Cracking ShadowCrypt: User Interface Attacks Against Security Enhancing Chrome Extensions

Mike Freyberger
mdf3@princeton.edu

Prateek Mittal
pmittal@princeton.edu

Michelle Mazurek
mmazurek@cs.umd.edu

ABSTRACT

In web applications it is difficult to assure users their data is being stored privately. In order to increase the privacy of users' data on all web applications it is important to design a system for secure input and output (I/O) within Internet browsers. This system 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. Recently, security researchers at UC Berkeley developed ShadowCrypt, which is a browser extension that can enable secure I/O on all text based web applications [15]. 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. One important aspect of a secure I/O system is that it must be robust against user interface (UI) attacks, which ShadowCrypt fails to address. First, we developed a comprehensive UI attack and tested the efficacy of the attack through a user study administered on Amazon Mechanical Turk. Only 1 (1.7%) of the 59 participants who were under attack successfully noticed the UI attack, which validates the stealthiness of the attack. Second, we present multiple attacks against ShadowCrypt that do not rely upon UI deception. These attacks expose the privacy weaknesses of Shadow DOM — the key browser primitive leveraged by ShadowCrypt. Finally, we present a framework for possible countermeasures in order to provide new generalizable knowledge that can be used in the future development of a secure I/O system within Internet browsers.

Keywords

ShadowCrypt; User Interface Attack; Web Secure I/O

1. INTRODUCTION

Smartphone and web applications have transformed peoples' lives. For example, calendar applications allow users to easily keep track of recurring events and birthdays. Facebook allows connecting with old friends and current friends to be seamless. Uber makes riding a taxi reliable with Uber's rating system and GPS tracking [22]. While all of these applications are incredibly useful, they all rely upon the user trusting them with their sensitive data.

We must trust Google will not leak our calendar with sensitive information regarding our meetings and tasks. We must trust Facebook will not allow employees to spy on our conversations with our friends. We must trust Uber executives will not stalk us when we use their service to get home after a late night at the office. However, Uber was recently fined by New York city for doing just that: a manager accessed information about specific users without their permission via an internal company interface [19]. As incidents like this are reported and privacy policies change rapidly, users may find it difficult to maintain the required trust.

The core problem is that 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.

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 web application would only be able to access an encrypted version 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). In an effort to create secure I/O within Internet browsers, researchers at UC Berkeley created a system called ShadowCrypt [15]. In this paper, we will explore the implementation of ShadowCrypt and the attack vectors that are possible against this system. Following a discussion of the attack vectors, we will present the important ideas behind possible countermeasures. By testing and attacking ShadowCrypt, we have determined important guiding principles that can be used when designing secure I/O within Internet browsers.

1.1 Contributions

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

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

provides in-content security indicators for all secure input nodes. All secure input nodes have a lock placed at the end of the input, a user specified border color, and a user specified passphrase. However, attackers can mimic these in-content security indicators in order to trick the user into thinking an insecure input node is secure. 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 bypasing 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.

More details on the attack are provided in Section 4.

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 while 59 were under attack. Of the 59 users who were under attack, only 1 user (1.7%) identified the attack. This user study validates the stealthiness of the user interface attack. More details on the design of the user study and the user study results are provided 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 wrote 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 not sufficient. This analysis demonstrates that it is important for the design of a secure I/O system to rely upon browser primitives that are originally designed for security and privacy.

Countermeasures. In addition to demonstrating the various types of attacks and validating their effectiveness, we present a framework for countermeasures against these attacks. The countermeasure must be able to provide DOM isolation and be robust against UI attacks. We hope that this paper serves to provide a foundation for a secure I/O system, which can be deployed into browsers in the near future.

2. BACKGROUND

2.1 Client Server Model

A web application comprises many different components. A diagram of these components and their interconnections is provided in Figure 1. 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. A few popular languages used to create the front-end of the server include Node.js [7], Python [11], Go [3], and PHP [8]. Once the front-end logic on the server has gathered the resources to serve the web request, 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

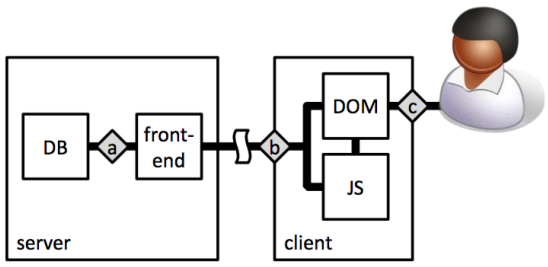

Figure 1: The client server model. A web application can encrypt data at one of the above chokepoints (A, B, or C). Image Source: He et al. [15].

When considering a user's privacy, it is important to consider which of these components have access to the user's data in plaintext. Often times, 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. For instance, CryptDB [20] encrypts all of the web application's information before it is stored in the database. Web applications that use CryptDB operate at chokepoint "A".

When executing at chokepoint "A", 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 right of the chokepoint to not handle their data maliciously or inappropriately. Therefore, the TCB includes all components except for the database. Applications that execute at chokepoint "A" are protected against database attacks, but do not provide strong privacy guarantees since any employee with the database keys can access user's pri-

vate information.

Similarly, when a system executes at chokepoint "B", all components to the right of the chokepoint are part of the TCB and all components to the left of the chokepoint only access the encrypted information. An example of an application that executes at chokepoint "B" is Mylar [21], which was developed by the same group that developed CryptDB.

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

2.3 Design Goals

Executing at chokepoint "C" 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.

- All client side JavaScript must be prevented from accessing the plaintext information as it 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 JavaScript must be prevented from accessing the information. Therefore, there must be a strong boundary that exists between the DOM of the majority of the page and the nodes that represent the secure input and output fields.

- 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.

- The system must be robust against user interface attacks. The system must be able to detect a user interface attack and notify the user before the user has a chance to enter private information into an insecure field.

- The system must not interfere with the usability of the site. The performance time must not be significantly reduced 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 Model

The main threat that is considered is a malicious web application that is exploiting the user's personal information. Therefore, the system must be designed in order 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. In this case, each secure input field must mark the encrypted data with an identifiable signature, so that the system knows to decrypt that entry into a secure output node. Therefore, a simple availability attack would be to always drop entries with the given signature.

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.

The attacker is capable of tampering with client side JavaScript code in order to bypass or block the secure I/O mechanism. However, client side JavaScript code cannot access the user's information that is stored inside the browser such as the user's keys and configuration settings.

2.5 Shadow DOM

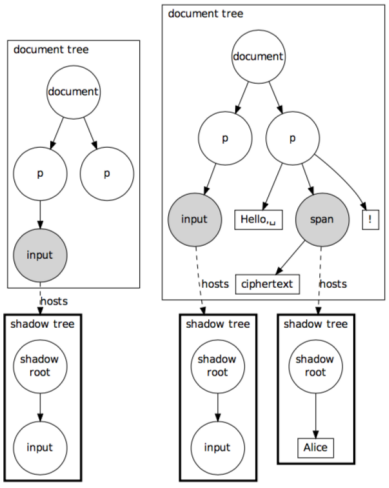

Figure 2: Shadow DOM creates encapsulated DOMs hosted by an element in the normal DOM. Image Source: He et al. [15].

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 element on the top left part of the diagram is a shadow host, which hosts the shadow tree in the lower left part of the diagram. Inside of this shadow tree is an input node encapsulated from the rest of the document tree. When this page is rendered, the styling and place-

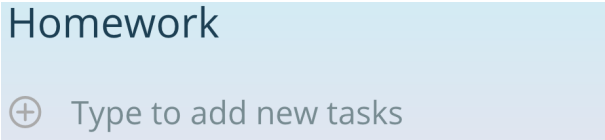

Figure 3: An example site with a standard input field.

Figure 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.

holder 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. This idea of hiding the plaintext in the shadow tree is demonstrated in the right half of Figure 2. Here the shadow host (the span in the top right box) only has access to the ciphertext information, and cannot see that the user entered Alice, which is the value stored in the shadow tree.

2.5.1 *Creating Shadow Roots*

Shadow roots are created via JavaScript, and cannot be created directly via the HTML of the page. Here is some sample code demonstrating how to create shadow roots.

```
<button>Hello, world!</button>
<script>
var host = doc.querySelector('button');
var root = host.attachShadow({'mode': 'open'});
root.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 root is rendered rather than the content in the shadow host.

3. SHADOWCRYPT

The inspiration for creating a system that supports secure input and output came from studying ShadowCrypt. ShadowCrypt is built with the threat model that is stated above, and is driven by many of the above design goals.

3.1 Overview

ShadowCrypt is a Chrome extension that runs before each web page is loaded. ShadowCrypt transforms each input text field into a secure input text field. ShadowCrypt makes input text 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 guarding 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 also uses the DOM isolation provided by Shadow DOM to implement secure output. When rendering text on the page, ShadowCrypt searches for the encryption fingerprint "=?shadowcrypt", which would indicate that the content was originally encrypted by ShadowCrypt. In order to present this information securely to the user, ShadowCrypt creates a `span` within a Shadow DOM that cannot be accessed by client side JavaScript. The content is then decrypted, and the plain text value is stored in the Shadow DOM and the encrypted value is stored in the normal DOM. Therefore, the web page can only access the encrypted value, but the plain text value will be rendered by the browser to the user. ShadowCrypt highlights information that has been decrypted to demonstrate to the user that the output is secure.

ShadowCrypt only runs on sites that the user has specified. Figure 5 demonstrates the options menu where the user can see all of the sites that are registered with ShadowCrypt. For each site, there are different configurable settings, demonstrated by Figure 6. 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 also a means of authentication for the user.

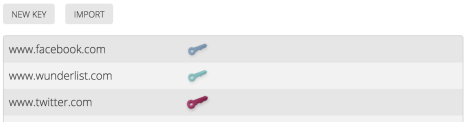

Figure 5: The ShadowCrypt options menu that lists all domains registered with ShadowCrypt.

3.2 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 secret color and passphrase, the user can be confident that 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 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 7.

3.3 User Interface Attacks

The authentication mechanisms that are put into place by ShadowCrypt may provide a false sense of security against UI attacks, which were not included in the original ShadowCrypt threat model. We argue that UI attacks are impor-

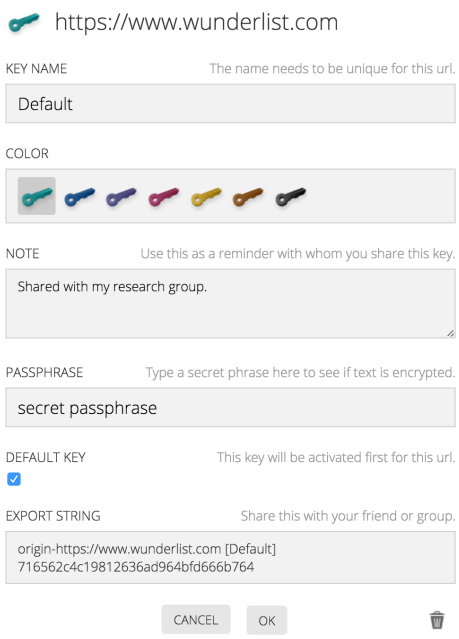

Figure 6: The ShadowCrypt options for each key. The configurable settings include the key name, key color, note, and passphrase.

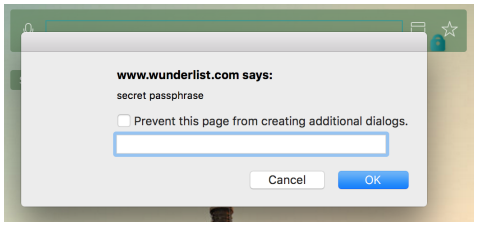

Figure 7: ShadowCrypt provides an alternate means of entering information into the input field through a pop up window. The pop up window includes the user's passphrase to prove to the user the pop window was produced by ShadowCrypt.

tant attack vectors that must considered for any similar secure I/O framework. The original ShadowCrypt paper makes clear that it is not concerned with DoS attacks, side-channel attacks, and UI attacks. Ignoring DoS attacks and side-channel attacks, such as the length of the plain text information, is very reasonable, as these types of attacks are unavoidable. However, ignoring UI attacks is a serious weakness in the design of ShadowCrypt. It is important to assure the user that a malicious web application cannot fool him into exposing his private information, which should be protected by ShadowCrypt or a similar secure I/O system. This paper will highlight in detail how ShadowCrypt is currently vulnerable to a UI attack. Through diving into the details of the UI attack, different attack vectors have been identified, and the final discussion will present different ways to design a system that is protected against these attack vectors.

3.4 Architecture

Given an overview of the fundamentals of ShadowCrypt,

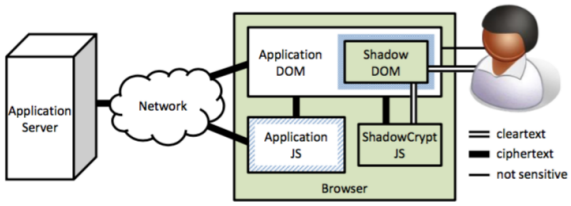

Figure 8: ShadowCrypt allows for the user's plaintext to only be available to the user, Shadow DOM, and ShadowCrypt. The Application DOM, Application JavaScript, and all server side code can only access the ciphertext. Image Source: He et al. [15].

Figure 8 demonstrates ShadowCrypt's architecture. First of all, the ShadowCrypt JavaScript file is loaded into its own isolated environment and run before the web page begins rendering or executing any of its own JavaScript. ShadowCrypt begins by accessing the user's keys, which are stored locally in the browser, and checks to see if the user has a key registered with the current domain. If so, ShadowCrypt then parses the DOM to find any input nodes that must be transformed into secure input nodes. It also parses for text to see if any elements contain the ShadowCrypt fingerprint: `=?shadowcrypt`. If there is any text element containing this fingerprint, the plaintext information will be determined by decrypting the original element's value with the user's key associated with the website's domain. Then the plaintext information will be encapsulated inside the Shadow DOM and the ciphertext will remain in the normal DOM. As Figure 8 demonstrates the application's JavaScript and DOM cannot access any of the user's plaintext information because the web application has been transformed, and all of the user's sensitive information is kept safe in the Shadow DOM. The plaintext information is only available to the user, the Shadow DOM, and the ShadowCrypt JavaScript code.

4. USER INTERFACE ATTACK

In this section, we describe an attack against the ShadowCrypt user interface. In Section 7, we will demonstrate that the Shadow DOM boundary does not provide the expected full isolation; the UI attack in this section, however, does not depend on that attack and will work even if Shadow DOM isolation is perfect.

The ShadowCrypt threat model admits that UI attacks may be possible; however, we believe that more thoroughly exploring the vectors for and effectiveness of such attacks is an important step toward creating a secure I/O system robust against such attacks.

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.

The first method focuses on how ShadowCrypt selects which inputs 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 [4]. 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.

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 var target = document.querySelector('.main-
    input');
2 target.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.

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.

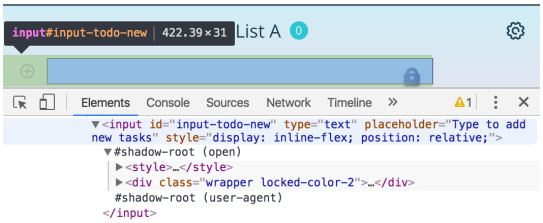

Figure 9: This site is not being attacked. The secure input field has been transformed by ShadowCrypt to host a shadow tree which will store the user's plaintext information.

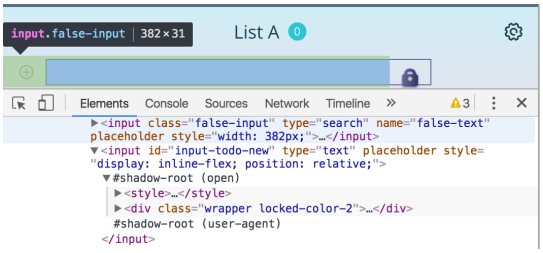

Figure 10: This site is being attacked. The attack includes an input field with type search, which makes this input field unencrypted. It is positioned directly on top of the secure input field. The unencrypted input field is slightly less wide in order to not conflict with the functionality of the lock at the end of the secure input.

These above methods can lead to select input nodes being left unencrypted; however, the attack is not complete. The attack must be supplemented in order to make it stealthy. If a client application is noticeably breaking ShadowCrypt, the user will complain, or find a new application. In order for the attack to be complete, it must be stealthier. The following aspects of the attack focus on 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. Figures 9 and 10 demonstrates how there is no difference between the look of the input field even though in one case there is an insecure input field on top of the secure input field. 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:

```
1 .false-input {
2   position: absolute;
3   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

Figure 11: A 2px green border that has been rendered by the ShadowCrypt code working correctly on a site that is not attacked.

Figure 12: A 1px green border and a 1px purple border around that. This is a demonstration of how the user interface attack attempts to mock the thicker border when the input field is focused.

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, so this portion of ShadowCrypt cannot be mocked perfectly.

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. Therefore, even though the key color is a configurable parameter, we assume the user will not actually change the 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. Figures 11 and 12 demonstrate how similar the focused inputs are even when the extra thickness is a different color. The user study, which is discussed in the next two sections, 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. We do not know the passphrase, so creating a mock pop 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 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 the user hits submit we have direct access to the plaintext because he was typing into an insecure input field. In order to get the associated ciphertext we trigger the ShadowCrypt encryption scheme, and then the ciphertext is available in the shadow host. In order to trigger the ShadowCrypt encryption we set shadow host's value to be the plaintext value. We then simulate an `input` event on the host node. This event will be captured by ShadowCrypt since the event's target is a shadow host. ShadowCrypt will proceed to encrypt the plaintext information, and set the shadow host's value to the ciphertext. We then submit to the database both the ciphertext and plaintext.

One complication is that the user can turn off ShadowCrypt at any given time, and then it is important to actually return the plaintext back to the client. To handle this case, we first check to see if the secure input field is locked. In order to check if the input is locked, we probe the ShadowCrypt encryption scheme with a SPACE. ShadowCrypt will only encrypt the SPACE if the input is locked, otherwise ShadowCrypt will not encrypt the SPACE and then we know that the input is not locked. If the secure input field is locked, we set a metadata field `stolen` to be true on the plaintext value. If the input is not locked, we set `stolen` to be false, so that the entry is returned to the client in plaintext form as expected.

All in all, in order to mock secure output we only return to the client the ciphertext values so that ShadowCrypt works as intended for the output fields. This concludes all of the attack vectors that must be implemented in order to execute a stealthy user interface attack. 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. The experimental version of the application was in fact being attacked for 50% of the participants, assigned randomly.

5.1 Sample Application

The sample application was strongly based on the one of the sample Meteor applications[9]. 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 set the login to be based on the user's Mechanical Turk id rather than email. The application is hosted using Digital Ocean[1] and is backed by a MongoDB[6] database hosted with MongoLab[5].

5.2 Survey Design

5.2.1 Using the sample application

To begin, the participant joins the sample application. 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 "was the application fun" and "was creating a task difficult."

5.2.2 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. 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 minimize biasing participants' security knowledge. At the very end of the survey, after affirming 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, 0 claimed not 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. There is also an explanation on passphrase, and it discusses how to hover over the lock in order to display the passphrase. To compare the user experience with and without Shadowcrypt, we next ask the same user experience Likert questions as in Section 5.2.1.

5.2.3 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. We make sure to be very clear that a compromised application will not put them or their computer in any danger. The user then is directed to the experimental application, where he is instructed to create a few lists and tasks, and explore mod-

ifying 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 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 1 details the demographics of the 105 participants who completed the user study. Key results for these users

Gender	
Male	Female
62 59.05%	43 40.95%

Age					
18-24	25-34	35-44	45-54	55-64	65+
13 12.38%	41 39.05%	35 33.33%	10 9.52%	3 2.86%	3 2.86%

Education					
High School or less	Some college	Associate	Bachelor	Trade Technical Professional	Master
9 8.57%	31 29.52%	15 14.29%	38 36.19%	5 4.76%	7 6.66%

Table 1: Participant Demographics.

are summarized in Table 2.

The most important takeaway from the user study is that only 1.7% of the participants who were under attack identified the user interface attack. This indicates that the positioning of the insecure input field was perfectly on top of the secure input field, so that the user could not tell the difference. Furthermore, this demonstrates the thicker border is not significantly noticeable by the users. Either the users did not change the key color and the extra 1px purple border was perfectly placed so that it looked just like ShadowCrypt, or users did change their key color but having an extra 1px border of a different color was not noticeable. Either way,

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

Table 2: The results demonstrate that 98.3% of users who were under attack did not notice the attack. 6 of the users who were not under attack claimed to be under attack, which is most likely a result of users being confused by ShadowCrypt or trying guess the correct answer.

this extra border was not an effective in-content security indicator. While it is possible there is a way to make the visual authentication mechanism more visible, it might be important for the secure I/O system to not rely upon the user to look for their visual symbol. 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 assumption was that users would not use the keyboard shortcuts very often, and therefore very few tasks would be fully encrypted. In fact, the 59 participants who were under attack created 570 tasks; we had access to the plaintext information for 527 and 43 were full encrypted. Therefore, the UI attack gained access to 92% of the tasks, and was identified by only 1.7% of participants.

6.1 Usability

Despite the user interface attack being highly effective, which demonstrates serious privacy vulnerabilities, the user study demonstrated ShadowCrypt to be user friendly — doesn't degrade the usability of the visited website. Based on the user experience questions, it seems that ShadowCrypt in its current state does not have a negative impact on the user experience of the application. 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 two-sample chi square test. The questions with their associated p-values are provided in Table 3. Each p-value is above 0.38; as such, there is no evidence of a difference in the distribution of the responses with and without ShadowCrypt.

The full results to the Likert questions in Table 3 are provided in Figure 13. In addition to the questions intended for comparison, we asked two direct questions to determine the usability of ShadowCrypt. These questions are provided in Table 4.

Based on these results, it is clear that ShadowCrypt does not negatively 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.

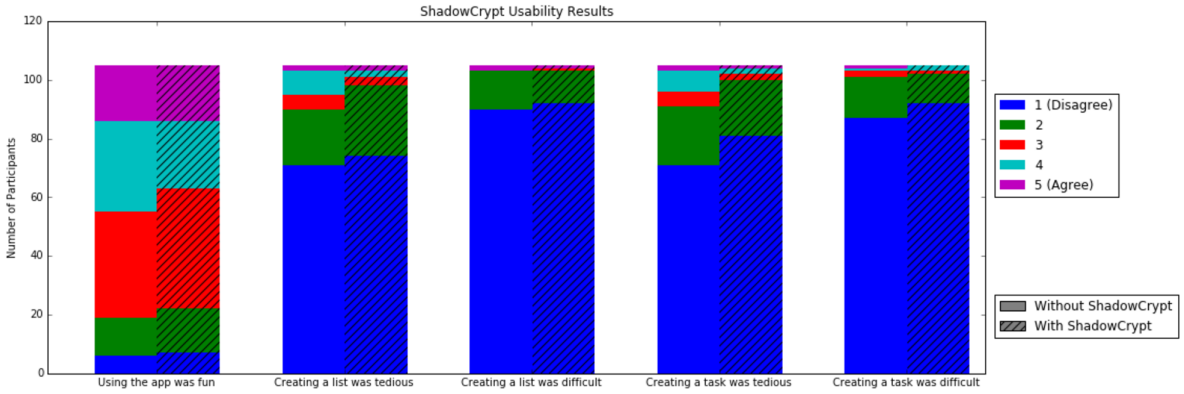

Figure 13: The usability results indicate that the users responses to the Likert questions were not significantly different with and without ShadowCrypt.

Statement	$\tilde{\chi}^2$ p-value
Using the to-do list application was fun.	.8
Adding a list on the to-do list application was tedious.	.4
Adding a list on the to-do list application was difficult.	.95
Adding a task on the to-do list application was tedious.	.39
Adding a task on the to-do list application was difficult.	.83

Table 3: Table comparing the user experience results of the application with and without ShadowCrypt. The p-values indicate there is no statistical evidence that there is a difference in the distribution of the results with and without ShadowCrypt.

Statement	Average Rating
You believe ShadowCrypt made the to-do list application less user friendly.	1.33 (σ^2= .37)
You would recommend ShadowCrypt to a friend.	3.69 (σ^2= .9)

Table 4: The participants responses 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.

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. Client side code can 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.

7.1.1 *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`. However, DOM properties within Chrome are no longer properties on instance objects, but are available through `getters` on the prototype chain [2]. Therefore, since in the original implementation ShadowCrypt relies upon setting the `shadowRoot` to `null`, this method will no longer work. ShadowCrypt must now change the `getter` method for the shadowRoot property. While changing the `getter` method for the `shadowRoot` property is possible, the client side application could get the original `getter` method back by accessing the method inside of an `iframe`.

7.1.2 *Multiple Shadow Roots*

Another means of accessing the information inside of the shadow tree is through the `<shadow>` insertion point inside of a younger shadow root. The client application can create another `shadowRoot` on the host node after ShadowCrypt has created a shadow root. ShadowCrypt originally attempted to prevent the creation of shadow roots on DOM elements that have been transformed by ShadowCrypt, but this is similarly no longer available due to Chrome changing how it handles DOM properties and functions. Now, the youngest

shadow root, which is owned by the client application, can use the `<shadow>` insertion point in order to gain access to all of the information inside of ShadowCrypt's shadow tree. The following code will gain 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;
```

While this means of getting access to the plaintext information is currently functional, the use of multiple shadow roots has been deprecated, and will not be supported for long [10].

The deprecation of multiple shadow roots, which took place in April 2015, is unfortunately very bad news for utilizing Shadow DOM as a primitive for a secure I/O system. Currently, `input` fields are supported by a user-agent shadow root. Therefore, creating a shadow root on `input` nodes has been deprecated along with the deprecation of multiple shadow roots [14]. Without the ability to create shadow roots on `input` nodes, input nodes cannot be made secure.

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. Then, even if there can be strong isolation between the DOM and the plaintext, it is important to make sure the system is not susceptible to UI attacks.

7.2 iCrypt

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`. 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 the secure output portion of the system. There are only a handful of input fields per web page, but there can be 100's of output fields per web page. 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.

7.3 UI Attack Defenses

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. UI attacks occur because sensitive UI elements are presented *out of context* [16]. The context 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*, the defense against UI attacks presented by *Clickjacking: Attacks and Defenses* by Huang et al., 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. 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.

This defense will unfortunately not prevent the UI attack that was discussed above. In this case, the secure input field will be covered by an insecure input field so the interface looks exactly as expected. Furthermore, this defense sanitizes actions that are delivered to sensitive elements. The defense prevents users from acting in sensitive elements if the element is obstructed in any way. In this case, the UI attack is preventing the user from interacting with the sensitive elements. Rather than trying to lure the user into interacting with a sensitive element without them realizing it, this attack is making the user interact with a non-sensitive element, hoping the user believes the element is sensitive. Therefore, the UI attack must be identified independent of user actions, which is not the case in Huang et al.'s defense. The defense in this case cannot wait for a user action and then make sure the context is correct, because in this case the sensitive element will receive no user actions if the attack is implemented correctly.

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. [13] 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.

7.3.1 Iron Frame

In order to protect the secure input fields, either the sensitive element must scan the page for elements that are being overlaid or the sensitive element must provide clear in-content security indicators. Depending on the state of these

indicators, the user knows when to stop interacting with the page. One possible indicator that would be a very strong indicator, and impossible for the attack to mock, would be to change the URL in the browser when the user is interacting with the secure input. This approach is suggested by the developer of Iron Frame, a recent defense mechanism designed to prevent UI attacks. Iron Frame was presented at DefCon in 2015, and will hopefully be deployed into browsers soon [17]. Iron Frame allows for iframes to detect if they are completely viewable. This allows for iframes to be disabled when they are not in view in order to prevent UI attacks. Furthermore, since Iron Frames allow for iframes to have output exclusivity, the browser can update the address bar when the user is interacting with a secured frame [18]. Updating the address bar would 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.

7.3.2 *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. 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. demonstrates that there are limitations of CSS checking, but it is possible that CSS checking would 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.

8. CONCLUSIONS

ShadowCrypt provided clear insights into the problem with web privacy today, and it is clear a secure I/O system is necessary. ShadowCrypt seemed to provide such a system; however, the Shadow DOM isolation is no longer a strong barrier, and the privacy guarantees of ShadowCrypt aren't met anymore. Furthermore, it is important for the secure I/O system to address UI attacks, which was outside of ShadowCrypt's original threat model. While a similar approach that is backed by iframes rather than the Shadow DOM is possible, it is always very important to consider usability and performance in the design of this system. It is probable that a new browser primitive must be developed that can support a strong barrier between certain DOM elements, while still being highly efficient. Understanding the weaknesses of ShadowCrypt and design goals for a secure I/O system will hopefully aid the development of a secure I/O system, which we hope to see deployed into browsers soon.

9. REFERENCES

[1] Digital ocean. https://www.digitalocean.com/.

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

[3] The go programming language. https://golang.org/.

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

[5] mlab. https://mlab.com/.

[6] Mongodb. https://www.mongodb.org/.

[7] Node.js. https://nodejs.org/en/.

[8] Php: Hypertext preprocessor. https://secure.php.net/.

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

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

[11] Welcome to python.org. https://www.python.org/.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] D. Kaminsky. Want these * bugs off my * internet. def con 23, August 2015. https://www.youtube.com/watch?v=9wx2TnaRSGs.

[18] 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.

[19] K. Kokalitcheva. Uber to settle with n.y. attorney general over 'god view' privacy breach, January 2016. http://goo.gl/ZHqFb9.

[20] 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.

[21] 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.

[22] H. Williams. Uber vs. taxis: Choice says uberx is cheaper and safer, September 2015. http://goo.gl/iXFYLw.