The Limitations of Web Privacy:

Cracking ShadowCrypt

Mike Freyberger

Electrical Engineering

Princeton University

Senior Thesis

Adviser: Prof. Prateek Mittal

Second Reader: Prof. Paul Cuff

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Electrical Engineering

Princeton University

May 2016

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

/s Mike Freyberger

Mike Freyberger

The Limitations of Web Privacy: Cracking ShadowCrypt

Mike Freyberger

Abstract

In web settings it is very difficult to assure users their data is being kept private. Recent work has been done to develop a browser extension that can make text based web applications private without trusting any part of the application itself. The name of this project is ShadowCrypt and is available in the Chrome Extension store. This extension fails to address a serious user interface attack which will be described in detail. The effectiveness of this attack was measured through a user study administered through Amazon Mechanical Turk, in which only 5.4% of participants noticed the attack. In addition to a demonstration of the effectiveness of the user interface attack, I created multiple fundamental attacks against ShadowCrypt, exposing the privacy weaknesses of ShadowDOM. Finally, a framework for possible countermeasures is presented in order to provide clear guidelines on how to design a secure input and output system within internet browsers.

Acknowledgments

I would like to thank my adviser Prof. Mittal for all of his insight and introducing me to ShadowCrypt. It was a great joy to work with him, and I am so grateful for all of his guidance as I researched web privacy. He has always pushed me to think about the problem deeply. Also, I am very excited to submit this research to CCS on May 23 because of Prof. Mittal's encouragement. Even if it doesn't get accepted, this has been a great experience of learning what it is like to do cutting edge research.

I would like to thank Michelle Mazuerk, an expert in the interconnection between usability and privacy. I went through many revisions of the user study, and her expertise was so valuable as I developed the user study.

I would like to thank Warren He and Devdatta Akhawe, the original authors of ShadowCrypt, for all their help as I started to work on this project. They were able to direct me to the most important parts of the project, and without their guidance this project would not have been as novel. They were able to direct me to the core ideas that would lead to doing research that would make serious contributions to the field of web privacy.

I would like to thank Paul Prucnal and Philip Ma for their guidance I as endeavored on a self driven thesis. Receiving feedback all throughout the process was really helpful, and made sure I stayed on track.

Lastly, I want to thank Amy Cutchin for all the amazing proof reading ever since my first research paper during my Sophomore year.

Contents

1 Introduction

Many smartphone and web applications today make peoples' lives much easier by making certain tasks much more efficient. 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 with friends reliable with Uber's rating system and GPS tracking [1]. While all of these applications are incredibly useful, they all rely upon the user trusting them with their sensitive data.

We trust Google will not leak our calendar with sensitive information regarding our meetings and tasks. We trust Facebook will not allow their employees to spy on our conversations with our friends. We trust Uber executives will not stalk us when we use their service to get home after a late night at the office. However, with privacy policies changing rapidly it is becoming very difficult to trust these entities with our data. Just recently Uber was fined by New York for a privacy breach. The New York manager was accessing information about a few specific users without their permission by using an internal tool that provides a full view of drivers and passengers in a specific area [2].

The core problem at hand is that when a company that is fighting to make it in an intense market, their privacy policies are not their primary focus. This leads to privacy policies being poorly defined, poorly enforced, or poorly executed. Users need to be put back in control in order for them to be confident their data is secure.

One important step towards putting users back in control of their information is creating a framework for secure input and output (I/O) within Internet browsers. This would allow a user to enter any information to any site, and be assured that this information will only be accessible by the user. In an effort to create secure I/O on the Internet, researchers at Berkeley created a system called ShadowCrypt [3]. However, this application is vulnerable to user interface attacks and relies upon a browser primitive that no longer provides the necessary isolation between secure and insecure information. This paper will explore the implementation of ShadowCrypt and the attack vectors that are possible against this system. Following a discussion of the attack surface, the important ideas behind possible countermeasures will be presented.

1.1 Contributions

I demonstrate, through a combination of attacks, that ShadowCrypt is vulnerable to a serious user interface attack. The attack is comprised of the following aspects:

- **Render an insecure input field** onto the DOM by bypasing the ShadowCrypt scheme of converting all input fields.

- **Position** the insecure input field directly on top of a secure input field so that the insecure input field looks and feels secure.
- I guess the **border** color used by ShadowCrypt, and demonstrate that color is an insufficient means of authentication for the user.
- I execute this attack as a **man-in-the-middle** attack by first gathering the plaintext information, and then triggering ShadowCrypt to encrypt the information so that the user cannot tell his information was intercepted.

This user interface attack was tested through a user study administered on Amazon Mechanical Turk. Of the 56 users who completed the study successfully, *only* ***5.4%*** of users identified the attack. This user study validates the effectiveness of the user interface attack.

In addition to the user interface attack, this paper presents a deep study of ShadowDOM, the browser primitive that empowers ShadowCrypt. I have written attacks against ShadowDOM directly, which demonstrate the privacy guarantees of the ShadowDOM, which ShadowCrypt relies upon, are not sufficient. Therefore, these attacks allow me to directly access the plaintext information via client side JavaScript code, which is the most important thing to prevent in a secure I/O system in the browser.

In addition to demonstrating the various types of attacks and validating their effectiveness, I present a framework for countermeasures against these attacks. The countermeasure must be able to provide DOM isolation and be robust against click jacking attacks. Hopefully this work provides the foundation for a secure I/O system, which can be deployed into browsers in the near future.

1.2 Paper Layout

This paper will begin by providing background information for the problem of secure I/O within browsers. Section 2 will review the problem layout, design goals, and the threat model. The fundamentals of ShadowCrypt, an application that attempts to provide secure I/O, will be covered in Section 3. Browser primitives that ShadowCrypt relies on will be explored in Section 4, and Section 5 will be fully dedicated to exploring the Shadow DOM, the core browser primitive used in ShadowCrypt. Section 6 will provide an overview of the user interface attack that is possible against ShadowCrypt. In order to measure the effectiveness and stealthiness of this attack, I created a user study. I will go over the design of the user study in Section 7 and results of the user study in Section 8. Section 9 will provide further discussion on possible countermeasures to the user interface attack, and a possible approach to implement secure I/O within the browser. Section 10 will provide concluding insights and thoughts.

2 Background

2.1 Client Server Model

There are many components that comprise a web application. 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 NodeJs [4], Python [5], Go [6], and PHP [7]. 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 clicking or keypresses. JavaScript files have API's that allow them to directly manipulate the DOM.

2.2 Chokepoints

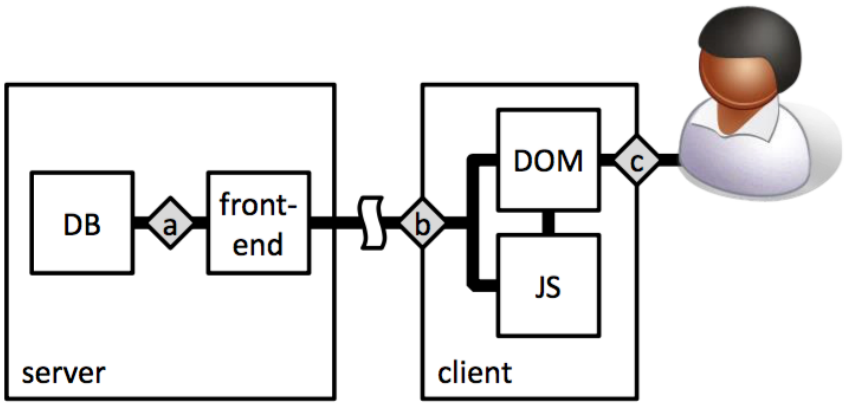


Figure 1: The client server model. A system can implement securuty at one of the above chokepoints (A, B, or C). Image Source: [3].

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, systems are built such that every component has access to the user's information. If the system has a stronger concern for security and privacy, they will operate at one of the chokepoints that are labeled in Figure 1. When operating at chokepoint "A", the database does not have access to the user's plaintext information. Therefore, the data is stored in an encrypted format on the

database. Web applications that use CryptDB [8] as their backend database execute 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 private 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 and need not be trusted. An example of an application that executes at chokepoint is Mylar [9], which was developed by the same group that developed CryptDB. With Mylar all sever side code only has access to the encrypted information.

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. This is the chokepoint where the security system must execute in order to put the user in control of his information.

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.

1. 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.

2. 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.

3. The system must be robust against user interface attacks. The system should 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.

4. 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.

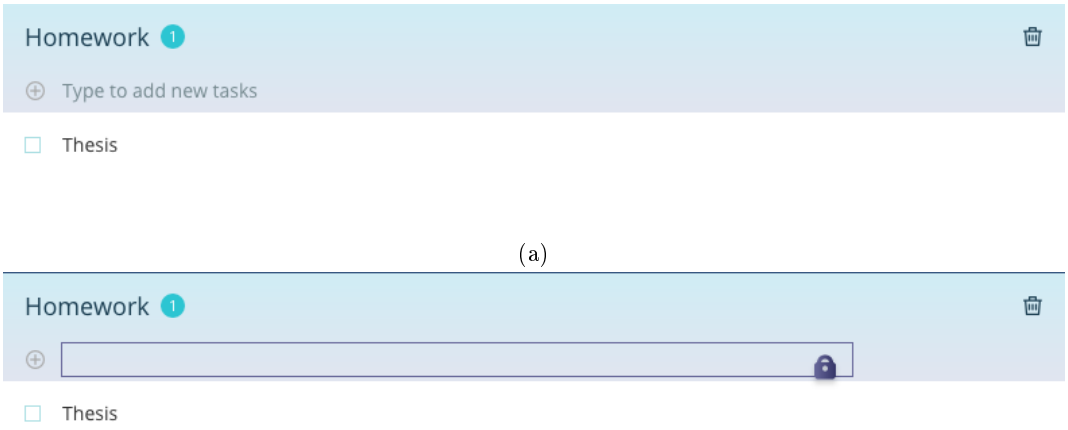
2.4 Threat Model

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.



(a)

(b)

Figure 2: (a) An example site with a standard input field. (b) The same site with the input field made secure by ShadowCrypt. Notice the border around the input field, and the lock to the right side of the input field.

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 exists as a chrome extension that runs before each web page is loaded. ShadowCrypt provides secure input by transforming each input text field into a secure input text field. The text fields are made secure by leveraging a browser primitive called ShadowDOM which can provide isolation between the DOM of the whole page, and input where the user enters his information. In other words, ShadowCrypt creates a barrier between the DOM of the web page and the input field that hosts the user's private information as he is typing. Furthermore, ShadowCrypt makes sure this boundary cannot be pierced by client side JavaScript. Therefore, the client side JavaScript cannot access the user's private information. An in depth description of ShadowDOM, the core technology empowering ShadowCrypt, and how it works will be provided in Section 5. In addition to modifying the HTML to provide this barrier, ShadowCrypt modifies the input field to highlight to the user that the input field has been made secure. Figure 2 demonstrates how ShadowCrypt transforms the user interface of input fields.

ShadowCrypt also uses the the DOM isolation provided by ShadowDOM 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 ShadowDOM that cannot be accessed by client side JavaScript. The content is then decrypted, and the plain text value is stored in the ShadowDOM and the encrypted value is left in the normal DOM. Therefore, the web page can only access the encrypted value, but the true 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 3 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 4. 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 a an aesthetic choice, it is also a means of authentication for the user.

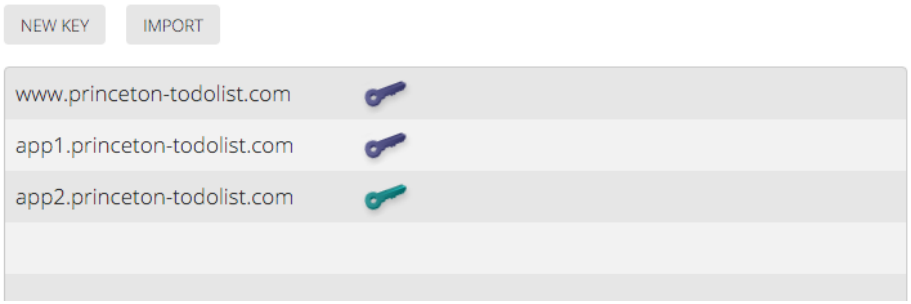


Figure 3: The ShadowCrypt options menu that lists all domains that are registered with ShadowCrypt.

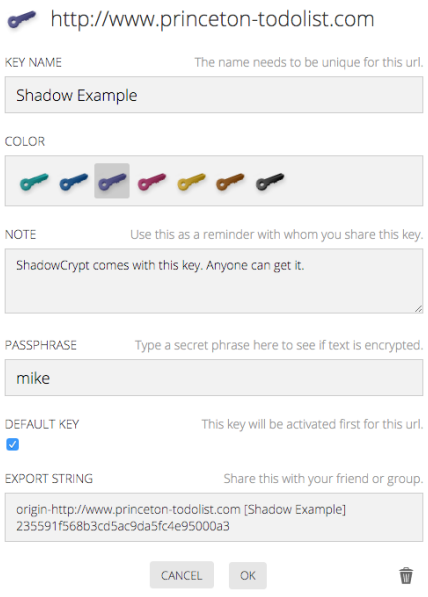


Figure 4: The ShadowCrypt options for each key. These configurable settings include the key name, key color, note, and passphrase.

3.2 Authentication

ShadowCrypt uses the key color and passphrase to authenticate the user. In most applications, the user must enter something he knows, such as a password in order to access the application. In this case, the authentication is reversed. The user sets a key color and passphrase, and the user then makes sure ShadowCrypt can reproduce these values correctly.

Therefore, with regards to key color, the color of the border and the lock must be the same color that the user sets for the domain. The color feature is important for making it difficult for an attacker to execute a clickjacking attack.

Similarly, the passphrase is a secret value that is only known by the user and ShadowCrypt. The passphrase is displayed to the user upon hovering over the lock, and upon hitting the the keyboard shortbut CTRL-‘, which brings up a pop up window as an alternate way of entering information. The pop up window along with the passphrase, which is set to "super secret passphrase", is provided in Figure 5.

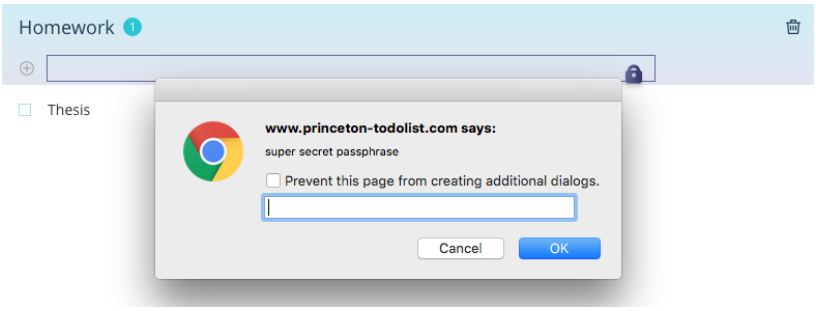


Figure 5: ShadowCrypt provides an alternate means of entering information into the input field through a pop up window. The pop up window includes the user's passphrase to prove to the user the pop window was produced by ShadowCrypt.

3.3 Click Jacking Attacks

The authentication mechanisms that are put into place by ShadowCrypt make it seem that it is attempting to prevent click jacking attacks; however, preventing click jacking attacks is not one of ShadowCrypt's original design goals. Shadow Crypt does not address this crucial aspect, and this paper demonstrates how important it is for click jacking attacks to be addressed. The original ShadowCrypt paper makes clear that it is not concerned with DoS attacks, side-channel attacks, and click jacking attacks. Ignoring DoS attacks and side-channel attacks, such as the length of the plaintext information, is very reasonable, as these types of attacks are unavoidable. However, ignoring click jacking attacks is a serious weakness to the design of ShadowCrypt. It is important to ensure the users of ShadowCrypt that a malicious web application cannot fool the user into exposing their private information, which should be protected by ShadowCrypt. This paper will highlight in

detail how ShadowCrypt is currently vulnerable to click jacking attacks. Through diving into the details of the click jacking attack, the 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

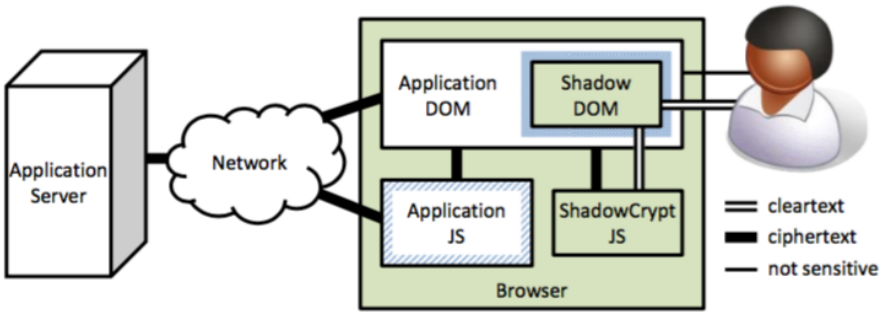


Figure 6: ShadowCrypt allows for the user's cleartext to be only available to the user, ShadowDOM, and ShadowCrypt. The Application DOM, Application JavaScript, and all server side code can only access the ciphertext. Image Source: [3].

Given an overview of the fundamentals of ShadowCrypt, Figure 6 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 ShadowDOM and the ciphertext will remain in the normal DOM. As Figure 6 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 ShadowDOM. The plaintext information is only available to the user, the ShadowDOM, and the ShadowCrypt JavaScript code.

4 Browser Primitives

While investigating ShadowCrypt's implementation I attempted many different attacks. While developing these attacks, I realized there are many critical browser primitives that ShadowCrypt relies upon in order

to develop their Chrome Extension. It is important to understand these primitives thoroughly to not only understand ShadowCrypt's implementation details, but also to understand what is currently available in browsers to create secure systems.

4.1 Isolated World

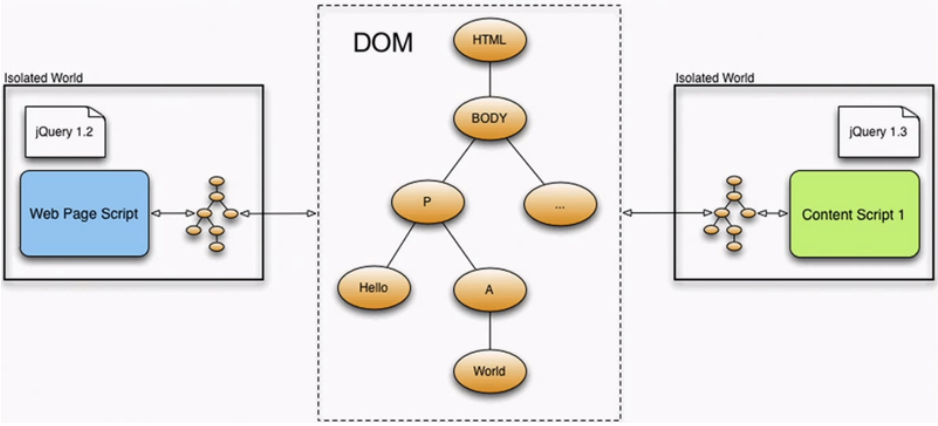


Figure 7: Isolated worlds allows each chrome extension and in-page JavaScript to execute in a different environment. Image Source: [10].

First of all, ShadowCrypt runs as a Chrome Extension; therefore, the environment in which Chrome Extensions execute is important. Google Chrome implements execution isolation between extensions and in-page JavaScript [10]. Therefore, in-page JavaScript does not have access to local variables and functions defined in the chrome extension and vice versa.

In addition to executing in an isolated environment, chrome extensions have access to privileged API's and privileged user data. For instance, in ShadowCrypt, the chrome extension has access to the Chrome API that retrieves all of the user's keys. The call used to get the user's key is provided below:

```
1    chrome.storage.sync.get(Startup.name, Startup.onGet);
```

This function takes as input parameters a String, which is used to index the storage database, and a callback function. Given this privileged access, it is important that in-page JavaScript cannot exploit any backdoor to gain access to this user data. In order to protect the extensions' privileged access, the in-page JavaScript and extensions interact with the DOM indirectly through a JavaScript object that represents the DOM, as demonstrated by Figure 7. This ensures the shared DOM does not provide a means for the in-page JavaScript to gain access to the privileged API's. Therefore, Chrome ensures extension developers that their applications will have an isolated execution environment with exclusive permission to specific user data.

4.2 Javasript Event Ordering

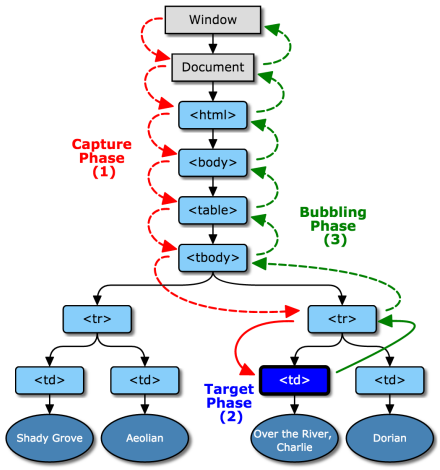


Figure 8: JavaScript events will be handled in a predefined order in three distinct phases. Beginning with `window` in capture phase, working down towards executing on the target in target phase, and ending on `window` in bubbling phase. Image Source: [11].

Events in JavaScript are critical to the implementation of ShadowCrypt. Event listeners must handle all key presses immediately in order to block the web application from gaining access to the keys pressed by the user. If the application could log all of the key presses, the application would gain access to the plaintext data. ShadowCrypt prevents the web application from accessing the key presses by registering events at the highest priority level. In JavaScript events are handled in three phases, as demonstrated in Figure 8. The first phase is the capture phase where all of the parent elements, beginning with the outermost parent can respond to an event that occurred on a target node. Therefore, events are handled first by a `window` event handler registered in capture mode. The next handler would be the document handler in capture mode, until it's the target's turn to run its own event handler. This first phase is the called the capture phase. When the target node for the event executes its own event handler, that is the second phase, called the target phase. Now the event propagates back up the DOM tree in order to execute any event handlers registered in the bubbling phase. Therefore, the final handler that could be executed would be an event registered on `window` in bubbling phase. An important note is that any handler can prevent the event from propagating along this predetermined path by calling `stopPropagation`.

ShadowCrypt registers events on all keypresses and inputs on `window` in capture mode. Therefore, ShadowCrypt will have access to respond to any event before a web application. Furthermore, since ShadowCrypt registers their events before the page has loaded, if a web application were to also register an event on window in capture mode, it would run second after ShadowCrypt's event has executed. The event registration is demonstrated below:

```
Widgets.init = function (win) {
    //add all event listeners in capture mode. therefore application cannot intercept them
    win.addEventListener('input', Widgets.onInputEarly, true);
    win.addEventListener('keydown', Widgets.onInterceptKey, true);
    win.addEventListener('keyup', Widgets.onInterceptKey, true);
    win.addEventListener('keypress', Widgets.onInterceptKey, true); };
```

Therefore, while originally I thought event handling would have been a difficult attack vector to prevent, the event registration in ShadowCrypt is done at the highest priority level, and the ShadowCrypt handlers prevent propagation, not allowing the web application to respond to the keypress.

4.3 Creating Events via JavaScript

Despite not being able to intercept events, I was also interested in creating events via JavaScript to simulate user actions. Creating custom events with JavaScript would allow me to simulate key strokes in a secure input field. Therefore, if I could render an insecure input field, and have the user type into the insecure input field, I could then simulate him typing into the secure input field. Then upon submitting the entry, I would have access to both the plaintext in the insecure input field, and the cipher text in the secure input field. This would allow the attack to execute as a man-in-the-middle attack, as the plaintext information would first be stolen, and then delivered to ShadowCrypt so that ShadowCrypt can execute as expected. Accessing the cipher text in addition to the plaintext is very important for making the attack stealthy. The details behind this portion of the attack will be provided in Section 6.

JavaScript provides an API for creating events where you can specify the event type and then dispatch the event from a specific DOM element. A snippet of code demonstrating this API is provided below:

```
targetNode = document.querySelector('#input-todo-new');
ev = new Event("input", { isTrusted: true }); //isTrusted is read only. setting isTrusted
    doesn't do anything
targetNode.dispatchEvent(ev);
```

Here the target DOM element is first selected, then the event is created using the standard event constructor, and finally the event is dispatched from the specific DOM element. As indicated by the comment, the constructor attempts to set the isTrusted event field, but this field is read-only. This field is very important because whenever an event is initiated by the user, isTrusted field is set to true, because the event is a real event initiated by the user. On the other hand, for all events created by JavaScript, the isTrusted field is set to false. This allows for systems to always check that field in order to ignore JavaScript based events. Upon checking the source code for ShadowCrypt, I noticed it is not validating that the event's isTrusted field is set to true.

Since ShadowCrypt is not sanitizing events based on the isTrusted field, it is possible to mock user events. I then attempted to create a keyboard event with a specific character. Upon investigating the API for creating keyboard events[12], I found that the charCode field associated with the event is also read-only. Therefore, for all keyboard events initiated by JavaScript, the charCode will default to 0, and I cannot simulate a specific type of keyboard event. I attempted to make a custom event[13] with charCode as a custom field, and then cast the custom event to a keyboard event, but this method also led to the charCode in the keyboard event to be 0. Therefore, even though I can dispatch an event via JavaScript, I cannot mock the necessary event fields.

It is clear the developers of the JavaScript API's were security conscious when setting which fields would be read-only. Without having access to creating keyboard events I had to use a different attack vector in order to gain access to both the plaintext and ciphertext. The attack vector that I ultimately exploited is described in Section 6.

4.4 Mutation Observer

Another attack vector involves manipulating the DOM after ShadowCrypt has initially executed. ShadowCrypt relies upon turning all input nodes into secure input nodes; however, if the application could not detect a change in the DOM after initial page load, a client could remove all secure nodes that ShadowCrypt created and replace them with insecure nodes. This attack is prevented by ShadowCrypt by relying upon the mutation observer primitive provided by browsers to detect changes to the DOM. Therefore, even if the client added an input node after initial page load, ShadowCrypt would be able to listen to this DOM change and also turn that new input node into a secure input node. The call used by ShadowCrypt to register the mutation observer on the entire document is provided below.

```
var Observer = {
    OPTIONS: {
        attributes: true,
        childList: true,
        characterData: true,
        subtree: true
    }
},
Observer.observer = new MutationObserver(Observer.callback);
Observer.observer.observe(doc.documentElement, Observer.OPTIONS);
```

ShadowCrypt specifies in its options to listen to changes to the attributes childList, characterData, and the subtree. This allows ShadowCrypt to listen to all DOM changes with a mutation observer.

ShadowCrypt relies heavily upon browser primitives such as isolated worlds, event ordering and the mutation observer. The most important browser primitive ShadowCrypt relies upon is the ShadowDOM, which will be discussed in detail in the following section.

5 Shadow DOM

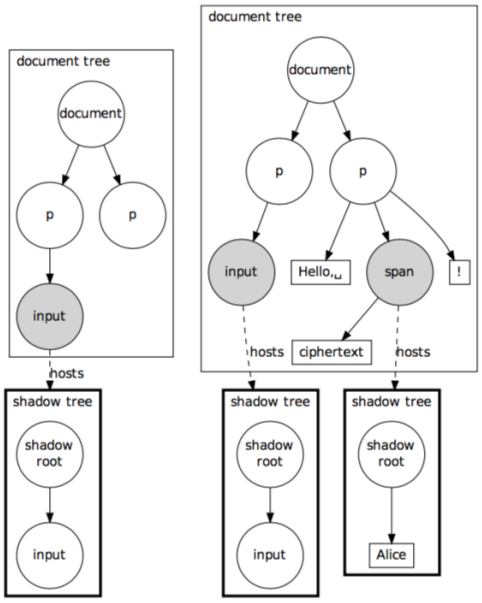


Figure 9: Shadow DOM creates encapsulated DOMs hosted by an element in the normal DOM. Image Source: [3].

This 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 [14]. In Figure 9, 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 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. This idea of hiding the plaintext in the shadow tree is demonstrated in the right half of Figure 9. 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.

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 = document.querySelector('button');
var root = host.createShadowRoot();
root.textContent = 'Hello from Shadow!';
</script>
```

The main API call is `createShadowRoot`, which is applied to a DOM element. The DOM element that calls `createShadowRoot` 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.

5.2 Multiple Shadow Roots

When the specification for Shadow DOM was first published and integrated into popular browsers, one could create multiple shadow roots underneath one shadow host. When a host had two shadow roots, it would render the content from the youngest shadow root. Here is a code snippet to demonstrate how this works:

```
<button>Hello, world!</button>
<script>
var host = document.querySelector('button');
var root1= host.createShadowRoot();
var root2 = host.createShadowRoot();
root1.textContent = 'Hello from Older Shadow!';
root2.textContent = 'Hello from Younger Shadow!';
</script>
```

This code will produce one button that says "Hello from Younger Shadow!".

However, using multiple Shadow Roots was deprecated as of April 2015[15]. Chrome currently supports multiple shadow roots, however a warning message is given to any user who uses multiple shadow roots. An example of this warning message is provided in Figure 10. Deprecating multiple shadow roots is important beyond preventing web developers from creating complex Shadow DOM structures. This will prevent web

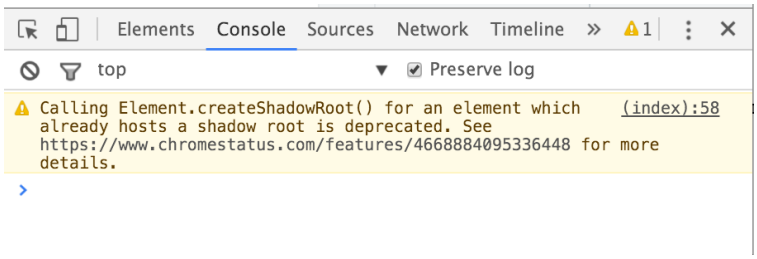


Figure 10: The warning message produced by the Google Chrome browser when a user creates multiple shadow roots underneath the same shadow host.

developers from creating a shadow root on any DOM element that has a shadow root by default. For instance, all input nodes have a user agent shadow root. Therefore, the following HTML

```
<input id="input-todo-new" type="text" placeholder="">
```

is transformed by the browser into:

```
<input id="input-todo-new" type="text" placeholder="Type to add new tasks">
  <!-- #shadow-root (user-agent)-->
  <div psuedo="-webkit-input-placeholder" id="placeholder" style="display: block !important;
     text-overflow: clip;">Type to add new tasks</div>
  <div id="inner-editor"></div>
</input>
```

Preventing the creation of a shadow root on input elements would fundamentally break the the implementation of ShadowCrypt, and it is clear based on the current specification that creating shadow roots on input elements will soon not be supported[16]. What this means for the future of ShadowCrypt will be addressed in Section 9. For now, we can assume that ShadowCrypt is still working based on the original specification of the Shadow DOM.

5.3 Shadow Piercing Selectors

In order to select elements inside of a shadow tree, you can use a a shadow piercing selector. For instance, the following code demonstrates two different ways to access all input nodes inside of the shadow tree that are hosted by the element whose id is equal to new-task.

```
document.querySelectorAll("#new-task::shadow input") \\option 1
document.querySelectorAll("#new-task /deep/ input") \\option 2
```

Shadow Piercing Selectors would similarly break ShadowCrypt fundamentally. These selectors would allow client side code to pierce into the shadow tree and access the user's plaintext information. These two particular shadow piercing elements have also been deprecated, but according to the December 2015 specification,

/deep/ will be replaced with > > > [17]. Therefore, even with the specification changing, piercing into the shadow trees will still be possible.

5.4 CSS Styling

At the publication of ShadowCrypt, Shadow DOM provided the ability for shadow roots to specify whether or not to inherit from the parent's style sheet. Each shadow root could set the field `applyAuthorStyles` to true or false. In ShadowCrypt, the secure input nodes set this field to false in order to prevent inheriting styles from the parent site. For secure output, the field is set to true in order to minimize the change to the look and feel of each website. Utilizing `applyAuthorStyles` has similarly been deprecated. For instance, if the parent site is using bootstrap to style each , the elements used to render the secure output will no longer be styled according to bootstrap.

Styling is more important than just the the usability of the site. Accessing style elements and changing style elements of a secure system can lead to a user interface attack. It is important for the specific styling of the secure elements to not be accessible by the web application, and the styling must be immutable. Furthermore, it must be hard to mimic the style of the secure elements in order to prevent a user interface attack.

5.5 Insertion Points

Even though the content of the shadow host is not rendered by default inside of the shadow tree, it is possible to insert the host's content into the shadow tree. Using the `<content>` insertion point it is possible to pull the content from the host into the shadow tree. This is very convenient for web developers as it allows for the shadow tree to define the presentation of the information, and the host only has to provide the content. More information can be found at [14].

In addition to the `<content>` insertion point, there is a `<shadow>` insertion point, which pulls information from older shadow roots. Therefore, if an element has multiple shadow roots, the youngest shadow root can include `<shadow>` insertion points to grab content from older shadow roots.

Here is an example taken and modified from a Shadow DOM blog [14] demonstrating how this works:

```
<div id="example1">Host DOM</div>
<script>
  var container = document.querySelector('#example1');
  var root1 = container.createShadowRoot();
  var root2 = container.createShadowRoot();
  root1.innerHTML = "<div>Root 1 Wins</div><content></content>";
```

```
  root2.innerHTML = "<div>Root 2 Wins</div><shadow></shadow>";
</script>
```

This piece of code will render:

Root 2 Wins

Root 1 Wins

Host DOM

The youngest shadow root is rendered first, followed by the content pulled-in from the older shadow root through the usage of `<shadow>`. The older shadow root renders its content followed by the hosts content, which is pulled-in using `<content>`. A live example of demonstrating how insertion points work can be found at https://jsfiddle.net/sgrybcqg/.

5.6 Event Model

Events that occur within internal elements of a shadow tree are often retargeted so that they look like they came from the host element rather than an element inside of the shadow tree [18]. This is relevant to ShadowCrypt because this allows for certain events to be mocked without requiring access to the elements created by ShadowCrypt and placed inside of a shadow tree. For example, here is what an input field becomes when it is transformed into a secure input field by ShadowCrypt.

```
1 <input id="input-todo-new" type="text" placeholder="Type to add new tasks">
2     <!-- #shadow-root (open)-->
3     <div class="wrapper locked-color-2">
4         <input class="delegate">
5     </div>
6 </input>
```

So when a user is interacting with a secure input field, the input events that occur when the user is typing are occurring inside of the input field with class "delegate" (line 4); however, these events are retargeted so that they appear to be coming from the input with id = "input-todo-new" (line 1). An example demonstrating event retargeting is available at https://jsfiddle.net/c83sjkxc/.

6 User Interface Attack

While exploring the attack surface against ShadowCrypt I found a way to successfully implement a user interface attack. This attack assumes the Shadow DOM boundary that ShadowCrypt attempts to create is

fully working. Therefore, the client side code cannot access any information from elements that are internal to the shadow tress created by ShadowCrypt. While the original threat model of ShadowCrypt admits it is vulnerable to user interface attacks, it is important to understand the attack vectors that go into executing a user interface attack. Understanding these attack vectors is a big step towards creating a system that is robust against user interface attacks, which is a critical component of a secure I/O system.

6.1 Rendering Insecure Nodes

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

The first method focuses on how ShadowCrypt selects which input nodes to make secure. The code used by ShadowCrypt to check if an element must be transformed into a secure input node is provided below:

```
Widgets.createAdapter = function (node, rule) {
  var tag = node.tagName.toLowerCase();
  if (tag === 'input') {
    if (node.type === 'text') {
      return new Widgets.adapters.Input(node, rule);
    }
  } else if (tag === 'textarea') {
    return new Widgets.adapters.TextArea(node, rule);
  } else if (tag === 'iframe') {
    return new Widgets.adapters.IFrame(node, rule);
  } else {
    if (node.contentEditable === 'true') {
      return new Widgets.adapters.ContentEditable(node, rule);
    }
  }
  return null;
};
```

This function indicates that ShadowCrypt is protecting against inputs with type text, 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 [19]. 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 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 get an insecure input field on the page.

Rather than relying upon these different input types, I also found a DOM manipulation that ShadowCrypt is not currently listening to, which can lead to an input node being left insecure. 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>
```

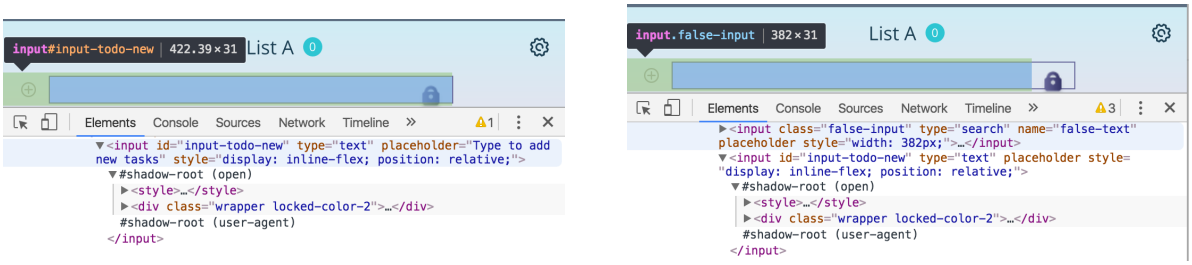
ShadowCrypt will transform the input with class `false-input` (line 3) into a secure input node, and ignore the input with class `main-input` (line 2) because it is of type "checkbox". Now, once the page finishes rendering, the user interface attack can run a very simple line of JavaScript:

```
1 Template.body.rendered = function() {
2   // select the target node
3   var target = document.getElementsByClassName('main-input')[0];
4   target.type = "text";
5 };
```

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 insecure.

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 insecure input field on the page.

These above methods can lead to select input nodes being left insecure; 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



(a) This site is not being attacked.

(b) This site is being attacked.

Figure 11: Input fields with and without an insecure input rendered on top of it. Each input field looks the same, so the positioning aspect of the attack can be achieved. a) The secure input field has been transformed by ShadowCrypt to host a shadow tree which will store the user's plaintext information. b) The attack includes an input field with type search so it is insecure. It is positioned directly on top of the secure input field. The insecure input field is slightly less wide in order to not conflict with the functionality of the lock at the end of the secure input.

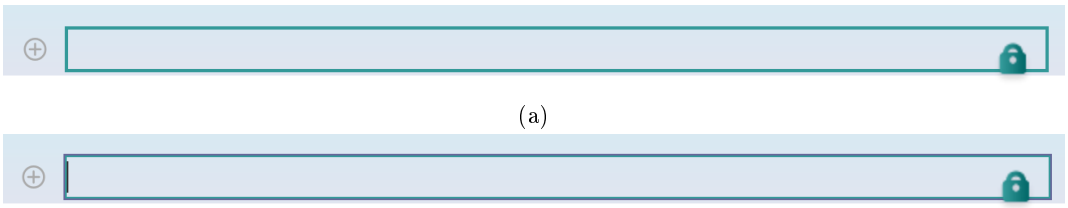
complete, it must be more stealthy. The following aspects of the attack focus on the stealthiness of the attack.

6.2 Positioning

In order to make the user feel that he is interacting with a secure input node, the insecure 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. Therefore, even though there is an insecure input node being rendered directly on top of the secure input node, it will still look exactly like a standard secure input field. Figure 11 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 I had to set the CSS properties of the insecure input field 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



(a)

(b)

Figure 12: (a) A 2px green border that has been rendered by the ShadowCrypt code working correctly on a site that is not attacked. (b) 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.

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.

6.3 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 I just apply a border with a fixed color around the insecure input field. In order to apply this border I 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 I 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, I 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. Figure 12 demonstrates 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 had a hard time noticing the extra thickness was a fixed purple color. A further discussion on how to best include UI markers in a secure input/output system will be explored in Section 9.

6.4 Keyboard Shortcuts

ShadowCrypt comes with two keyboard shortcuts. CTLR-‘ 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. I 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 I mocked the thicker border, I 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. Similarly, CTRL-SPACE changes the state of the lock into a grayed out lock and the default border is removed. I cannot change the internal state of the shadow tree, so this is impossible. I 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. However, as discussed in Section 4, 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, I 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, I set the value of the secure input host node to be the value that is currently in the insecure input node. I 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. I then simulate hitting TAB so that the secure input field is focused, and I 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, I simulate hitting SHIFT-TAB, which will return the focus to the insecure input field.

6.5 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, I 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 I store both the plaintext and the ciphertext in the database. Then I only return the ciphertext to the client so that ShadowCrypt secure output works as expected and I 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 I gain access to the plaintext information, and then allow for ShadowCrypt to work as expected.

When the user hits submit I have direct access to the plaintext because he was typing into an insecure input field. In order to get the associated ciphertext I trigger the ShadowCrypt encryption scheme, and then the ciphertext is available in the shadow host. In order to trigger the ShadowCrypt encryption I set shadow host's value to be the plaintext value. I 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. I 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, I first check to see if the secure input field is locked. If the secure input field is locked, I set the metadata field `stolen` to be true on the plaintext value. If the input is not locked, I set the metadata field `stolen` to be false, so that the entry is returned to the client in plaintext form as expected. In order to check if the input is locked, I 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 I know that the input is not locked.

All in all, in order to mock secure output I 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 I administered a user study on Amazon Mechanical Turk. The design of the study will be covered in

Section 7 and the results will be covered in Section 8.

7 User Study Design

The user study was designed to determine the usability of ShadowCrypt as well as the stealthiness of the user interface attack. The first step in designing the study was to design a sample to-do list application which would be used as the foundation for the user interface attack. Once the sample application was complete, I implemented the attack on the sample application. In order to measure the effectiveness of the attack, I had one site that was not attacked and another site that was attacked. I then asked the participants if they could notice which one was attacked.

7.1 Sample Application

The sample application was strongly based on the one of the sample Meteor applications[20]. This application allows for users to create an account, and then once logged in to create to-do lists, and add various tasks to each list. The sample application was modified slightly so that users were forced to login and could not create public lists/tasks. Furthermore, originally the login required email and password, but to keep the participants email private, I set the login to be based on the user's mechanical turk id rather than email. The application is hosted using Digital Ocean[21] and is backed by a MongoDB[22] database hosted with MongoLab[23]. An example of the application can be found at princeton-todolist.com. The applications that are used for the main test in the study are hosted at app1.princeton-todolist.com and app2.princeton-todolist.com. The attacked version is located at app2.princeton-todolist.com.

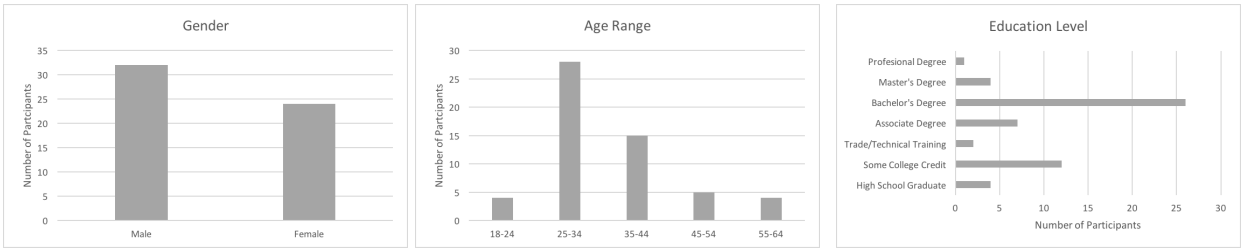
7.2 Survey Design

7.2.1 Demographic Information

I first ask a few demographic questions to make sure the participants cover a wide range of internet users. I gather the participants' gender, age range, and education level. Here are some plots demonstrating the demographics of the participants:

7.2.2 Joining the sample application

In the next section of the survey the participant joins the sample application at princeton-todlist.com. The participant registers within the application with his Amazon Mechanical Turk ID and a password. Upon



(a) (b) (c)

Figure 13: Demographic information of the participants. a) Gender b) Age Range c) Education Level

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.

7.2.3 User Experience

In order to measure the user experience of ShadowCrypt, I first measure the user experience of the to-do list application without ShadowCrypt. I ask the participant five user experience questions. The questions are in the form of statements that the participant agrees or disagrees with on a scale from 1-5. I ask the participant if he found the application fun. I then ask the participant if he found that adding a list/task was tedious/difficult.

7.2.4 Installing ShadowCrypt

The next section walks the participant through installing ShadowCrypt. The user installs a slightly modified version of ShadowCrypt. I modified the original source code in order for ShadowCrypt to be on by default on all of the sites that are relevant to the study. In order for the participant to understand how to use ShadowCrypt, and why it is useful, I created a tutorial video. I made sure to cover every aspect of ShadowCrypt in order to introduce as little bias as possible. The tutorial video can be found at https://mdf3.wistia.com/medias/2x2lirbeyg. Based on the data provided by the video hosting platform, many of the participants watched the video completely. Furthermore, at the very end of the survey, after affirming the participant that they would be paid, I ask for an honest answer to whether or not they watched the full video. Out of the 65 participants, only 1 participant claimed to have not watched the video completely.

7.2.5 Using ShadowCrypt

With ShadowCrypt installed and the tutorial complete, the participant creates a few lists and tasks with ShadowCrypt enabled. I also ask the user a few questions to verify ShadowCrypt is working correctly. I

make sure that there is a lock next to the input field, and that upon hover the lock displays the key name and passphrase. I also ask the user to verify the border color is purple.

7.2.6 ShadowCrypt User Experience

I ask the user the same user experience questions as before, but now with ShadowCrypt enabled in order to measure how ShadowCrypt affects the user experience. I also ask the user if he would recommend ShadowCrypt to a friend, and a more direct question regarding how much ShadowCrypt hampered the user experience, if at all.

7.2.7 ShadowCrypt Options

This section walks the user through using the ShadowCrypt options menu. This section is very important for the final question because it is important for the user to understand how to change the key color and passphrase in order to test if the application is compromised. This section walks the user through navigating to the options menu and updating the key color and passphrase.

7.2.8 ShadowCrypt Authentication

This section walks the user very explicitly through the aspects of ShadowCrypt that must be checked in order to make sure ShadowCrypt is working correctly. Therefore, this section walks the user through creating a new task using the CTRL-‘ keyboard shortcut. There is an explanation on passphrase, and it discusses how to hover over the lock in order to display the passphrase. Lastly, this section explains the CTRL-SPACE keyboard shortcut, which can be used to toggle the input field to unlocked.

7.2.9 ShadowCrypt Under Attack Part 1

I tell the user that he is about to go to 2 different versions of the application. I explain that 0, 1 or 2 of the applications are compromised. I 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 app1.princeton-todolist.com, and 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 application, the user reports whether or not he believes the application was compromised, along with an explanation of his answer.

In this study Application 1 was also not compromised, and Application 2 was always compromised. Ideally it would have been randomized; however, it was difficult to keep all of the data in sync in the case of randomization. While the approach I chose to use was simpler, it possibly added a bias where individuals would just guess that Application 2 was compromised because it's second.

7.2.10 ShadowCrypt Under Attack Part 2

This section is the same as the previous section, except it now directs the user to app2.princeton-todolist.com.

7.3 Amazon Mechanical Turk

The participants were recruited through Mechanical Turk. I limited the participants to Mechanical Turk Masters that are located in the US. I also make sure the turkers' approval rates are above 95% and that they have had 100 approved HITs. Applying these restrictions allowed for the majority of my participants to successfully complete the survey. The survey took around 30 minutes to complete, and I paid the participants $3.50.

7.4 IRB

This study was determined to not be human subjects research as defined by DHHS regulations. An official form claiming this study is not human subjects research can be provided upon request.

8 User Study Results

Before releasing this study to a wide audience, I administered a pilot study to 25 people on Amazon Mechanical Turk. I also had 2 peers and 2 advisers complete the study. Based on the feedback from this pilot study I fixed a CSS bug that caused the attack to be noticeable on smaller screens. Furthermore, I was able to make the user study easier to follow and the video tutorial was edited in order to give a fuller explanation of the benefits of ShadowCrypt.

With the attack and user study polished, I released the user study to 65 people through Amazon Mechanical Turk. Many of the participants were able to compete the study; however, some of the users' data had to be deleted for various reasons:

- 4 people did not actually interact with the sample sites. I went to the backend database and found certain participants claimed to have completed the study, but did not actually create any tasks or lists on the sample to-do list applications.
- 1 person failed attention checker questions.
- 3 people could not figure out how to use the keyboard shortcuts.

Result	Percentage of Participants
Neither Application 1 nor Application 2 are compromised	73.2%
Application 1 or Application 2 is compromised due to key management confusion.	21.4%
Application 2 is compromised.	**5.4%**

Table 1: The results demonstrate the majority of users did not notice any attack on Application 2. Many users were confused by key management and expected changes they made to certain keys on one site to affect their key properties on a different site. Most important, only 5.4% of the participants were able to identify the user interface attack, demonstrating the effectiveness of the attack.

- 2 people failed to install ShadowCrypt correctly. Upon installation there was no lock on the right side of the input, which indicates ShadowCrypt is working.

After removing the participants who failed to complete the entire user study, there were 56 people left. The results of these participants are summarized in Table 1.

The results indicate that it was difficult for the users to notice the user interface attack. The middle row of Table 1 represents the users who, based off of their comments, were confused by the key management aspect of ShadowCrypt. The users were asked to change the key color and passphrase on the sample application, but then when they came to Application 1 or 2, they expected these edits to carry over. However, since the sample application, Application 1, and Application 2 were on different domains, the edits to the key configuration for princeton-todlist.com did not change the key configuration for app1.princeton-todolist.com. It's possible that explaining key management better during the user study would have decreased the number of participants who were confused by key management.

The most important take away from the user study results is that only 5.4% of the participants 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. 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 clickjacking attack might be more effective at preventing users from being fooled by a clickjacking attack. A deeper discussion on how to prevent clickjacking attacks will be presented in Section 9.3.

Also, in the attack I turned the attack off whenever the user hit the CTRL key because I could not mock

Statement	Average Rating Without ShadowCrypt	Average Rating With ShadowCrypt
Using the to-do list application was fun.	3.54 (σ =.98)	3.5 (σ =.96)
Adding a task on the to-do list application was difficult.	1.32 (σ =.76)	1.34 (σ =.76)
Adding a task on the to-do list application was tedious.	1.59 (σ =.94)	1.5 (σ =.89)
Adding a list on the to-do list application was tedious.	1.68 (σ =.97)	1.55 (σ =.92)

Table 2: Table comparing the user experience results of the application with and without ShadowCrypt. The averages demonstrate ShadowCrypt does not negatively impact the user experience.

Statement	Average Rating
You believe ShadowCrypt made the to-do list application less user friendly.	1.8 (σ =1.03)
You would recommend ShadowCrypt to a friend.	3.5 (σ =1)

Table 3

the keyboard shortcuts. My assumption was that users would not use the keyboard shortcuts very often, and therefore it would not be a significant amount of tasks that were fully encrypted. Based on the 56 people who completed the survey correctly, on the attacked version of the application there were 531 tasks created. Of these tasks, I had access to the plaintext information for 484 and 47 were full encrypted. Therefore, the attack gained access to 91% of the tasks, and was identified by only 5.4% of participants.

8.1 Usability

Despite the user interface attack being highly effective, which demonstrates serious privacy vulnerabilities, the user study demonstrated ShadowCrypt to be very user friendly. 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. The users were asked to rate their agreement on a scale of 1-5 for the statements in Table 2 (1 is strongly disagree, 5 is strongly agree). In addition to the questions intended for comparison, I asked two direct questions to determine the usability of ShadowCrypt. These questions are provided in Table 3. Without a significant number of participants, I cannot make any strong claims on the usability of ShadowCrypt. However, it seems that ShadowCrypt does not negatively impact the user experience of the site. This is a very important part of the secure I/O system. If the system is not user friendly, there will not be strong adoption. In the following section I will further discuss the fundamentals behind the secure I/O system, how ShadowCrypt fails to address these fundamental issues, and what to do moving forward.

9 Discussion

9.1 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.

9.1.1 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 [24]. 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`.

9.1.2 Shadow piercing selectors

Accessing the `shadowRoot` property directly is not the only way to gain access to the information inside of the shadow tree. As discussed earlier there are shadow piercing selectors that can be used to access Shadow DOM elements. The following code snippet can be used to access the plaintext information typed into an input field made secure by ShadowCrypt:

```
1 document.querySelector('#input-todo-new::shadow .delegate').value;
```

This statement will select the input node with the id "input-todo-new", which is the shadow host. Then

"::shadow" is used to pierce into the Shadow DOM, and the ".delegate" selects the input field inside of the Shadow DOM. This specific shadow piercing selector is deprecated and will stop working soon in Chrome, but there is another shadow piercing selector in the updated specification [17].

9.1.3 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. 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 = document.querySelector('#input-todo-new').createShadowRoot();
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.

The deprecation of multiple shadow roots 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. 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 click jacking attacks.

9.2 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` will encrypt the plaintext information, and send a message to the parent window. The browser extension will 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 for 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.

9.3 Clickjacking 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 clickjacking attacks. Clickjacking attacks occur because sensitive UI elements are presented *out of context* [25]. 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 clickjacking attacks occur by tricking a user into believing they are interacting with one thing, but really they are interacting with something completely different. InContext, the clickjacking defense 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 clickjacking 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 clickjacking 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 clickjacking 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 clickjacking 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.

Given the different nature of this attack, either the sensitive element is forced to scan the page for elements that are being overlaid or the sensitive element must have clear indicators when the user is interacting with the sensitive element. When these indicators are there, the user knows 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 to prevent clickjacking.

9.3.1 Iron Frame

Iron Frame was presented at DefCon in 2015, and will hopefully be deployed into browsers soon [26]. 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 clickjacking 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 [27]. 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.

9.3.2 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 feature in order to witness the attack. Relying upon the user to notice the clickjacking 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 clickjacking 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.

10 Conclusion

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 clickjacking 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 I hope to see deployed into browsers soon.

References

[1] Hayley Williams Uber Vs. Taxis: Choice Says UberX Is Cheaper And Safer. http://www.gizmodo.com.au/2015/09/uber-vs-taxis-choice-says-uberx-is-cheaper-and-safer/

[2] Kia Kokalitcheva Uber To Settle With N.Y. Attorney General Over 'God View' Privacy Breach. http://www.fortune.com/2016/01/06/uber-new-york-settlement/

[3] 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*, pp. 1028–1039.

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

[5] Welcome to Python.org. https://www.python.org/

[6] The Go Programming Language. https://golang.org/

[7] PHP: Hypertext Preprocessor. https://secure.php.net/

[8] Popa, R. A., Redfield, C., Zeldovich, N., & Balakrishnan, H. 2011. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 85–100.

[9] Popa, R. A., Stark, E., Valdez, S., Helfer, J., Zeldovich, N., & Balakrishnan, H. Securing web applications by blindfolding the server. NDSI (2014).

[10] Content Scripts - Google Chrome https://developer.chrome.com/extensions/content_scripts

[11] UI Events Specification https://www.w3.org/TR/DOM-Level-3-Events/

[12] KeyboardEvent https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent

[13] CustomEvent https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent

[14] Dominic Cooney. Shadow DOM 101 http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/

[15] Webapps/WebComponentsApril2015Meeting https://www.w3.org/wiki/Webapps/WebComponentsApril2015Meeting

[16] Shadow DOM W3C Working Draft 15 December 2015: Methods https://www.w3.org/TR/2015/WD-shadow-dom-20151215/#methods-1

[17] Shadow DOM W3C Working Draft 15 December 2015: Composed Trees https://www.w3.org/TR/2015/WD-shadow-dom-20151215/#composed-trees

[18] Shadow DOM 301 http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-301

[19] HTML5 Input Types http://www.w3schools.com/html/html_form_input_types.asp

[20] Todos | Build a collaborative task app with Meteor https://www.meteor.com/todos

[21] Digital Ocean https://www.digitalocean.com/

[22] MongoDB https://www.mongodb.org/

[23] mLab https://mlab.com/

[24] DOM Attributes now on the prototype chain https://developers.google.com/web/updates/2015/04/DOM-attributes-now-on-the-prototype-chain?hl=en

[25] Huang, L.-S., Moshchuk, A., Wang, H. J., Schechter, S., and Jackson, C. Clickjacking: Attacks and Defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security '12, USENIX Association.

[26] Dan Kaminsky. I Want These * Bugs off My * Internet. DEF CON 23. https://www.youtube.com/watch?v=9wx2TnaRSGs

[27] Dan Kaminsky. I Want These * Bugs off My * Internet. DEF CON 23. Slide 71-72. http://www.slideshare.net/dakami/i-want-these-bugs-off-my-internet-51423044