack. Lastly, ShadowCrypt requires the user to manage settings for each of their keys. The concept of key management is very complicated, and it is difficult to say if users really understood what they were doing when they were configuring the various settings for each key. Despite these limitations, the results from the user study, which are presented in the following section, make clear that ShadowCrypt in its current state is reasonably usable, but vulnerable to a comprehensive UI attack.

6 User Study Results

Table 4 details the demographics of the 105 participants that completed the user study. Among all recruited participants, 26 dropped out of the study (not counted among the 105) during the introduction where they watched a video about ShadowCrypt and explored a few parts of its interface; we excluded these participants from our results. Key results are summarized in Table 1. The most important takeaway is that only 1 out of 59 participants who were under attack identified the user interface attack. This indicates that the UI attack is stealthy and that the existing security indicators are not effective. While it is possible there is a way to make the visual authentication mechanism more visible, passive security indicators (that require the user to notice a problem) are often unsuccessful [16, 38]. A system that is proactive and notifies the user of a UI attack might be more effective at preventing users from being fooled. A deeper discussion on how to prevent UI attacks will be presented in Section 7.3.

We turned the attack off whenever the user hit the CTRL key because we could not mock the keyboard shortcuts. Our expectation was that users would not use the keyboard shortcuts very often, and therefore very few tasks would be fully encrypted. Only five participants mentioned the keyboard shortcuts in the survey, although this only provides limited information because it comes from a free response question. Of the 570 tasks created by the participants who were under attack, only 43 tasks were created using the keyboard shortcut (8%).

Participant Type	Participant Response	Number of Participants
Not Under Attack	Site not compromised	40 (87%)
	Site compromised	6 (13%)
Under Attack	Site not compromised	58 (98.3%)
	Site compromised	1 (1.7%)

Table 1. The results demonstrate that only 1 out of 59 users who were under attack noticed the attack. 6 of the users who were not under attack claimed to be under attack, which is most likely a result of trying guess the correct answer.

Statement	W	Z	p	r
Using the to-do list application was fun.	219	1.12	.251	.078
Adding a task on the to-do list application was difficult.	22.5	0.91	.350	.063
Adding a list on the to-do list application was difficult.	13.5	1.01	.398	.070
Adding a task on the to-do list application was tedious.	25.5	3.55	.001	.245
Adding a list on the to-do list application was tedious.	50.0	2.80	.006	.194

Table 2. Results of the Wilcoxon Signed-Rank test comparing the user experience results of the application with and without ShadowCrypt.

Statement	Ratings Median	Q1	Q3
You believe ShadowCrypt made the to-do list application less user friendly.	1	1	2
You would recommend ShadowCrypt to a friend.	4	3	4

Table 3. The participants' responses (1 is strongly disagree; 5 is strongly agree) indicate that they are more likely to recommend ShadowCrypt to a friend than not, and that on average ShadowCrypt did not make the to-do list application less user friendly. Q1 and Q3 indicate the first and the third quartile respectively.

Therefore, the UI attack gained access to 92% of the tasks.[2]

For the participants who believed they were under attack, we asked how they could tell. Here are some excerpts:

– "I do not see my updated changes to either the key color or passphrase ..." 3 participants not under attack noted discrepancies between configurable passphrase and color. This suggests that performing these checks can be confusing and error-prone.

2 Note that our attack always succeeded in obtaining plaintext if the CTRL key was not used.

Metric	Percentage	Participants
Male	59.05%	62
Female	40.95%	43
H.S. or less	8.57%	9
Some college	29.52%	31
Associate	14.29%	15
Bachelor	36.19%	38
Trade/Technical	4.76%	5
Master	6.66%	7
18-24 years	12.38%	13
25-34 years	39.05%	41
35-44 years	33.33%	35
45-54 years	9.52%	10
55-64 years	2.86%	3
65+ years	2.86%	3

Table 4. Demographics of our sample.

- "I could do everything even though the locked icon was there," wrote one participant not under attack. This participant may not have understood what ShadowCrypt is meant to prevent.
- "ctrl space doesnt work," wrote one participant not under attack. This was the only mention of a keyboard shortcut. However, it is unclear whether this was due to a technical issue or a misunderstanding of the shortcut.

6.1 Usability

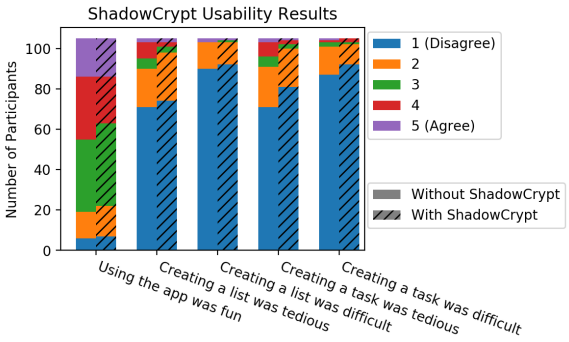

Fig. 9. Usability results: Users responses to the Likert questions with and without ShadowCrypt.

Despite the user interface attack being highly effective, which demonstrates serious privacy vulnerabilities, the user study demonstrated that ShadowCrypt doesn't significantly degrade the usability of the visited website. We asked the participants to agree or disagree with five statements on a 5-point Likert scale, with and without ShadowCrypt (1 meaning strongly disagree, 5 meaning strongly agree). In order to test the difference between the responses to the Likert questions with and without ShadowCrypt we used a Wilcoxon Signed-Rank test. The questions with their associated test statistic, z-score, p-value, and effect size (r) are provided in Table 2. For three of the five questions, the p value is greater than 0.25, which indicates there is not significant evidence there is a difference in the distribution of the responses with and without ShadowCrypt. For the two questions regarding the tediousness of creating a task and list, the p value is less than 0.05, which indicates there is significant evidence there is a difference in the distribution of the responses with and without ShadowCrypt. In these two questions, the responses indicate creating a task and list with ShadowCrypt was actually less tedious. This is potentially due to ordering effects; in our user study we always taught the user the todo list application first and then ShadowCrypt second. Ideally we would have varied the order to prevent ordering effects, but in this case that was not possible because we could not teach ShadowCrypt before teaching the user how to use the todo list application. *This result suggests that any decrease in usability caused by ShadowCrypt was small enough that it was outweighed by the ordering effect.* The full results to the Likert questions in Table 2 are provided in Figure 9.

In addition to the questions intended for comparison, we asked two direct usability questions (Table 3). The results highlight that the vast majority of users found ShadowCrypt not to negatively impact the site from a usability point of view, and that the majority of users would recommend ShadowCrypt to a friend.

Based on these results, it is clear that ShadowCrypt does not significantly impact the user experience of the site. This is a very important part of the secure I/O system, because if the system is not user friendly, there will not be strong adoption.

7 Discussion

In this section, we discuss the fundamentals behind the secure I/O system, how ShadowCrypt fails to address these fundamental issues, and how this information can be generalized for the future development of a secure I/O system with Internet browsers.

7.1 Shadow DOM future

The user study demonstrated ShadowCrypt is vulnerable to a user interface attack. Even though it could not be mocked fully, the areas that could not be mocked were very hard to notice. Furthermore, the pop-up method for entering text could not be mocked, but the poor usability of entering information into a pop-up led many participants to not use this method. It is clear that the user interface attack is an attack vector that must be considered; however, there are more serious concerns preventing ShadowCrypt, in its current state, from providing secure I/O. Most importantly, the Shadow DOM boundary that ShadowCrypt relies upon is no longer available. Below, we will discuss the mechanisms that allow client side code to access the plaintext information that is supposed to be encapsulated inside of the shadow tree. The Shadow DOM was originally developed as a means to support web developers, and was never intended to be used as a security enhancement. As such, it is not surprising that the specifications of the Shadow DOM have changed to the point that it can no longer provide the security guarantees that ShadowCrypt relied upon. Our UI attack *does not* depend on any of the weaknesses of the Shadow DOM boundary, but it is important to understand all of the potential attack vectors, so that a secure I/O system can ultimately be developed.

DOM Properties. The most simple means of accessing the shadow tree is through the `shadowRoot` property on the host node. The original ShadowCrypt code tried to establish the Shadow DOM boundary by setting the property to `null`. This removed the native binding by which the browser would return the shadow tree to the application. However, Chrome has since changed its implementation of DOM properties so that they use "getter" functions on the prototype chain [2]. To interpose fully on the getter's behavior would pose additional overhead. While an extension can replace the getter in the application's window, any other windows, such as the content of an `iframe`, would still expose the original getter. The full DOM API provides several ways to create `iframes`; an extension would have to mediate all of them to prevent access to the native `shadowRoot` getter. This approach is more challenging not only in terms of development but also maintenance and performance overhead.

Multiple Shadow Trees. Another means of accessing the information inside of a shadow tree is attaching a second shadow tree (a "younger" one) to the same shadow tree. A `<shadow>` element in this younger shadow tree will render the contents of the older shadow tree, and a method (`getDistributedNodes`) is provided to access what the older shadow tree contains. ShadowCrypt originally attempted to prevent the creation of additional shadow trees on DOM elements that already host ShadowCrypt's shadow tree, but this is similarly no longer effective, due to Chrome changing the implementation of DOM methods from elements' own properties to functions defined in the element's prototype. The following code gains access to the information in the input field:

```
root = doc.querySelector('#input-todo-new')
    .attachShadow({mode: 'open'});
root.innerHTML = "<shadow></shadow>";
shadow = root.querySelector("shadow");
shadowCryptNodes = shadow.getDistributedNodes();
plaintext = shadowCryptNodes[1].querySelector('.delegate')
    .value;
```

Ensuring complete mediation [37] of access to the `attachShadow` method is challenging because every window object has a reference to the method, and there are many ways to create new windows. While this way of accessing the plaintext information currently works, the use of multiple shadow trees has been deprecated, and will not be supported for long [6].

The deprecation of multiple shadow trees, which took place in April 2015, is bad news for utilizing Shadow DOM as a primitive for a secure I/O system. Currently, `input` fields are supported by a shadow tree created and managed by the browser itself. Therefore, creating a shadow tree on `input` nodes has been deprecated along with the deprecation of multiple shadow trees [20]. One could work around this by (i) changing the document structure to add a parent to the input for hosting the shadow tree or (ii) attaching a shadow tree to the parent of the input and mirroring the input's siblings as well. These alternative approaches involve either application-visible changes or monitoring of additional parts of the page, both of which increase the complexity and performance overhead of the design.

Given that Shadow DOM is not suitable for providing the fundamental browser primitive behind a secure I/O system, it is important to determine what browser primitive can support a secure I/O system.

7.2 iFrame Based Isolation

Given the current browser primitives, it seems the only option that would provide strong isolation between elements on one page would be to use an `iframe`, which is currently being explored with the priv.ly project [4]. Due to the same origin policy, the `iframe` and the client application would be of different origins so that there

would be no way for the client application to get any information from the content inside of the `iframe`. Furthermore, all mouse and keyboard events inside of an `iframe` cannot be accessed by the parent window, so there is no need to worry about the event model or event retargetting with iframes. This system would function very similar to ShadowCrypt, but rather than relying upon shadow trees to store the plaintext information, the plaintext information will be hosted within a dedicated `iframe`. The parent window will only have access to the encrypted information. In order to communicate between the parent window and the iframes, the browser extension can utilize postMessages in order to have cross-origin communications. In this system, the user would enter text into an input within an `iframe`, and upon submissions, the `iframe` would encrypt the plaintext information, and send a message to the parent window. The browser extension would register a listener on the parent window to take the encrypted information, and submit the encrypted information as if the user was submitting a form in the parent window.

Utilizing iframes is very effective at creating secure input fields. However, there are serious performance downsides for secure output. There are only a handful of input fields per web page, but there can be hundreds of output fields. If each `iframe` causes a performance hit, utilizing iframes would be inefficient. Future work involves measuring the performance hit for a secure I/O system backed by iframes.

8 Related Work

Given a browser primitive that can provide isolation between the plaintext information and the rest of the web site, it is important for the system to be robust against UI attacks. In this section we explore the related work that can be developed upon in order to create a secure I/O system for browsers that is robust against UI attacks.

Previous research on UI security for the web initially focused on securely displaying the browser's *chrome*, which are the UI elements that surround the arbitrary, untrusted web content (not to be confused with Google's web browser named Chrome). These elements including the address bar and connection security indicators. Felten et al. [18] and Ye et al. [39] have demonstrated attacks that take advantage of older browsers' flexibility in allowing untrusted web page code to hide critical elements of the browser chrome. Their attacks render a fake address bar from within the web page, which allows the attacker to display any address, thus fooling a user into thinking that she is visiting a different site. In later browsers, the ability to hide the address bar is removed. Additionally, Chen et al. have proposed a formal specification of correct behavior of the chrome's elements [10].

A similar progression has appeared in mobile UI security. In current mobile operating systems, there is no secure indication of what application is on screen. Bianchi et al. propose to display the current application's developer in an area controlled by the operating system [9], behaving similar to the HTTPS Extended-Validation indicator in a browser.

Similarly, Petracca et al. relied upon a trusted display area where the system could display messages to the user without any risk of an application tampering with this area of the user interface [31]. In addition to a trusted display area, Petracca et al. implements an enhanced version of user driven access control to sensor data. Each request is uniquely identified by a tuple combining the application owner, set of sensors requested, operation being requested on the set of sensors, the user event triggering the request, the user interface of the widget capturing the event, and then user interface configuration capturing the event. Petracca et al. found that this resulted in a similar number of authorization requests to first-use authorization mechanisms.

Later research has turned its attention to the security of UI elements within the page content, involving attacks such as clickjacking. Huang et al. propose that clickjacking and related UI attacks occur because sensitive UI elements are presented *out of context* [24]. They define a notion of context that includes visual context, which is what the user should see, and temporal context, which is the timing of a user's action. Many UI attacks occur by tricking a user into believing they are interacting with one thing, but really they are interacting with something completely different. With *InContext* visual integrity is enforced by comparing an OS screenshot of the area with the sensitive element, and a reference bitmap of the sensitive element in isolation [24]. This defense protects users against many types of UI attacks. For instance, imagine a PayPal element for a $1000 product is being rendered on a page. On top of the price is a node making it look like the price is only $10. In this case, a user click action on the Pay button will be blocked because the sensitive PayPal element has been overlaid with an element, which makes the OS screenshot and reference bitmap not equal. The core idea behind this event is to sanitize the user actions

inside of the sensitive element to make sure the context is correct.

The InContext integrity defense will not prevent our UI attack. In our case, the secure input field will be covered by an insecure input field such that the interface looks exactly as expected. InContext sanitizes actions that are delivered to sensitive elements by preventing users from acting on a sensitive element if the element is obstructed in any way. However, our UI attack lures the user into interacting with an insecure element that belongs to the untrusted application. Filtering out interactions with the trusted element thus does not prevent the user from entering text into the attacker's untrusted element in this attack.

Dhamija et al. propose using a user-specific image to provide mutual authentication between the user and the application [14]. Their system moves particularly sensitive inputs, such as password fields, into the browser chrome so that they can be securely displayed without interference from other web content. ShadowCrypt's popup input is similar, showing a prompt from the browser chrome, but with a textual passphrase instead of an image. It also attempts a weakened variant by showing a custom border color within web content, but our experiments showed that this is not noticeable enough to be secure. The use of a different color and passphrase for each site in the interest of mutual authentication presents a mental burden for the user, and it remains to be determined how much this affects the security of such a system.

Dong et al. propose a more heavy-handed approach to ensuring that sensitive user interface elements are presented in context, by presenting them in a standalone rendering engine completely separate from the browser process [15]. This rendering engine, called the Crypton-Kernel, establishes a secure path between the application and the GPU display buffers. While this effectively prevents any attacks from web content that tries to overlay secure inputs and outputs, it also limits the design flexibility of those elements, due to the complete separation.

Légaré et al. propose a solution that relies on the implementation details of Chrome browser to prevent UI attacks [29]. The platform developed by Légaré et al., BeesWax, uses the small 4px square that is dedicated to each Chrome extension to display to the user whether or not the in-focus input node is secure. This 4px square cannot be modified or obscured by the web application, so this signal cannot be tampered with or covered up by an attacker. However, it is unclear if this small icon can be used to successfully mitigate UI attacks. Furthermore, this countermeasure has a strong dependence on the specifics of Chrome, and it does not scale well to other browsers or mobile devices.

A retroactive change to prevent interactive elements from being overlaid would break existing sites. Inputs can end up overlaid intentionally, for example, when a page creates a modal window [8] containing a form. Kaminsky has identified many other design features which can interfere with an element's rendition, such as scrolling, clipping, transformations, and shading effects [26].

Iron Frame. Instead of preventing overlays, Kaminsky proposes Iron Frame [25], a system to *detect* overlays and to allow the browser and the application to indicate this, so that the user would know if it was safe to interact. The Iron Frame system informs iframes of whether they are completely visible. This allows the iframes disable themselves when they are not in view, in order to prevent UI attacks. Furthermore, when the iframes are completely visible and in focus, the browser can update the address bar [26]. Updating the address bar could be a very strong visual indicator to the user that he is interacting with a secure input field, and this would be impossible to mock.

Trusting the user? Unfortunately, even with the security guarantees of Iron Frame, the secure I/O system must rely upon the user to identify a missing security indicator in order to witness the attack. Relying upon the user to notice the UI attack is a very poor defense, even if the indicator is very obvious and impossible to mock. Wu et al. report that users do not pay attention to additional security indicators [38]. Therefore, ideally the secure I/O design would be able to identify that there is an element overlaid on top of it, and alert the user. Huang et al. demonstrate that there are limitations of CSS checking, but it is possible that CSS checking might be the best approach for this problem. In order to have a robust defense against UI attacks the secure I/O system cannot rely upon the user to identify the attack; instead, the system must identify the attack and alert the user.

9 Conclusions

In this paper, we explored the practical limitations of secure I/O systems for web applications, such as ShadowCrypt. ShadowCrypt provided clear insights into the problem with web privacy today, and it highlighted the benefits of a secure I/O system for web applica-

tions. However, we have uncovered a powerful UI attack that can steal the users' sensitive data, despite ShadowCrypt's attempts to protect it. In our user study, only 1 out of 59 participants noticed our UI attack, which demonstrates the stealthiness of the UI redress attack. Furthermore, existing UI defenses, such as InContext integrity, do not mitigate our attack. In addition to our UI attack, we have demonstrated that ShadowCrypt's implementation based on Shadow DOM no longer provides the necessary isolation between trusted and untrusted data. While a similar approach could use iframes to provide the isolation, it would impose a heavier performance overhead. These problems of technical implementation and secure user interface design are important to the ultimate security and usability of a secure I/O system. While the utility of a secure I/O system is appealing, we conclude that future research and browser enhancements on performant UI isolation and secure UI design are needed to develop a usable system.

Acknowledgements. We thank Giuseppe Petracca for his help on this paper. We thank the anonymous reviewers for their feedback. This material is in part based upon work supported by the National Science Foundation under Grants No. CNS-1553437 and CNS-1409415. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Company info | facebook. http://newsroom.fb.com/company-info/.

[2] Dom attributes now on the prototype chain. https://goo.gl/DEitmJ.

[3] Html5 input types. http://goo.gl/oDbNhf.

[4] Priv.ly project homepage. https://priv.ly/.

[5] Todos | build a collaborative task app with meteor. https://www.meteor.com/todos.

[6] Webapps/web components april2015 meeting. https://goo.gl/NZ0he2.

[7] A. Afanasyev, J. A. Halderman, S. Ruoti, K. Seamons, Y. Yu, D. Zappala, and L. Zhang. Content-based security for the web. In *Proceedings of the 2016 New Security Paradigms Workshop*, pages 49–60. ACM, 2016.

[8] Anthony. Best practices for modal windows. http://uxmovement.com/forms/best-practices-for-modal-windows/, March 2011.

[9] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 931–948. IEEE, 2015.

[10] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y. M. Wang. A systematic approach to uncover security flaws in gui logic. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 71–85, May 2007.

[11] K. Collins. Google collects Android users' locations even when location services are disabled. https://qz.com/1131515/google-collects-android-users-locations-even-when-location-services-are-disabled/, November 2017.

[12] D. Cooney. Shadow dom 101, January 2013. http://goo.gl/Rbu0w.

[13] P. De Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Protected web components: Hiding sensitive information in the shadows. *IT Professional*, 17(1):36–43, 2015.

[14] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88. ACM, 2005.

[15] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting sensitive web content from client-side vulnerabilities with cryptons. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1311–1324. ACM, 2013.

[16] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1065–1074, New York, NY, USA, 2008. ACM.

[17] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander. Helping johnny 2.0 to encrypt his facebook conversations. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 11. ACM, 2012.

[18] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web spoofing: An internet con game. *Software World*, 28(2):6–8, 1997.

[19] B. Fuhry, W. Tighzert, and F. Kerschbaum. Encrypting analytical web applications. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop*, pages 35–46. ACM, 2016.

[20] D. Glazkov and H. Ito. Shadow dom w3c working draft 15 december 2015, December 2015. https://goo.gl/JgL7e8.

[21] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 402–416, May 2008.

[22] D. Guarini. Experts say facebook leak of 6 million users' data might be bigger than we thought, June 2013. http://www.huffingtonpost.com/2013/06/27/facebook-leak-data_n_3510100.html.

[23] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1028–1039, New York, NY, USA, 2014. ACM.

[24] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 413–428, Bellevue, WA, 2012. USENIX.

[25] D. Kaminsky. Want these * bugs off my * internet. def con 23, August 2015. https://youtu.be/9wx2TnaRSGs.

[26] D. Kaminsky. Want these * bugs off my * internet. def con 23. slide 71-72, August 2015. http://www.slideshare.net/dakami/i-want-these-bugs-off-my-internet-51423044.

[27] F. Lardinois. Gmail now has more than 1b monthly active users, February 2016. https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/.

[28] S. Liang, Y. Zhang, B. Li, X. Guo, H. Guo, X. He, Z. Liu, and C. Jia. Shadowpwd: practical browser-based password manager with a security token. In *Proceedings of the ACM Turing 50th Celebration Conference-China*, page 30. ACM, 2017.

[29] J.-S. Légaré, R. Sumi, and W. Aiello. Beeswax: a platform for private web apps. In *Proceedings on Privacy Enhancing Technologies*, pages 24–40. PETS, 2016.

[30] A. T. Ozcan, C. Gemicioglu, K. Onarlioglu, M. Weissbacher, C. Mulliner, W. Robertson, and E. Kirda. Babelcrypt: The universal encryption layer for mobile messaging applications. In *International Conference on Financial Cryptography and Data Security*, pages 355–369. Springer, 2015.

[31] G. Petracca, A.-A. Reineh, Y. Sun, J. Grossklags, and T. Jaeger. Aware: Preventing abuse of privacy-sensitive sensors via operation bindings. In *26th USENIX Security Symposium*, pages 379–396. USENIX, 2017.

[32] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[33] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, Apr. 2014. USENIX Association.

[34] C. Reber. Our future: Wunderlist joins microsoft, June 2015. https://www.wunderlist.com/blog/our-future-wunderlist-joins-microsoft/.

[35] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 219–232, New York, NY, USA, 2009. ACM.

[36] S. Ruoti, D. Zappala, and K. Seamons. Messageguard: Retrofitting the web with user-to-user encryption. *interface*, 9:6.

[37] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[38] M. Wu, R. C. Miller, and S. L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 601–610, New York, NY, USA, 2006. ACM.

[39] Z. E. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):153–186, 2005.