Click This, Not That: Extending Web Authentication with Deception

Timothy Barron
Yale University
New Haven, CT, USA
timothy.barron@yale.edu

Johnny So
Stony Brook University
Stony Brook, NY, USA
josso@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
Stony Brook, NY, USA
nick@cs.stonybrook.edu

ABSTRACT

With phishing attacks, password breaches, and brute-force login attacks presenting constant threats, it is clear that passwords alone are inadequate for protecting the web applications entrusted with our personal data. Instead, web applications should practice defense in depth and give users multiple ways to secure their accounts.

In this paper we propose login rituals, which define actions that a user must take to authenticate, and web tripwires, which define actions that a user must *not* take to remain authenticated. These actions outline expected behavior of users familiar with their individual setups on applications they use often. We show how we can detect and prevent intrusions from web attackers lacking this familiarity with their victim's behavior. We design a modular and application-agnostic system that incorporates these two mechanisms, allowing us to add an additional layer of deception-based security to existing web applications without modifying the applications themselves.

Next to testing our system and evaluating its performance when applied to five popular open-source web applications, we demonstrate the promising nature of these mechanisms through a user study. Specifically, we evaluate the detection rate of tripwires against simulated attackers, 88% of whom clicked on at least one tripwire. We also observe web users' creation of personalized login rituals and evaluate the practicality and memorability of these rituals over time. Out of 39 user-created rituals, all of them are unique and 79% of users were able to reproduce their rituals even a week after creation.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Intrusion detection systems*; *Multi-factor authentication.*

KEYWORDS

Intrusion detection, Deception, Multi-factor authentication

ACM Reference Format:
Timothy Barron, Johnny So, and Nick Nikiforakis. 2021. Click This, Not That: Extending Web Authentication with Deception. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Virtual Event, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3433210.3453088

ASIA CCS '21, June 7–11, 2021, Virtual Event, Hong Kong.

ACM ISBN 978-1-4503-8287-8/21/06…$15.00
https://doi.org/10.1145/3433210.3453088

1 INTRODUCTION

Many applications which are used on a daily basis—such as email, online editors, social media, and file sharing—are located on the web. By using them, we entrust these applications with our highly personalized accounts that often contain sensitive information. As such, these applications are attractive targets to malicious web users. The primary (and sometimes only) defense used by web applications are passwords, but they are often insufficient. Part of the problem is that even if users create strong passwords, prior work has shown that these passwords are often re-used across many sites [11]. Accounts can be compromised through brute-force (repeatedly guessing passwords) or credential-stuffing (trying known credentials from one site on other sites) [16]. In fact, researchers have spent decades trying to improve upon or replace passwords [7], but there has so far been no alternative that enjoyed widespread adoption.

Some web applications opt to address the limitations of passwords by supplementing them with multi-factor authentication (MFA). Password-based authentication schemes rely on only one factor of authentication: they challenge the client to provide something that they would know if they were the real user. Multi-factor authentication schemes attempt to challenge the client with combinations of the following: what the real user would know (e.g. password), what the real user would have (e.g. phone), and what the real user would be (e.g. fingerprint). A common implementation is to ask a user to verify their identity by entering a one-time code sent via SMS to their registered phone number.

The downside to MFA is that many implementations add an inconvenient burden when logging in, making it unpopular. In 2015, Petsas et al. found that only 6.4% of Google users had enabled two-factor authentication [22]. Even for the users who do rely on MFA, online services are aware of the friction MFA adds to the user experience and thus try to use it as little as possible, e.g., when changing a password or when logging in from a new location.

In this paper, we present two methods for extending authentication on web applications beyond passwords. We propose *web tripwires* and *login rituals*, two complementary ideas that take advantage of users' familiarity with the applications they use often. One defines what a user *should* do, while the other defines what they *should not* do, in order to remain trusted by an application.

Web tripwires are deceptive intrusion detection mechanisms similar to honeytokens [31] that are added into an application. They are specific to each user allowing them to customize their own intrusion detection traps. Contrastingly, login rituals are specific

sets of user actions that must be taken immediately after initial authentication. These are intended to be used in addition to passwords, but one advantage is that they are specific to an application so they cannot be re-used across web applications. Compared to MFA,these can be active at all times with less disruption to normal use because they align with how a user interacts with the application already. Moreover, our proposed mechanisms can be deployed in parallel with other MFA and be used to further reduce friction between users and MFA systems, e.g., prompt users for extra codes and tokens only when they trigger a web tripwire or violate a login ritual.

We summarize the contributions of this paper as follows.

- We propose web tripwires and login rituals for protecting web applications against unauthorized access.
- We implement a system demonstrating how these mechanisms can be realized in practice. We show that they can be deployed on top of a variety of web applications by testing with five popular open-source applications: WordPress, phpMyAdmin, Roundcube, ownCloud, and ShareLaTeX.
- We evaluate our system with real web users, demonstrating the strength of tripwires as a defense, with up to an 88% detection rate, and the ability of users to understand and remember login rituals over time, with rituals successfully completed in 74% of the time.

2 DESIGN

This section will describe the two proposed mechanisms and the design goals. There are a few goals which apply to both mechanisms. First, we are primarily concerned with protecting logged-in users. This means we are not concerned with public parts of applications and our threat model assumes that an attacker has either hijacked a session or guessed/stolen the login credentials, and was able to complete multi-factor authentication if there was any. Two important overarching goals for both mechanisms is that we want our solution to be generally applicable to most if not all web applications and we want to be able to deploy it on top of existing applications with minimal configuration and without the web server having any knowledge of our system.

2.1 Web tripwires

The idea behind web tripwires is that complicated web applications such as Gmail or Facebook provide custom experiences to different users. These users become very familiar with their own personal view, but an intruder who has broken into an account does not have that same familiarity. Therefore, a user can set up tripwire elements that only they know about. A tripwire is a trap that can be avoided by the legitimate owner of an account, but can trigger countermeasures when activated (red elements in Figure 1). If a tripwire access is detected, we assume it is due to an intrusion since the legitimate user knows to avoid it. False positives are still possible if the real user forgets or mis-clicks. To handle these cases, we want to support per user policies that trigger countermeasures if the number of tripwires accesses within a recent time window exceeds a certain threshold. These countermeasures include logout, banning an IP address, or disabling an entire account. Each of these countermeasures needs to be enforced by our system without intervention from the application server itself.

Figure 1: Abstract example of a web page with ritual elements (green) and tripwire elements (red). Ritual elements can be chained together to create login rituals that must be performed immediately after logging in. Tripwire elements must be avoided at all times.

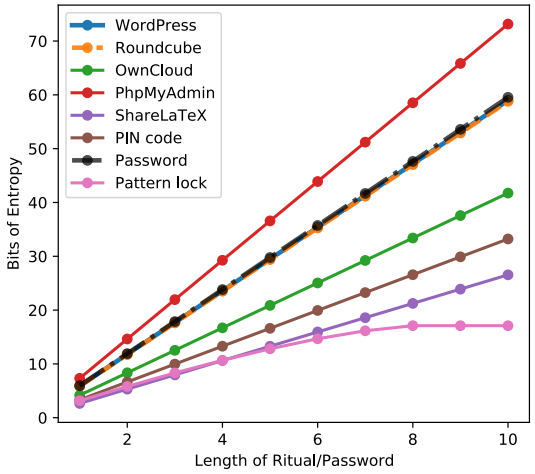

Figure 2: Comparison of rituals on different applications to alphanumeric passwords, n-digit PIN codes, and 3x3 pattern locks. Bits of entropy are calculated as $\log_2(L)$ where L is the number of possibilities for length n.

We define multiple types of tripwires to address different attack scenarios. The first is an *injected* tripwire. This is a new fake element which is added into the web page by our system. The user can create it to make it look like part of the real application and appear attractive to attackers. The goal is that these injected tripwires should be difficult to identify for anyone except the legitimate owner of an account. We anticipate this type of tripwire being most effective against human attackers who are manually exploring a compromised account.

The next type is an *existing* tripwire. This is a real part of the application which a user rarely (if ever) uses. This may be because

the functionality is duplicated elsewhere, or because it corresponds to a feature that the user does not need. These tripwires are inherently indistinguishable from the rest of the application because no changes are made on the client side when an existing element is made into a tripwire. This type of tripwire is effective against bots and human intruders that have a specific set of actions they plan to take. Transforming rarely used parts of the application into tripwires is similar to the idea of web debloating in which unused parts of applications are removed to reduce attack surface [5]. In our case, the elements remain to be used for intrusion detection.

2.2 Login rituals

A login ritual is an additional authentication requirement immediately following a standard password login. The idea is that after users log in to a web application, they will then perform a prescribed series of actions to confirm that they are the real owner of the account (green elements in Figure 1). This is conceptually similar to entering a second password, but with a number of concrete advantages. The first is that a login ritual is specific to a particular web application and cannot be re-used across multiple sites, thereby organically addressing the problem of password reuse. For example, a user could not reuse their banking login ritual on their social-media account, not because the two sites explicitly forbid them, but simply because they do not share common UIs. The second advantage is based on the assumption that many users have individual habits after logging in to websites they use frequently. This makes rituals easier to remember and convenient since a ritual can be a sequence of actions the user was going to do anyway. Users can create their own rituals, and an intruder is logged out (and potentially blocked at the network level, after repeated ritual failures) if they do not immediately complete the ritual sequence.

The number of possible rituals that a user can create will determine how difficult it is for an attacker to guess the correct sequence of actions. The number of possibilities depends on the application, with more complicated user interfaces providing more options from which to build rituals. We use an analogy to passwords where one chooses a sequence of n alphanumeric symbols. In this case there are 62 symbols to choose from (or 95 with special characters) to create one of 62^n potential passwords of length n. A less secure, but still commonly used scheme is 4-digit PIN codes, in which case there are only 10 symbols leading to 10,000 possibilities. We also compare to 3x3 pattern locks, popularized by Android, which provide up to 389,112 possibilities at length 8 [33]. In the case of rituals, the symbols available to choose from are the interact-able elements within an application. To evaluate the number of possible rituals, we crawled the pages of our evaluated web applications (WordPress, Roundcube, OwnCloud, PhpMyAdmin, and ShareLaTeX) counting the number of links to estimate the number of symbols available. Unlike passwords, the number of symbols varies depending on which page the user is currently viewing. We chose to take the average number of links found across all crawled pages within each application and call that its number of symbols. We then estimate the number of possible rituals to be s^n where s is this average number of symbols and n is the length of the ritual. Note that this is likely to be an underestimate since we only count <a> tags, whereas in practice, multiple HTML elements can have click-related event handlers, making them candidates for use in a ritual.

Figure 2 shows how many bits of entropy are available for rituals in each application and compares to three other types of passwords. We see that rituals are capable of providing similar levels of entropy to these other techniques. PhpMyAdmin provides more options than alphanumeric passwords and ShareLaTeX, which has fewer UI elements, is similar to pattern locks until the pattern reaches its maximum length of 8 and falls behind. Any of these may be sufficient if we assume the user makes good random passwords and the attacker makes random guesses. In each case, when the length is 5 or more, the probability of guessing the code/password/ritual within 5 attempts is less than 0.1%. As long as the number of failed attempts is capped or rate limited, guessing attacks are very unlikely to be effective. Rather, the primary concern with passwords and PIN codes is re-use of common passwords and re-use across applications. As we mentioned above, this is where rituals have the advantage. Rituals are application-specific, and if the user chooses elements that they created, such as an email folder, then they may also be entirely user-specific and thus cannot be re-used. Since login rituals provide offer similar protection against guessing attacks, while avoiding the pitfalls of passwords, we have shown that they contribute more to a user's security than having a second password.

When used together, login rituals and web tripwires describe the expected behavior from a real user by outlining actions that they *must do* and those that they *must not do*. An intruder will fall outside this expected behavior when their unfamiliarity with the real user's specific environment leads them to either fail the login ritual or activate countermeasures by clicking on tripwires.

2.3 Threat model

Our defenses address attackers who have gained unauthorized access to a user's account on a web application using brute-forced or stolen login credentials. Attackers may interact with the web application through the user interface in a web browser, or they may send individually crafted requests directly. The tripwires portion of our defense also addresses attackers who have hijacked an existing user session either by stealing a session cookie or through access to a user's logged in device. Login rituals will not apply in this case if the real user had already completed their ritual in that session. We assume that the attacker is aware of the existence of rituals and tripwires, but does not have knowledge of the victim's specific setup of ritual steps and tripwire placements.

3 ARCHITECTURE AND IMPLEMENTATION

Our goal is to design a system which can be deployed on top of any web application with a minimal need for application-specific configuration. In order to inject tripwires and enact countermeasures, our system needs to be able to inspect incoming requests and modify outgoing responses. To satisfy these requirements, we implemented our system using mitmproxy [9].

We use mitmproxy as un uncircumventable reverse proxy so that all requests from all clients are received by the proxy, and when it is done manipulating requests it forwards the requests to the application server. The responses from the server return to the proxy where they can be manipulated again, then sent back to the

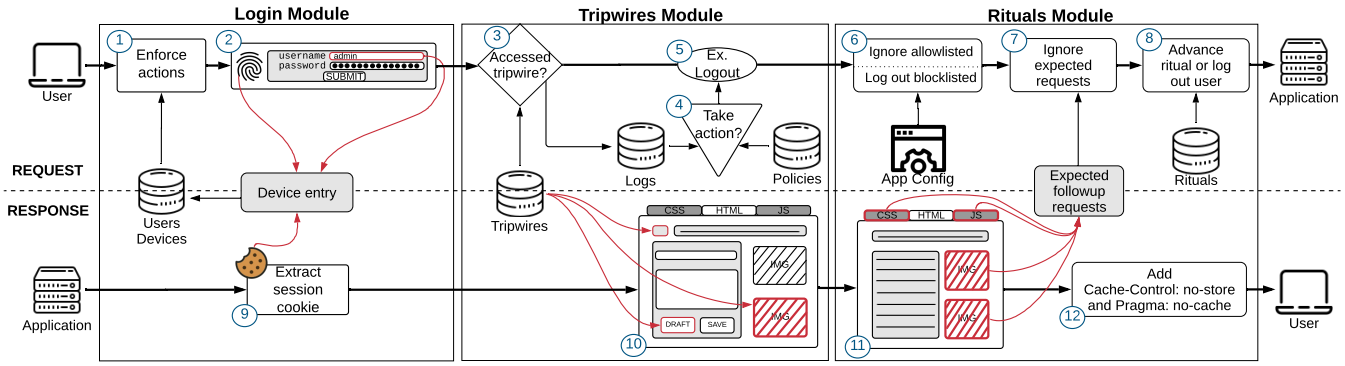

Figure 3: The architecture of our reverse proxy system. The three modules shown are added to mitmproxy's request/response pipeline. The top half shows operations on requests, while the bottom half shows manipulations of the responses.

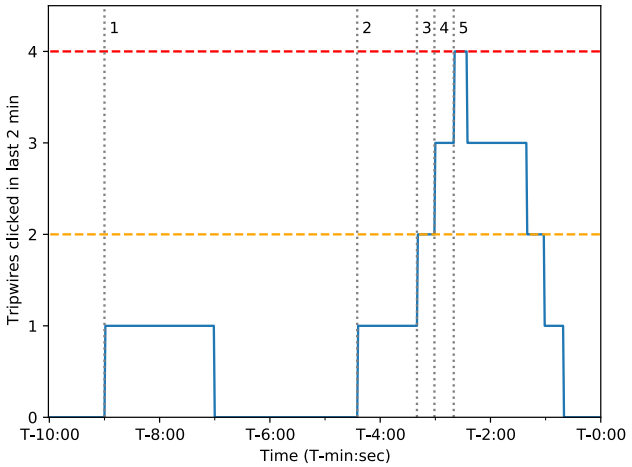

Figure 4: Policy enforcement example. Dotted vertical lines represent five tripwire accesses. The orange and red lines are thresholds for a logout and ban action respectively. The blue line is the number of tripwires within the last two minutes.

client. To set up the reverse proxy so that all requests go through it first, the DNS records for the application can be changed to point to the proxy and the actual application server may remain behind a firewall. This is similar to an application adopting a CDN service (such as Akamai) or anti-DDoS service (such as Cloudflare) where these services are responsible for handling client requests and forwarding some of these requests to the true application server, as necessary. Aside from this change, the original web application remains oblivious to our system. It is also possible to set up our proxy alongside other network middleboxes, including load balancers, by ensuring that each application server instance has its own tripwire/ritual reverse proxy instance.

Mitmproxy's modular design allows us to create custom add-ons that are added to a pipeline of handlers that each request and response passes through. The overall architecture of our system is shown in Figure 3. The three modules shown are mitmproxy add-ons, and we describe the workings of each of these below:

3.1 Identifying logins

Since our system is designed to protect logged-in users, it is important that we are able to recognize when a request is coming from a logged-in user and to be able to distinguish between users. To do this, we inspect incoming requests and check form fields submitted in the body or encoded in the URL for a username and password (Step (2) in Figure 3). Once we identify that a request contains a pending login, we save it in memory and wait for the corresponding response. If a response sets a session cookie and matches a saved pending login (9), then our system infers that the login was successful and we create (or update) the user and device records. These include the username, session ID cookie value, IP address, and a device fingerprint derived from the User-Agent header.

For all future requests, when a module needs to know which user is responsible for a given request, we can look up the user in our database by their session cookie. The specific field names used for usernames, passwords, and session cookies depend on the application therefore our system relies on configuration parameters that an administrator would populate when setting up our system for the first time.

3.2 Web tripwires

Our database contains a list of tripwires for each logged in user. This includes the type of tripwire (*injection* or *existing*), the request path that we will use to detect when this tripwire is clicked, an anchor location at which to inject the tripwire, and the HTML snippet that is injected. These can be set up by the users themselves and/or populated automatically with default tripwires for each new user our system encounters.

There are three parts to the tripwire implementation which we describe below.

3.2.1 Injection. On every outgoing response for a logged-in user that carries an HTML body, we check the list of tripwires for that user and determine if any need to be injected on this page (i.e. on the current outgoing response). If so, we parse the HTML, use the stored anchor selector to find the correct location, and insert the stored tripwire HTML snippet (10). The result is that the client receives the web page with the embedded tripwire(s). Listing 1 shows a simple example of injecting a tripwire. In this case, our system

```
<li class="mailbox unread" id="rcmliTmV3cw" ... >
  <a href="./?_task=mail&amp;_mbox=News" ... >
    News
  </a>
</li>
<li class="mailbox unread" id="rcmliaGVzdB" ... >
  <a href="/?_task=mail&amp;_mbox=Financials" ... >
    Financials
  </a>
</li>
<li class="mailbox unread" id="rcmliUHJvbW90aW9ucw" ... >
  <a href="./?_task=mail&amp;_mbox=Promotions" ... >
    Promotions
  </a>
</li>
```

Listing 1: A tripwire with the highlighted HTML snippet injected below the anchor point `#rcmliTmV3cw`. Some attributes of the elements are replaced with "..." for brevity.

is adding a fake email folder within Roundcube (a popular open-source email web application that we use in this paper) between *News* and *Promotions*. The tripwire is saved in the database with the anchor location corresponding to the `<li>` element for News. This allows our system to select that specific location to insert the highlighted HTML snippet.

3.2.2 Detection. To detect when a user has interacted with one of the tripwires, we retrieve the list of tripwires for that user and check whether an incoming request matches the path of any of their tripwires ③. In the case of *injected* tripwires, this path is in the `href` attribute of the injected HTML snippet. For *existing* element tripwires, this is the same request path that it would generate normally. When we detect that the user interacted with a tripwire, we log the event in the database and trigger a policy check. Note that because the detection of tripwires happens at the server side, there is nothing sent to clients (e.g. some special property of the markup) that could be used by attackers to differentiate between regular links as opposed to existing and injected tripwires.

3.2.3 Policy enforcement. After a tripwire interaction is detected we check the policy for the user and determine whether to trigger a countermeasure ④. A policy is defined for a user as `<X,Y,Action>` for a time period `X`, a threshold `Y`, and one of the actions described below. We check the log of tripwire events for the user/device during the last `X` seconds, and if the number of events within that period exceeds `Y`, then we take the corresponding action.

An example of this process is illustrated in Figure 4. The five vertical dotted lines indicate times when we detected tripwires were accessed. The orange line corresponds to a policy `<120, 2, logout device>` and the red line corresponds to a more strict policy `<120, 4, ban device>`. The first tripwire accessed is essentially forgiven. After the third, the user is logged out of this device. When they log back in and continue to click on two more tripwires their device is banned. The blue line then tapers off as the two minute sliding window continues and no other tripwires are clicked. This does not undo the ban on the device because device and account ban actions have a separate duration that can be configured as part of the policy. Note that in a real world deployment a longer time window may be appropriate so that an intruder has more trouble avoiding the

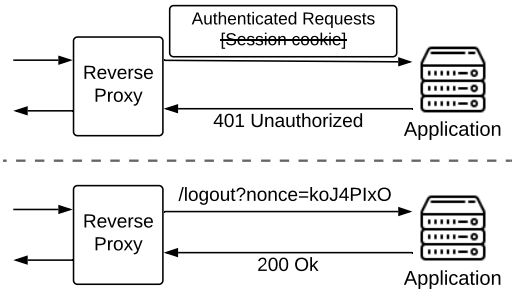

Figure 5: Two methods for implementing the logout countermeasure. (Top) Stripping session cookies. (Bottom) Send logout request with nonce extracted from HTML page.

thresholds via a "low and slow" attack. We also allow tripwires to be weighted so that some can be treated as more severe than others, but in this example all the weights are equal.

Enforcing the countermeasures is also handled by the reverse proxy. There are four actions that our system can take: i) logout a single device, ii) logout all devices for a user, iii) ban a device, and iv) ban all devices for a user.

When we log out a user they can log in again if they know the password, but this addresses cases where an intruder hijacked an existing session (e.g. stealing someone's session cookie or using an unlocked workstation with an authenticated session). There are two methods for logout which are shown in Figure 5. An intuitive solution is to send a logout request to the server on behalf of the user ⑤. This is complicated by the fact that many applications will require a nonce with the logout request to prevent users from being logged out by Cross-Site Request Forgery (CSRF) attacks. Our system is capable of extracting logout URLs, including the nonce, from HTML pages found in responses, but this requires a configuration specifying a selector to the logout button. The other solution is to strip the session cookie from requests before passing them on to the application. This will typically result in a 401 response or redirection to the login page if the resource required the user to be logged in. To do this, our system saves a flag for the device/user ⑤ and continues stripping the blocked session cookie until it sees a new successful login ①.

To ban a device or user, we set a flag in the database ⑤ and we can drop their future requests ①. The duration of the ban is a configurable part of the policy. Once the ban expires, we reset the flag and allow requests to continue. Bans can be made indefinite, but this may require administrator attention to un-ban users or devices in some cases. Banning a device is an appropriate action if the intruder guessed or stole the login credentials. Banning the entire account can prevent an attacker logging in from a new device. We reason that the small friction that this step adds (i.e. the legitimate owner of the account contacting the administrator, changing their password, and getting access back into the account) is preferable compared to allowing attackers to freely roam into a system with compromised credentials.

Overall, the policies and actions supported by our system allow administrators to choose between a wide range of countermeasure responses, not only in an inter-environment fashion (e.g. securing

Table 1: Quantity and types of follow-up requests triggered automatically by the browser after logging in and loading the landing page of each application.

	WordPress	Roundcube	OwnCloud	PhpMyAdmin	ShareLaTeX
HTML	2	4	4	2	1
CSS	13	3	31	7	1
Images	5	21	36	33	0
Scripts	27	9	110	40	2
Data	0	0	5	5	0
Fonts	0	0	3	0	4
Other	0	3	0	0	1
Total	47	40	189	87	9

a critical web application vs. a less critical one) but also in an intra-environment one (e.g. setting stricter policies for administrators vs. regular users, on a single web application). While the countermeasures range from inconvenient to severe, a design with multiple thresholds allows for policies to enact more strict countermeasures progressively as the number of tripwire accesses increases.

3.3 Login rituals

Following every traditional login, the rituals module enforces a user's login ritual until it is completed or broken. Described below are several steps required to handle certain complexities of login rituals.

If rituals are enabled then the ritual progress is set to zero for a device after a login is detected in the login module. While the ritual progress is between zero and the length of the ritual, our system inspects requests and increments the ritual progress if and only if the request matches the current step (8). Otherwise the ritual is broken and the device is logged out using one of the methods shown in Figure 5. When the ritual is completed, the progress is cleared and all requests from the device will skip over the rituals module until the next login.

Even though it is simple to determine if a request matches the next ritual step to advance the progress, rejecting requests can be much more difficult. The reason is that any action from a user can cause the browser to request dozens of additional subresources, such as, iframes, CSS, scripts, images, and videos. For example, Table 1 shows the number of follow-up requests generated from the landing pages of the applications we tested. These requests will not necessarily happen in the same order, some may not happen at all if they are cached, and new subresources can be added over time as an application evolves and updates. We need to be able to allow these requests without breaking the ritual when they are the consequence of a user action that was a valid part of the ritual. The goal then is to determine when a request is actually outside of the ritual so that we can log out the device. We describe specific challenges and our solutions below.

3.3.1 Response types. One way to handle follow-up requests could be to allow requests for static resources based on their content type. The problem with this is that the type cannot always be determined from the request, and if our system waited for the response to determine the type, then it has already passed the potentially harmful request to the application server. Therefore, our solution needs to only evaluate requests so that actions outside of the ritual do not impact the server.

3.3.2 Anticipating follow-up requests. The most direct solution is to consider the logic that causes a browser to initiate follow-up requests. Our system predicts these requests by monitoring the content of the application server's responses for HTML which will be interpreted by the client browser. When the rituals module encounters an HTML response to an authenticated user who has not yet completed their ritual, it parses the response and looks for tags (`<img>`, `<link>`, `<script>`, etc.) that the browser will use to fetch additional resources automatically (11). From these tags, we build a list of anticipated follow-up requests. When we receive these requests we know that they were triggered by part of the ritual, so we can forward them along to the server without logging the user out (7). For each of these anticipated requests, we also keep a time-to-live value so that we only allow it for a short time after seeing the parent document that we expect to trigger it. This approach to identifying follow-up requests allows our system to continue recognizing ritual requests even as the content of a site changes over time.

The method above works best for applications that assemble the view on the server side because our system will know what requests to expect. Some applications will also use JavaScript to trigger requests for additional subresources which were not referenced in HTML. While we could search for URLs in JavaScript resources that pass through our proxy, they may be hard to find if built dynamically and it is hard to know whether they are triggered on load or require user interaction first.

We tackle this challenge by taking advantage of client-side capabilities at the time of ritual creation. When a user first creates their login ritual using our browser extension user interface, we can record all requests made to the application server as well as what was clicked on. By keeping track of when each request was sent and when each user action was performed, we can construct an attribution chain that tells us "user action A triggered requests B, C, and D." This technique is less resilient to changes in the application because it gives us only a single snapshot, but it supplements the above approach and allows us to handle requests that are generated from JavaScript and cannot be identified in HTML responses.

3.3.3 Dynamic requests from JavaScript. Recording requests during ritual creation identifies requests made from JavaScript, but if the generated requests change as the application evolves over time we may not recognize them. We first address two types of behavior causing unknown JavaScript requests that can be dealt with using application specific configurations set up by an administrator.

Some applications use AJAX requests to send heartbeat messages. These are used to tell the server that the session is still active and they are triggered by a timer without interaction from the user. As such, they will not necessarily be captured at the time of ritual creation, nor expected based on a parsed HTML document. If a heartbeat message is received while a user is completing their login ritual, then our module would treat it as a ritual violation since it did not know to expect that request. Because these requests follow a predictable pattern, an administrator can observe the form of heartbeat requests and add that specific heartbeat endpoint to an allowlist to be ignored by our system.

A related issue results from requests sent by JavaScript that contain a timestamp or nonce in a URL parameter. While the behavior

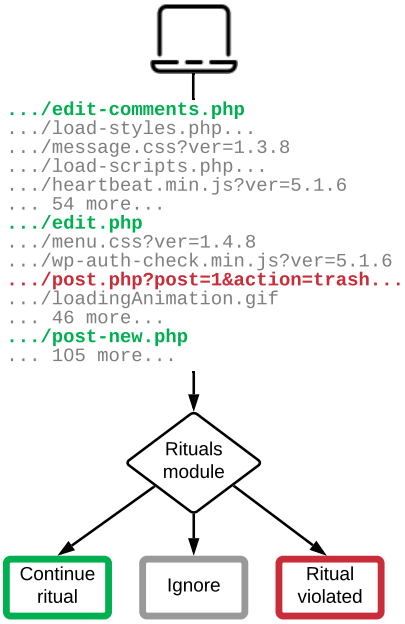

Figure 6: A shortened example of a ritual for WordPress. The user's ritual includes clicking on Comments, Posts, and then Add New. The requests in green correspond to these actions. Requests in grey are follow-ups to be ignored. The request in red is a request to trash a post which violates the ritual and would log the user out.

that triggered the request may have been observed and recorded during ritual setup, it will appear different each time. In general, our system needs to be as strict as possible in matching URLs while enforcing rituals because the format of requests varies widely depending on the application, and even query parameters can have a dramatic impact on the resource that will be returned from the application. However, when a request differs only by a timestamp or nonce component, it is appropriate to treat it the same as the initial anticipated request. This also requires an administrator with some knowledge of the web application's behavior, but our system supports a configuration that specifies query keys to ignore while still evaluating the rest of the request URL.

A more significant challenge comes from changes in an application's content which are fetched by name from JavaScript. For example, a new image that was not seen when the ritual was created can be fetched by a script. Our system would not know to ignore this request even though it was loaded automatically as a follow-up request to a valid ritual action. This can be addressed by configuring more general patterns of request URLs to ignore during ritual enforcement. As an example, one might add requests for image resources to the allowlist if scripts are fetching images and more are added over time. It may be an acceptable risk to allow access to images without completing a ritual, but this depends on the application. To complement the allowlist and reign in potentially dangerous actions, an administrator can also configure a blocklist of request patterns that must not be allowed until the ritual is fully completed. Checking against these configured lists is the first procedure applied to requests in the rituals module (6).

3.3.4 Caching. Caching is used for resources that are accessed frequently but modified infrequently to reduce latency and improve performance. If the responses for a user's ritual actions are cached, then the reverse proxy will not see the request and the ritual cannot be advanced. Our system addresses this by first clearing the browser's cache for the application while a ritual is being created, and then adding the `Cache-Control: no-store` and `Pragma: no-cache` headers for future responses to the ritual paths (12) set up by the user. These headers ensure that the user's browser will trigger the necessary requests to complete the ritual as users interact with the web application. Note that we do not need to disable caching for the entire application. Only the initial requests corresponding to ritual actions are affected. All other responses, including static resources fetched by follow-up requests to the ritual action, may still be cached as normal.

Figure 6 shows a real example of a login ritual for the WordPress web application and how our system will react given requests that continue the ritual, requests that need to be ignored, and requests that violate the ritual.

3.4 User interface

An important part of both tripwires and rituals is that they will be the most memorable and most effective if users create their own. This allows them to take advantage of their own familiarity with the application and makes it more likely they will remember the ritual and which tripwire elements to avoid in the future. This means we want to make it easy for users to set them up, without requiring them to understand HTML or HTTP requests.

To this end, we built a user interface as a Chrome extension which gives us client-side access to the user's interactions with the web page and allows us to highlight elements to help guide users. For a visual demonstration of the interface, we encourage readers to visit `click-this-not-that.github.io` which has videos showing tripwire and ritual creation.

The same session cookies that we use to recognize users in the tripwires and rituals modules are used to authenticate users creating new tripwires or rituals. Changing either of these is similar to changing an existing password, so we suggest that a real deployment may want to force the user to re-authenticate before using this interface.

In ritual creation mode, the browser extension first clears the browser's cache to make sure all requests will be sent to the server. The user can then click on a sequence of elements throughout the application while the extension records all outgoing requests to the application server and the elements that were clicked. The current steps of the ritual are shown in an overlay box and when the user is finished they click save in this overlay. The ritual is sent to the back-end and enforced following the next log in. As discussed above, the recorded requests are used to enforce the ritual and `Cache-Control: no-store` and `Pragma: no-cache` headers are added to responses for ritual requests so that our system will continue to receive requests for those paths after future logins.

Setting up tripwires offers users more flexibility in terms of customizing what their tripwires look like and where they are placed. Again the user can enter tripwire-creation mode from within our extension which will disable navigation and start highlighting

elements that the user is hovering over with their mouse. They can click to select an element and then choose which type of tripwire they are trying to create. If they choose *existing*, that part of the application will be treated as a tripwire. If they choose *injection* then the selected element becomes the anchor point at which to insert a new tripwire element. The default behavior is to duplicate the selected element. The user can then customize the content and tripwire URL path. These are the most important options for the believability of a tripwire. We allow the user to choose a deception that they think is most effective. To make this easier in some cases, we provide a suggestion for the tripwire path. This is based on a Markov model of a subset of URLs in the Common Crawl dataset [1] and seeded with strings from the application so that the generated path matches the other paths. Advanced settings can allow a user to directly edit the HTML snippet to be injected or adjust the injection location, but our goal is to abstract this away from most users.

4 USER STUDY

In order to test rituals and tripwires, we designed an online user study and recruited participants using Amazon Mechanical Turk and student emails. There are two parts to this study which are described in detail below.

All parts of the study were coordinated through a web portal we built to walk participants through the instructions and tasks. From there they were directed to a Roundcube webmail application where they would interact with tripwires and login rituals. We chose Roundcube as the application because its interface is similar to other webmail clients such as Gmail, and user familiarity with the interface is an important aspect of tripwires and rituals. We populated an account on Roundcube with emails by subscribing to mailing lists and further customized the inbox by manually creating emails, such as bill notifications, social media invites, and personal messages. We created a large number of accounts and copied all these emails into each new mailbox. Each user was given a different account so that none of their actions would impact other participants, but the set up remained the same.

To obtain a clear picture of what each study participant did on Roundcube, we made use of a commercial record-and-replay service [2]. We embedded the session recording script into every page on Roundcube and modified it to proxy events through our study server. This allowed us to verify that a participant had successfully recorded some action before allowing them to continue, and to save a reference to the recording for later use. A video showing a live example of a user going through the first part of the study is available at `click-this-not-that.github.io`.

4.1 Effectiveness of tripwires

The first part of the user study was designed to test how well tripwires perform at detecting intrusions. The setup of the study was inspired by Salem and Stolfo [25] who conducted a user study in 2011 to test decoy documents deployed in the file system of a laptop. They recruited 40 volunteers who were given 15 minutes with a laptop to steal information.

Participants in our study were given a scenario in which they have broken into a webmail account with stolen credentials that we provided. To motivate their exploration, we asked them to find three flags embedded in different parts of the application and enter the discovered flags' values in a form on the study web page. These flags were embedded images with secret words on them. We created 16 tripwires spread throughout different parts of Roundcube (listed in Table 2). Some tripwires are visible only within certain views. For example, two tripwires were added to the list of email folders which are in the inbox view. Other tripwires, like the injected "Account" link, are visible from any view. Instead of targeting a specific number of total tripwires, our strategy was to have a few tripwires present across the most common views. Note that in real deployments of our system, there is no right or wrong number of tripwires. The decision could be made individually by each user or by the administrators of a system, finding the right compromise between false positives (i.e. benign users triggering their own tripwires) and false negatives (i.e. attackers not triggering tripwires). Participants were warned that parts of the application would be tripwires that would detect their intrusion if they clicked on them. We did not provide any other details about how tripwires work or where they were placed. All of the participants' sessions recordings were saved and our system recorded each tripwire access. We chose not to enable any policies with countermeasures during this experiment so that we could observe how many tripwires would be encountered without interruption.

Table 2: Tripwires included in the user study sorted by the number of times participants clicked on them.

Type	Element	Number of clicks
Existing	Special folders settings section	47
Injection	Fake account settings category	42
Injection	Duplicate Important folder	27
Injection	Financials folder	27
Injection	Advanced search icon	11
Injection	Advanced settings section	11
Injection	Search tools button	9
Injection	Account link	6
Injection	Work addresses group	4
Injection	Help link	4
Existing	Add contact button	3
Existing	Contact QR code	3
Injection	Archive icon	1
Existing	Add new identity button	1
Injection	Duplicate save draft button	0
Injection	Email delete button	0

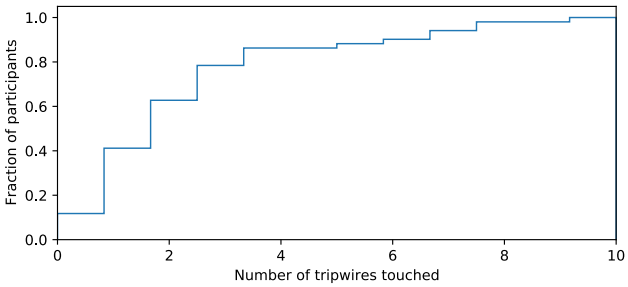

Figure 7: Cumulative distribution of the number of tripwires clicked by each participant that found at least one flag.

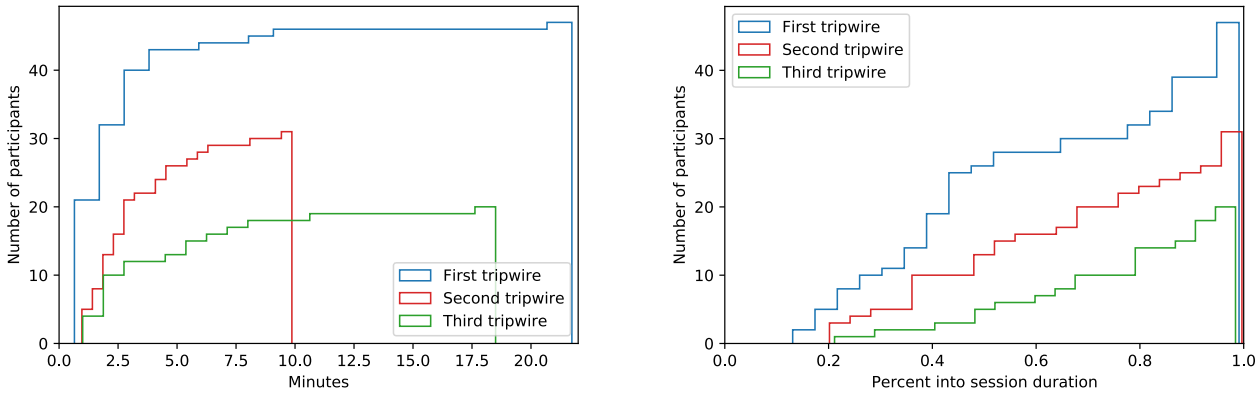

Figure 8: Cumulative distribution of time taken before a simulated attacker clicked on one, two, or three tripwires, presented in raw time on the left and normalized by the overall duration of sessions on the right.

Many users who came to our study site did not follow through on the tasks, so we focus the analysis in this part on 51 participants who successfully found at least one of the flags embedded in the application (41 of them found all three). Figure 7 shows the distribution of the number of tripwire accesses recorded by our system for each user. The most notable result is that 88.2% of users clicked on at least one tripwire. This indicates that the tripwires system has a good chance of detecting intrusions.

Since we want to detect these intrusions as soon as possible, it is useful to see how long it took before the tripwires were encountered. Figure 8 shows how long since loading the login page it took for users to click their first, second, and third tripwires. 91.1% hit their first tripwire within 5 minutes. Depending on how strict the tripwire policies are set, this could log users out right away. Otherwise, 66.7% could be caught in the first 10 minutes with a more relaxed policy that waits for two tripwires to be clicked.

Finally, we examine the types of tripwires that were most effective. Table 2 shows how many times each tripwire was clicked. While *injected* tripwires were more attractive than *existing* tripwires overall, the fact that the most clicked tripwire was *existing* shows that this can be effective. A real user needs to be careful that *existing* tripwires are not elements they need to use. To be conservative in our placement, the other *existing* elements we chose were located in deeper UIs of the web application. We see that certain prominent *injected* elements such as email folders received a high number of clicks. This type of tripwire can be particularly effective because they are hard to distinguish even for someone who does have intimate knowledge of the normal user interface for another account. The wide range of clicks for different tripwires indicates that successful detection is less about the total number of tripwires added and more about their specific placements and how enticing they appear.

4.2 Ritual composition and memorability

In the second part of our study, we explained the concept of login rituals, then instructed participants to log in to Roundcube and take a sequence of actions to create a login ritual for themselves. We observed 39 ritual creations, ranging in length from 1 to 24 with an average of 6.1 steps.

Table 3: Categories of actions chosen by participants when creating their login rituals.

Folders	Emails	Settings	Top level pages	Other
82.05%	33.33%	5.13%	30.77%	25.64%

Since participants were not specifically instructed on what elements make a good ritual, we examine the choices they made based on their understanding. We manually labeled the rituals and categorized actions as follows.

Folders Opening an email folder on the inbox page.
Emails Opening or marking an email.
Settings Clicking on settings sub-menus.
Top level pages Navigating to Mail, Contacts, or Settings.
Other Any other actions such as refreshing, searching, composing an email, etc.

Table 3 shows the percentage of rituals that included each category. 82% of the rituals involved opening an email folder. We believe this is a good choice because it is reproducible and in a real-world setting they could use folders they created making it easier to compose a unique combination of actions. Along those lines, we notice that all 39 rituals created by participants were unique. This diversity suggests that users can create individualized rituals that are difficult for an intruder to guess.

The next part of the study asked participants to return to our site each day to reproduce the ritual they created. This allowed us to learn how well users can remember their rituals. Participants recruited through email received email reminders and both groups were able to opt in to push notifications when it was time to return for the next step. Since this was an online user study and returning each day is an additional inconvenience, we saw the participation decrease significantly even though compensation was greater for finishing all tasks. 14 participants returned for at least 5 days in the following week.

	1	2	3	4	5	6
1	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	✓	✓
4	✓	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	✓	✓
6	✓	✓	✓	✓	✓	M
7	✓	✓	✓	✓	✓	M
8	X	✓	M	✓	✓	✓
9	X	✓	✓	✓	X	✓
10	R	✓	M	✓	✓	✓
11	X	✓	✓	✓	M	X
12	R	R	M	R	✓	✓
13	X	X	X	X	✓	X
14	M	X	X	X	X	X

Figure 9: Ritual completion over time. Each row represents one study participant. Each cell indicates whether they completed the ritual (✓), failed the ritual (X), asked for a reminder (R), or missed the day (M).

Figure 9 displays the results of the ritual follow-up tasks for each of these 14 participants. Each cell represents a day where they either completed the ritual, failed the ritual, or missed the task. They also had the option to ask for a reminder if they could not remember their ritual. In this case, they could click a link to view a recording of what they did to set up the ritual. Days when they requested reminders were counted as failures in our analysis, but for the two participants who used this feature, we see that they were eventually able to reproduce the ritual. A real world deployment could implement some form of reminder for a limited time to help new users become accustomed to login rituals.

Half of the participants managed to complete their ritual every time. In some cases of failure, they had forgotten only a single step or duplicated a click. These cases are similar to a user mistyping their password. The fact that we see these same users succeed in their other attempts indicates that they would likely be able to re-login and complete the ritual. 11 out of 14 participants (79%) succeeded more often than they failed. We had hypothesized that some users might remember their rituals at first and then forget them, but we did not observe this pattern. In fact, failures (including requests for reminders) were more likely to occur earlier in the week, then corrected afterwards. In Figure 10 we see the distribution of ritual lengths among these 14 participants, and notice that the only case with no successes was also the one with the longest ritual. Meanwhile, the cluster of points in the top left represents the shorter and easier to remember rituals.

4.3 Ethical considerations

We obtained approval from our IRB to conduct this online user study. The session recordings collected from each user do not include any personal information and are carried out on a dummy application set up specifically for this study. AMT workers recruited for only the tripwires task were paid $1.00, while others recruited for tripwires

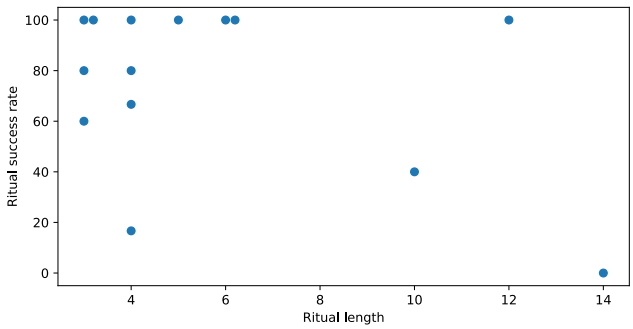

Figure 10: Observed success rate at reproducing rituals based on the length of the ritual.

and rituals tasks were paid $8.00. Students recruited through email were entered into a raffle for a $50.00 gift card. Email addresses and AMT worker IDs, which were used to organize participation, were removed from the data-set following completion and compensation.

5 PERFORMANCE

There is a performance trade-off in using our system. Our reverse-proxy solution intercepts requests, checks a database for specific tripwire or ritual paths, and parses/modifies HTTP responses. In order to measure the overhead that this incurs, we set up tests using a Selenium crawler to log in and load the main page on five open-source web applications.

We compare the page load times under three different setups to quantify the slowdown caused by our system. Each test begins with an empty client-side cache, so we are measuring performance when all resources need to be fetched from the server. As a control, we first run the crawlers against the web application itself with no proxy. We then repeat this process with mitmproxy running in reverse proxy mode, but not loading any additional modules. Finally, we run the crawlers against the application with the tripwires and rituals modules enabled on mitmproxy. Figure 11 shows the results of these performance tests. We report the steady-state performance after an initial run to warm up server-side caches. The performance overhead varies across each application, but we see that introducing a reverse proxy increases load times with the greatest slowdown seen in phpMyAdmin. Despite the fact that our modules parse and sometimes modify requests/responses, the results show that the parsing and potential modification has a small effect compared to mitmproxy alone. One of the weaknesses of mitmproxy is that it is single threaded, but there is the potential to develop a proxy which allows rewriting requests and response with a focus on performance. Proxy-based web re-hosting services, such as `proxysite.com` (ranked in the Alexa top 5K), are examples of successful large-scale applications with similar requirements because they rewrite responses to point links back to their proxy domain.

Finally, it is important to stress that when a page-load time increases from 2 seconds to 4 seconds, this does not mean that users stare at a blank screen for an additional 2 seconds. Each page is comprised of HTML code and tens of remote resources, all of which will have to be fetched, *before* a browser marks a page

as completely loaded. Web browsers start rendering content as soon as the first remote resource is loaded, allowing users to start "consuming" the content and interacting with the page, long before the page-load event is triggered. As such, we argue that even with a non-optimized solution such as mitmproxy, the perceived delay will be significantly smaller than the measured page-load delay.

6 DISCUSSION

Limitations and future work

Phishing web pages can convince users to give up their login credentials while masquerading as a real web application. Particularly sophisticated phishing pages can also behave as a web proxy, forwarding requests to the real application and responses back to the victim [13]. This type of man-in-the-middle attack can observe multi-step authentication from a user to capture their cookies and hijack their session. In the same way that this attack bypasses most forms of two-factor authentication, it is also capable of bypassing login rituals by fooling victims into logging in and completing their ritual through the phishing proxy. Tripwires, however, are still valuable in this setting because the real user knows not to click on tripwires. Therefore, the users' specific tripwires will not be identified to the attacker by observing their behavior through the phishing proxy. Once the session has been hijacked, the attacker will still have to avoid the tripwires. If they trigger a tripwire policy, then their stolen session could be invalidated and their device blocked.

Our user study evaluated how users will approach creation of rituals, but due to challenges associated with a fully online study with no live guidance, we did not evaluate tripwire creation. Future work could build on the creation interface and observe how users approach the design and placement of deceptive elements. Additionally, if these are deployed on a live application, there is an opportunity to measure false positives for these custom tripwires over time.

Role of deception in web authentication

In this paper we showed that it is possible to build an application-agnostic system that can add deceptive capabilities to the authentication component of web applications, enabling users to further authenticate themselves via the mechanisms of tripwires and rituals. Our user study showed that, even for users who have no particular incentive in getting it right, the majority were able to choose rituals and successfully use them to authenticate themselves to the web application, over a period of one week.

We want to underline that the proposed system is *not* the panacea for all web authentication problems and, in certain cases, it is entirely reasonable for administrators to choose *not* to deploy it, given the need to train users in using it. In other cases, only some of the features provided by our system will be useful. For example, in environments, where attackers are able to register their own accounts, they may be able to compare the links available to their accounts, against the links in a compromised account, thereby identifying the injected links that serve as tripwires. The administrators of these environments can therefore configure our system to only use existing links, thereby removing that differential-analysis capability from attackers.

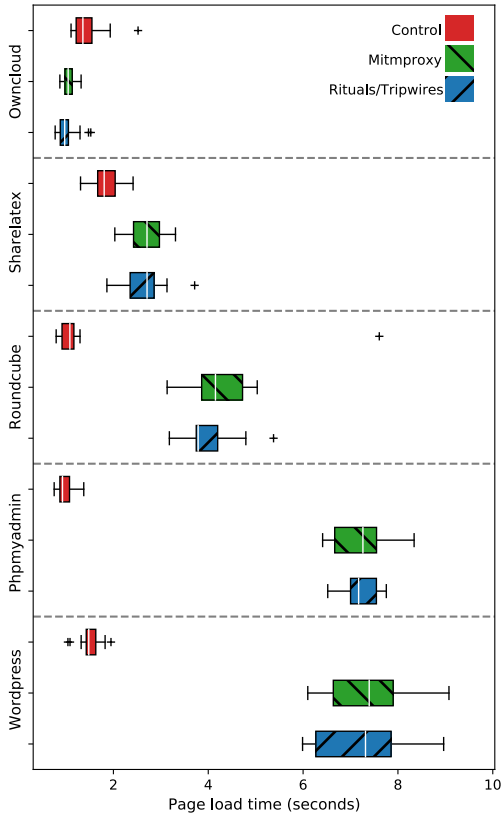

Figure 11: Results of performance tests showing loading times of 5 applications running on their own, with basic mitmproxy, and with our tripwires and ritual modules enabled.

In general, our goal with this paper was to design, implement, and evaluate a generic system for using deception in web authentication, which the administrators of different deployed web applications can use according to their needs and nature of their applications. This is one of the main reasons why we will be open-sourcing our system, to enable both the research community as well as security practitioners to experiment with deception and further understand exactly how it can be applied to real-world deployments of web applications.

7 RELATED WORK

Web tripwires are closely related to the idea of honeypots [30]. Honeypots are traditionally thought of as fake servers, but many works have extended the concept to create honeytokens, decoys, and traps for various other contexts and types of resources [4, 6, 14, 18–20, 23, 35]. Additionally, the use of deception for intrusion detection has been explored in the context of decoy files [8, 25, 34], decoy mobile applications [27], and even decoy source code [21].

Intrusion Detection via Decoy Files. In 2009, Bowen et al. [8] began a foundation for using decoy files to detect intrusions. They formally defined a set of seven generally applicable properties of decoys to guide design and deployment. They introduced three types of

decoy files which can be detected in different ways. Honeytokens embed fake sensitive information, such as webmail credentials and bank account/credit card numbers that are monitored on external systems. They attempted to evaluate the decoys using honeypots managing to attract 20 attackers.

In 2011, Salem and Stolfo [25] expanded on the work of Bowen et al. Rather than relying on honeypots and real world attackers, they chose to evaluate decoy files with user studies to find the right number of decoys balancing false positives with detection rate. They showed that 10-20 decoy files placed on volunteers' own systems generated a tolerable number of false positives over a 7 day period. In another experiment, they simulated attackers using 40 volunteers who were given 15 minutes with a laptop to steal information. They were given instructions and a personal back-story motivating them to steal data for financial gain. They were split into four groups varying the number of decoys on the system. They found that all volunteers were detected by at least one decoy and increasing the number of decoys beyond 20 did not lead to a significant increase in the number of accesses. This offers compelling evidence that decoy files can used effectively for intrusion detection and produced results which guide future decoy deployments. The setup of the tripwire portion of our user study was inspired by this work, in terms of the number of participants, the task we gave them, and the number of tripwires we chose to deploy.

In 2015, Voris et al. [34] continued to build on decoy files with a focus on automatic deployment. The authors used the same type of user studies as Salem and Stolfo, but they evaluated different schemes for automatically naming and placing decoy files. Their work also included a long-term study of false positives with their automatic deployment system. With decoy files on volunteers' personal systems for a period of 2 months, they found that after an initial spike just following deployment, decoy accesses were rare for the majority of the experiment. This shows that users can get used to decoys on their system and generate few false positives. We expect that the same is true for our web tripwires and rituals when deployed on often used applications, because their success is based on long-term familiarity.

Deception on the Web. Recent work has explored some forms of deceptive web resources for intrusion detection as well as obfuscation contributing to moving target defenses. In 2017, Han et al. [15] created a reverse proxy system to inject deceptive web elements and evaluated their effectiveness in a capture the flag setting. Their work focused on web elements that are hidden from the user, such as cookies, form fields, fake accounts, and URLs in robots.txt and HTML comments. Fraunholz and Schotten [12] used a similar system but also included techniques such as returning false version info and status codes meant to deceive crawlers rather than detect intrusions. More recently in 2020, Sahin et al. [24] used similar deceptive elements, but with the addition of a clone of the web application to which attackers were redirected after detection. These works primarily focus on defenses that will catch crawlers and vulnerability scanners. Compared to these works, our system implements tripwires as elements of the user interface. We focus on protecting individual user accounts, and the personalized setup for each means that users should have the familiarity necessary to avoid false positives without having to hide our deceptive elements.

Network-level Login Rituals. The concept of login rituals in this paper is inspired by the idea of port knocking and single packet authentication (SPA). The idea behind port knocking is to keep a strict firewall which drops all incoming connections until a monitor sees SYN packets sent to a predetermined set of secret ports. Then the server port is opened up to allow communication for a limited time. SPA is very similar, but uses only a single packet which contains a hashed password, instead of sending multiple requests where the set of ports is the password. In 2006, Sebastien Jeanquier [17] summarized the basics of port knocking and addressed practicalities, limitations, and improvements. Some challenges identified include allowing only the client that completed the knock, replay attacks from an eavesdropper, and a single shared secret between multiple clients. Several works have sought to improve port knocking protocols over the years [3, 10, 32], but the core concept is the same. Login rituals are similar in that there is a sequence of steps that a user must take before they are considered fully authenticated. If a user account is analogous to the server which opens its port, then one difference is that rituals do not conceal the existence of the user. However, advantages over port knocking are that the dependence on the specific application precludes password re-use, and rituals can make use of actions that a user would take anyway.

Intrusion Detection via Behavior Modeling. In the area of modeling user behavior for attack detection, Shonlau et al. [28] collected a data set of UNIX commands and presented techniques for change detection to identify masquerading users. Salem and Stolfo [26] also used behavior modeling, but focused on searches in a file-system. In a web setting, Solano et al. [29] modeled mouse and keyboard behaviors on the login screen to introduce a layer of risk-based authentication. Tripwires and rituals are conceptually related to behavior modeling in that there is an expected user behavior and an attacker who deviates by not completing a ritual or by clicking on a tripwire will reveal themselves as an intruder. A significant distinction is that both of our mechanisms are deterministic. False positives only occur when a user makes a mistake. Additionally, because users create their tripwires and rituals, they are aware of what they need to do to remain trusted, whereas statistical behavior models are invisible to benign users. This makes false positives more frustrating since the user does not know exactly what they did to trigger an alert.

8 CONCLUSION

In this paper, we presented two techniques – web tripwires and login rituals – that rely on deception to extend authentication in web applications. We take advantage of users' habits and familiarity with applications they use every day to help protect their accounts. Tripwires and rituals do not suffer from the re-use problem that is common with passwords and can be combined with existing MFA systems to reduce user friction and increase an account's security in cases of sophisticated phishing attacks. We demonstrated how these defenses can be realized with our reverse proxy system and evaluated the mechanisms through an online user study. We found that up to 88.2% of simulated attackers could be detected by web tripwires and half of our study participants were able to complete their rituals every time, even a week after setting them up. While this work is not meant to replace standard authentication

practices, our results show that these deception-based mechanisms can provide effective layers of additional security to existing web applications, in an application-agnostic fashion.

Acknowledgments: We thank the reviewers for their valuable feedback. This work was supported by the Office of Naval Research under grant N00014-20-1-2720 as well as by the National Science Foundation under grants CNS-1813974 and CNS-1941617.

Availability: To encourage exploration of tripwires and rituals as practical defenses for real-world applications, we plan to open-source our system and share anonymous data obtained from our user study. These will be be made available at:

`click-this-not-that.github.io`

REFERENCES

[1] 2020. Common Crawl. https://commoncrawl.org/the-data/get-started/

[2] 2020. Mouseflow: Session Replay, Heatmaps, Funnels, Forms & User Feedback. https://mouseflow.com/features/.

[3] Hussein Al-Bahadili and Ali H Hadi. 2010. Network security using hybrid port knocking. *IJCSNS* 10, 8 (2010), 8.

[4] Spiros Antonatos, Iasonas Polakis, Thanasis Petsas, and Evangelos P Markatos. 2010. A systematic characterization of IM threats using honeypots. In *ISOC Network and Distributed System Security Symposium (NDSS)*.

[5] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*. 1697–1714.

[6] Marco Balduzzi, Payas Gupta, Lion Gu, Debin Gao, and Mustaque Ahamad. 2016. Mobipot: Understanding mobile telephony threats with honeycards. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 723–734.

[7] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. 2012. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*.

[8] Brian M Bowen, Shlomo Hershkop, Angelos D Keromytis, and Salvatore J Stolfo. 2009. Baiting inside attackers using decoy documents. In *International Conference on Security and Privacy in Communication Systems*. Springer, 51–70.

[9] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. 2010–. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/ [Version 5.0].

[10] Rennie Degraaf, John Aycock, and Michael Jacobson. 2005. Improved port knocking with strong authentication. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 10–pp.

[11] Dinei Florencio and Cormac Herley. 2007. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*.

[12] Daniel Fraunholz, Daniel Reti, Simon Duque Anton, and Hans Dieter Schotten. 2018. Cloxy: A context-aware deception-as-a-service reverse proxy for web services. In *Proceedings of the 5th ACM Workshop on Moving Target Defense*. 40–47.

[13] Kuba Gretzky. 2018. Evilginx 2 - Next Generation of Phishing 2FA Tokens. https://breakdev.org/evilginx-2-next-generation-of-phishing-2fa-tokens/.

[14] Payas Gupta, Bharath Srinivasan, Vijay Balasubramaniyan, and Mustaque Ahamad. 2015. Phoneypot: Data-driven Understanding of Telephony Threats. In *NDSS*.

[15] Xiao Han, Nizar Kheir, and Davide Balzarotti. 2017. Evaluation of deception-based web attacks detection. In *Proceedings of the 2017 Workshop on Moving Target Defense*. 65–73.

[16] Troy Hunt. 2017. Password reuse, credential stuffing and another billion records in Have I been pwned. troyhunt.com. (2017).

[17] Sebastien Jeanquier. 2006. An Analysis of Port Knocking and Single Packet Authorization MSc Thesis.

[18] Steffen Liebergeld, Matthias Lange, and Collin Mulliner. 2013. Nomadic Honeypots: A Novel Concept for Smartphone Honeypots. In *Workshop on Mobile Security Technologies (MoST)*. San Francisco, CA.

[19] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. 2016. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of the 2016 Internet Measurement Conference*. 65–79.

[20] Youngsam Park, Jackie Jones, Damon McCoy, Elaine Shi, and Markus Jakobsson. 2014. Scambaiter: Understanding targeted Nigerian scams on craigslist. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[21] Younghee Park and Salvatore J Stolfo. 2012. Software decoys for insider threat. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. 93–94.

[22] Thanasis Petsas, Giorgos Tsirantonakis, Elias Athanasopoulos, and Sotiris Ioannidis. 2015. Two-factor authentication: is the world ready? Quantifying 2FA adoption. In *Proceedings of the eighth european workshop on system security*. 1–7.

[23] Fabien Pouget, Marc Dacier, and Hervé Debar. 2003. White paper: honeypot, honeynet, honeytoken: terminological issues. *Rapport technique EURECOM* 1275 (2003).

[24] Merve Sahin, Cédric Hebert, and Anderson Santana de Oliveira. 2020. Lessons Learned from SunDEW: A Self Defense Environment for Web Applications. In *NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb'20)*.

[25] Malek Ben Salem and Salvatore J Stolfo. 2011. Decoy document deployment for effective masquerade attack detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 35–54.

[26] Malek Ben Salem and Salvatore J Stolfo. 2011. Modeling user search behavior for masquerade detection. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 181–200.

[27] Malek Ben Salem, Jonathan Voris, and S Stolfo. 2014. Decoy applications for continuous authentication on mobile devices. In *Symposium on Usable Privacy and Security (SOUPS)*, Vol. 2.

[28] Matthias Schonlau, William DuMouchel, Wen-Hua Ju, Alan F Karr, Martin Theus, and Yehuda Vardi. 2001. Computer intrusion: Detecting masquerades. *Statistical science* (2001), 58–74.

[29] Jesús Solano, Lizzy Tengana, Alejandra Castelblanco, Esteban Rivera, Christian Lopez, and Martın Ochoa. 2020. A few-shot practical behavioral biometrics model for login authentication in web applications. In *NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb'20)*.

[30] Lance Spitzner. 2003. *Honeypots: tracking hackers*. Vol. 1. Addison-Wesley Reading.

[31] Lance Spitzner. 2003. Honeytokens: The other honeypot.

[32] Vikas Srivastava, Alok Kumar Keshri, Abhishek Dutta Roy, Vijay Kumar Chaurasiya, and Rahul Gupta. 2011. Advanced port knocking authentication scheme with QRC using AES. In *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*. IEEE, 159–163.

[33] Sebastian Uellenbeck, Markus Dürmuth, Christopher Wolf, and Thorsten Holz. 2013. Quantifying the security of graphical passwords: the case of android unlock patterns. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 161–172.

[34] Jonathan Voris, Jill Jermyn, Nathaniel Boggs, and Salvatore Stolfo. 2015. Fox in the trap: thwarting masqueraders via automated decoy document deployment. In *Proceedings of the Eighth European Workshop on System Security*.

[35] Susan Marie Wade. 2011. SCADA Honeynets: The attractiveness of honeypots as critical infrastructure security tools for the detection and analysis of advanced threats. (2011).