Knocking on Admin's Door: Protecting Critical Web Applications with Deception

Billy Tsouvalas and Nick Nikiforakis

Stony Brook University, NY, USA
{vtsouvalas,nick}@cs.stonybrook.edu

Abstract. In this paper, we introduce PageKnocker, a deception-based supplementary authentication mechanism, aimed primarily at protecting the public-facing authentication endpoints of critical web applications. PageKnocker is inspired by the network security concept of port knocking, and uses the requests to the web application as a means of authentication for the login page. Specifically, the authentication is successful (i.e. the user gets to access the login page of a web application), if the user's request sequence matches their personal predefined request sequence. In this manner, PageKnocker offers web application administrators the comparative advantage of knowing the nature of a visitor even before that visitor sends the first set of credentials. Alongside PageKnocker, we introduce two deceptive login environments, one overtly deceptive and one clandestine, towards which we direct any unauthenticated user attempting to reach the real login page. To evaluate the security and usability of page knocks, we deploy PageKnocker-protected honeypots in the wild and perform a separate user study, showing that PageKnocker can resist tens of thousands of brute-forcing bots, while remaining usable and intuitive.

1 Introduction

Following the increasing popularity of interactive web services over the last two decades, reliance on web applications for everyday tasks has become common place. Whether it be email services, e-commerce, online learning, news outlets, or social media, web applications have permeated almost every aspect of the online experience. As a result, web applications have become primary targets for malicious actors, whose objectives span a wide array, ranging from violations in confidentiality and data privacy to larger scale operational disruptions.

The main and first line of defense of web applications is credential-based authentication. In order to accommodate users, web applications provide login pages, where users are asked to provide their usernames and passwords. These public-facing login endpoints are of critical importance since they are the entry points to sensitive and private parts of the web application. Apart from implementation-based security vulnerabilities [1], login pages as public-facing endpoints are inherently unprotected, since anyone can access the login page and attempt to bypass the authentication. Particularly for the authentication endpoints intended for a web application's administrator, there is no reason for them to be accessible (and therefore targetable) by all visitors of that web application.

With the weaknesses of credential-based authentication being well-documented (including users choosing low-quality passwords and reusing them across different websites [2,3,4]), alternative and/or supplementary authentication schemes have been

proposed and adopted. Although such supplementary authentication techniques tend to improve certain usability and security aspects of web authentication, they have not managed to sufficiently and effectively address the issues surrounding passwords [5,6,7,8]. Thus, credential-based authentication still remains the primary, and oftentimes, sole means of authentication for the majority of web applications.

This combination of easily-accessible authentication endpoints *and* weak or reused passwords has been abused extensively by credential stuffing bots, with estimates as high as one in five of all authentication requests coming from malicious automated systems [9]. In credential stuffing attacks, attackers obtain username, password pairs from third-party sources (most commonly products of breaches [10]) and employ bots to test these credentials by injecting them in the respective fields of a web application's login page. Given the elevated privileges of administrators, it follows that administrator accounts are very often primary targets in such campaings [11], with examples such as the Uber 2016 data breach that lead to over 57M personal details of Uber users being leaked after a credential stuffing attack targeted the company's administrators and developers [12].

Recognizing the unprotected nature of public-facing login endpoints, and considering the extent of bot-driven credential stuffing attacks against administrator accounts, in this paper, we introduce PageKnocker, a novel deception-based supplementary authentication mechanism on top of passwords, which protects public-facing authentication endpoints. The authentication scheme of PageKnocker is based on request sequences, meaning that a user is granted access to the login page by making a predefined user-specific sequence of requests to the web application. For example, in order to access the `https://example.com/admin` authentication endpoint, PageKnocker enforces that a user must first request `https://example.com/about` followed by `https://example.com/categories/` and `https://example.com/news`, all of which are preexisting pages on the protected web application, which are turned into page-knocking sequences.

PageKnocker is inspired by the widely known network security concept of port knocking [13]. By employing PageKnocker, we capitalize on the familiarity of administrators with the web application - specifically with its User Interface (UI) - and we manage to effectively protect public-facing login pages, while allowing for a usable system that requires no special software or hardware. Furthermore, PageKnocker works as a gateway to deception, by introducing a deceptive layer to the authentication, enabling effective active defenses and attacker behavior analysis, all while protecting exposed endpoints of the web application. This paper makes the following original contributions:

- We introduce PageKnocker, a novel application-agnostic deception-based authentication mechanism based on request sequences, aimed at limiting access to web-application authentication endpoints.
- We implement PageKnocker using two different deceptive techniques related to what attackers see when trying to illicitly access a protected endpoint.
- We evaluate PageKnocker using both honeypots in the wild (collecting over a million requests from over 21,000 distinct IP addresses) as well as through a user study with security experts as participants.

To inspire additional research in the area of deception-based defenses, the source code along with a video demo of PageKnocker are available at `https://pageknocker.github.io/`.

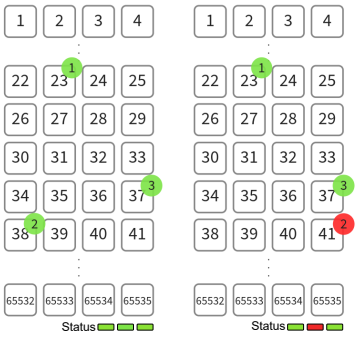

Fig. 1: *Successful (left) and unsuccessful (right) port knocking completion on arbitrary port knocking sequence* {23,38,37}. *Numbered square represent the system ports, while numbers on the upper-right edges of the squares represent the order of the knocks.*

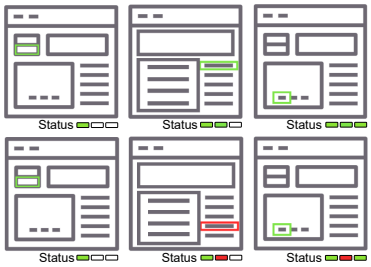

Fig. 2: *Successful (top) and unsuccessful (bottom) page knock completion on abstract webpage example. Green contours represent correct knocks, while red ones represent incorrect ones.*

2 Background

PageKnocker draws inspiration from the network-security concept of "port knocking" [13]. Port knocking is employed as an additional authentication method, to allow only authorized clients to reach a specific network port of a remote server. Namely, port knocking allows access to a client only if it has sent packets to a secret predefined sequence of ports of the server.

Taking the SSH daemon as an example, TCP port 22 is by default closed to all connecting clients on a server employing the port-knocking mechanism. If a client sends a sequence of packets to other ports on that server in the correct order, that client is considered authorized and will now be able to connect to the SSH daemon listening on port 22 (that port remains closed for everyone else except that specific client). Figure 1 shows a high-level visual representation of the port-knocking concept in action. For an arbitrary port knocking sequence consisting of the ordered set of ports {23,38,37}, a client is authenticated if they send packets to port 23, then to port 38, and finally to port 37. The left part of Figure 1 represents a successful port knocking attempt, while the right part depicts a failed authentication attempt. Once a client successfully completes the port-knocking authentication, it can now establish a connection to that network service and proceed with traditional password-based/key-based authentication.

In the system described in this paper, we employ the same logic where users are allowed to reach the public-facing login page of the web application, only if they have previously visited the correct URLs of the web application in the correct order. We underline that, while PageKnocker is inspired by the concept of port knocking, the two are completely different in design and implementation, primarily due to being active at different layers of the OSI model (application layer vs. transport layer).

3 Design

PageKnocker is a supplementary authentication mechanism on top of passwords that protects the public-facing login page of critical web applications. It operates by evaluating a sequence of requests to the web application against predefined user-specific sequences of requests. The user chooses their own page knocking sequence, in the same manner that they would choose their passwords, and they are allowed to reach the login page of the web application, only if they recreate that sequence.

3.1 Page Knocks

We define a page knock as a sequence of GET requests to the web application. These requests can be arbitrary but they work best when they are on contiguous (i.e. sequentially reachable) pages, so that users can conveniently navigate through them. To authenticate, the user needs to make a specific series of requests to the web application in a particular order. If this request sequence matches the user's predefined page knocking sequence, they are allowed to proceed (e.g. be allowed to access the web application's login form). If web clients try to visit the PageKnocker-protected authentication endpoint *before* finishing their page knock, PageKnocker can either deny them access, or engage them in cyber deception.

In the context of page knocks, matching a user's request sequence to a predefined page knock refers to finding a subset of the user's requests, which includes the predefined knocks in the correct order. For example, provided a predefined page knocking sequence: (A_2,A_5,A_{17}) where A_i represent pages of the web application, for $i=1,...,n$, with n being the total number of available pre-authentication pages of the web application, $(A_1,A_3,A_2,A_5,A_{17},A_2)$ is a correct match, while (A_2,A_1,A_5,A_{17}) is not. Putting page-contiguity aside, the possible combinations of a page knocking sequence of length N is n^N.

In this manner, a PageKnocker-protected web application is able to distinguish between legitimate users and attackers even before they reach the web application's login form. Thus, PageKnocker allows us to adjust our defenses based on a priori information and effectively protect the main authentication of the web application. Fig. 2 shows a visual representation of how users interact with a PageKnocker-protected web application in the context of a three-page knock. The length and specific pages of a knocking sequence can be selected in any manner of way, e.g., capturing the actions that a user already performs when they visit a webpage before they attempt to authenticate.

Since the strength of a page-knocking sequence is tied to the number of contiguous, pre-authentication pages that are available to users, we evaluate the number of such pages on different types of popular web applications. Using the Tranco dataset [14], we collect all the pages of the most popular websites which make their sitemap available and categorize them based on their target industry. Since page knocks are made up of contiguous pages, we divide the collected pages by path depth, i.e. *example.com/foo* has a depth of one, *example.com/foo/bar* has a depth of two, etc. We present the number of pages per path depth of the website categories along with their average in Fig. 3. We observe that on average, websites contain upwards of 1,000 webpages for a path depth between two and four. This demonstrates that there is a significant number of webpages that may be employed as parts of a page knock sequence. Moreover, considering the pages of other path depths as well, we confirm the substantial number of possible page knocking sequence combinations for all website categories.

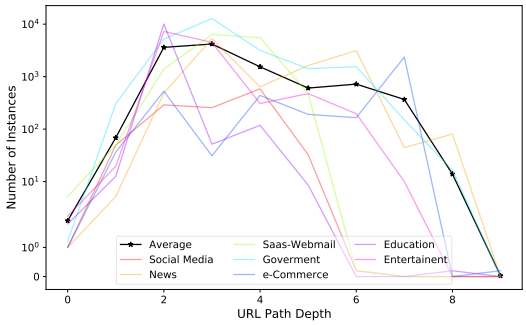

Fig. 3: *URL Path Depth for Different Website Categories*

3.2 Applicability and Deception Strategies

PageKnocker's primary objective is to protect the public-facing login page of web applications by introducing an additional layer of authentication to the login endpoint that is invisible to web clients. Protecting the login page is of critical importance, since the vast majority of login pages are public-facing and accessible by anyone, leaving them entirely unprotected against brute force and credential stuffing attacks. Table 4 of the Appendix lists the predefined login page paths for administrators and backend users for the top ten Content Management System (CMS) based on their market share. We observe that for the vast majority of CMS applications (over 70% of the market share), the publicly available predefined login page paths allow anyone to reach a web application's sensitive login form by merely requesting a well-known path.

PageKnocker allows web applications to differentiate between legitimate users who are attempting to login vs. attackers who have stolen credentials or are about to start brute-forcing their way into the application. Given an incomplete pageknock, web applications can deny access to the login form, request more information in the form of Multi-Factor Authentication (MFA) and the solving of CAPTCHAs, or engage users in deception. In this paper, we focus on the last approach by implementing support for completely hiding a login form (i.e. giving users an HTTP 404 error) or allowing them to reach a login form but always denying them access, even if attackers provide correct credentials. The former strategy can be used to confuse attackers about the nature of the web application (in relation to Table 4 of the Appendix), whereas the latter keeps them engaged in order to study their authentication attempts and protect other systems from attacks.

3.3 Threat Model

PageKnocker can protect all publicly-accessible authentication endpoints from attackers launching brute-force attempts or operating with stolen credentials. While all authentication endpoints are in scope for PageKnocker, it is best suited to protect endpoints which are intended to be used by a small number of administrators and are the ones primarily targeted by web bots [11].

Even though PageKnocker is situated at the server side and therefore completely invisible to benign and malicious clients, we operate under the assumption that an attacker may have knowledge of the existence of page knocks, but no knowledge of the specific sequences chosen by users.

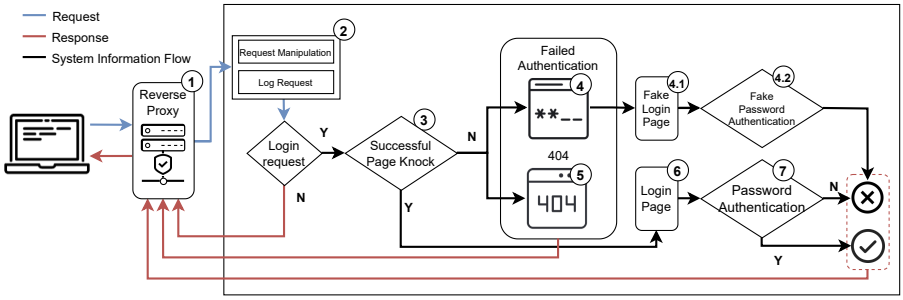

Fig. 4: **PageKnocker** *architecture with request and response flows*

3.4 Security Properties

Here, we briefly describe a number of non-obvious security properties of PageKnocker that differentiate it from existing authentication countermeasures.

- **No special software/hardware necessary.** PageKnocker does not require any special software or hardware. The organic clicks on a web application's UI are converted into page knocks and used to protect authentication endpoints.
- **Resistant to shoulder surfing.** A user's page knocks on a PageKnocker-protected web application are indistinguishable from regular browsing. As such, PageKnocker increases the complexity of a shoulder-surfing attack, which now necessitates both the correct page knocks, as well as the credentials.
- **Invisible to MITM.** Unlike traditional port knocking, page knocking happens at the application layer and inherits all the confidentiality and integrity guarantees of TLS.
- **Gateway into deception.** PageKnocker can be employed alongside other technologies and, even before attackers send their first authentication attempt, transparently engage them in deception, i.e., route their traffic to different servers, serve them content containing honeytokens, etc.
- **Disincentivizing credential reuse.** Since PageKnocker turns a web application's own UI into a means of authentication, it makes it difficult for users to reuse credentials across unrelated websites (e.g. users cannot use the same page-knocking sequence for a blog and an online bank, even if they choose the same password).
- **Compatible with existing authentication mechanisms.** Along with protecting the login page of a web application, PageKnocker can be used in parallel with existing supplementary authentication mechanisms, such as MFA and Single Sign-On (SSO), thus enhancing the application's overall security stance.

4 Implementation

In terms of PageKnocker's implementation, our objective is for the system to be lightweight, application-agnostic, and capable of resisting real attackers, without being cumbersome for benign users.

4.1 Overview

Below, we outline the main functionality of the proposed framework, along with the main components that address these functions.

Containerization PageKnocker employs Docker containers for every component of its architecture, allowing us to easily interchange parts of the system, without disrupting the remaining components.

Server-side request manipulation Given that our system needs be able to monitor and manipulate requests and responses to the web application, we employ mitmproxy [15]. We place the `mitmproxy` in front of the web application, allowing us to intercept, inspect, and manage the HTTP requests to the application server, and manipulate the HTTP responses that the web application sends to the user.

Reverse Proxy We employ an Nginx reverse proxy, to manage the TLS certificate of the web application's domain, handle TLS termination, and forward the traffic to the `mitmproxy` and web application docker containers. This reverse proxy is placed between the client and the dockerized containers of the web application.

4.2 Request-Response Flow

Fig. 4 shows PageKnocker's architecture, along with the request and response flows. Below, we elaborate on each step of these flows.

User Request - Reverse Proxy ①. A client request to the web application is first delivered to the Nginx reverse proxy. The reverse proxy handles the TLS termination and forwards the request's URI to the host of the web application. It is the reverse proxy that manages the domain redirection and manages the TLS certificates.

Server-side Request Manipulation ②. After the TLS termination of the Nginx reverse proxy, the request reaches `mitmproxy`. Regarding the received requests, we use `mitmproxy` in two ways. First, we log the incoming request, along with the IP address of the client, the cookie for our web application, and the request method, path, and timestamp. We then evaluate whether it is a login request by comparing the request's endpoint with the known authentication path of the protected web application (popular endpoints shown in Table 4 of Appendix). This is a configuration option that must be provided the first time PageKnocker is deployed to protect a web application. Non-login requests are ignored by PageKnocker, whereas login ones are processed according to the next steps. We note that the entirety of the pageknock-authentication procedure takes place in the mitmproxy module, before a request reaches the web application.

PageKnocker and Login Environments ③. Having established that a visitor is requesting access to the web application's login page, we evaluate whether they have completed their page knocking sequence. PageKnocker consults the logs that it collects for each request made to the web application, identifying each visitor by existing web-application cookies or by IP address, depending on the deployment configuration. We then associate to each user the requests that they make to the web application sorted by their timestamps. Thus, we have time-ordered request sequences for all visitors to the web application, and we can evaluate if a user has managed to successfully complete the predefined page knock. We only authenticate users whose page knocks were matched against their predefined knocking sequence within a configurable window of time (set to 10 minutes for our prototype). This means that a correct page knock is only valid for a short period of time, and needs to be repeated after this time expires.

Dealing with page knock violations. Given that PageKnocker's authentication scheme provides information on the nature of the user before they reach the login page, we are able to adjust the response of the system. In this context, we develop

two different deception-based login environments, which are triggered by a failed page knock authentication; a Failed Authentication (FA), and an HTTP 404 environment. A user that has failed the page knocking authentication, is redirected to one of these two environments, depending on the deployment configuration, upon requesting to access the actual login page.

Failed Authentication ④. The Failed Authentication (FA) login environment looks exactly like the real login page of the web application. The deceptive element of this environment lies in the functionality of the main authentication. In particular, we circumvent the credential (username/password) authentication of the web application using mitmproxy, and we deny access to the user, whatever credentials they may provide; even correct ones. Specifically, this is achieved by manipulating the web application response, when the predefined page knocking sequence has not been matched. An attacker that has failed to complete the predefined page knocks, is directed to a page that is a clone of the authentic login page (4.1), mimicking the wrong-credentials error message of the protected web application, regardless of which credentials the attacker uses (4.2) . Attackers who are unaware of PageKnocker will eventually conclude that the stolen credentials were not valid (even if they really were) against the protected web application and stop attacking. Sophisticated attackers who are certain about the validity of their credentials may eventually infer the presence of a PageKnocker-like defense system leaving them with unclear next steps. An attacker who keeps engaging with a PageKnocker-protected web application risks getting studied, fingerprinted, and potentially engaged in additional deception.

Page Not Found (HTTP 404) ⑤. When attackers who have not completed the predefined page knocks request the login page, PageKnocker responds with a HTTP 404 page. The goal of this deceptive strategy of PageKnocker is to confuse attackers regarding the nature of the attacked web application. Bots that engage in fingerprinting to identify the nature of a web application (so that they can attack the right authentication endpoint) may conclude that the fingerprinting was faulty and move to another target. We report on the behavior of bots when encountering a PageKnocker-protected web application in Section 5. As before, sophisticated attackers attacking a specific target may again conclude that the web application is protected by PageKnocker and is therefore engaging them in deception. Attackers who choose to keep attacking the target, do so knowing that all information that they have discovered regarding that web application may be part of a deception strategy aimed at studying them and keeping them engaged.

Login Page ⑥. The FA and 404 login environments are triggered by a request to the login page from a user who has not completed the predefined page knocks. A user that has completed the predefined page knocks is authenticated, as per the page knocks, and is therefore allowed to access the true login page and the true password-authentication component of the protected web application ⑦, without any further interventions/content modifications by PageKnocker.

5 Security Evaluation

In this section, we report on a three-month long experiment where we deployed PageKnocker-protected web applications on the public web, to understand how bots interact with them. In the next section (Section 6) we present a separate user study

Table 1: *Requests received in the wild (2 honeypots per page-knock length)*

Page Knock Length	Total	GET [%]	POST [%]
Short	138,919	17.19	82.15
	387,749	59.15	40.71
Medium	133,886	19.00	80.61
	128,826	42.32	57.28
Long	129,963	39.50	60.09
	128,443	19.78	79.79
Overall	1,047,786	32.82	66.77

(a) *Requests to the six FA honeypots*

Page Knock Length	Total	GET [%]	POST [%]
Short	21,387	94.07	4.52
	21,823	96.54	1.99
Medium	27,712	97.37	1.46
	17,028	94.52	2.55
Long	22,390	93.84	4.75
	17,918	94.90	2.14
Overall	128,258	95.21	2.90

(b) *Requests to the six 404 honeypots*

simulating both sophisticated attackers with knowledge of PageKnocker's presence, as well as characterizing the usability of our system.

To evaluate our approach against bot attacks, we deployed 12 WordPress honeypots secured by PageKnocker using different page knock pages and knock lengths. Half of the honeypots engaged attackers in failed-authentication (FA) deception (all credentials attempted by non PageKnocker-authenticated users are always wrong) and the other half presented unauthenticated attackers with our 404 (Page Not Found) approach.

For the six FA deployments, we specify two with a page knocking sequence of a single request (which we will refer to as "short" page-knock length), two with a sequence of three requests ("medium"), and two with a sequence of five requests ("long"). We establish page-knocking sequences of the same lengths for the six 404 honeypots. Lastly, we note that every predefined page knocking request sequence was unique among all of the aforementioned deployed honeypots.

5.1 Experimental Setup

All 12 versions of the page-knocking-adapted web applications were deployed on the public web for a period of 90 days on newly registered domain names. We associate each request sent to PageKnocker with the IP address of the web client. The page knocking authentication is therefore carried out by identifying the page knocks that each individual IP address makes to the web application. To ensure that our honeypots are comparable to real-world deployed web applications we used Let's Encrypt [16] to obtain valid TLS certificates for all of them. An additional advantage of using TLS certificates is that, in addition to IP-address-scanning web bots, the announcement of our certificates on Certificate Transparency (CT) [17] is also likely to attract CT bots [18].

5.2 Results

In the span of three months, our deployed honeypots received 1,176,044 requests from 21,935 unique IP addresses, with 1,047,786 (89.09%) and 128,258 (10.91%) requests made to the FA and the 404 honeypots, respectively. Specifically towards authentication endpoints, the deployed honeypots received a total of 790,541 requests (67.22 % of all requests), with 761,909 (96.38%) requests towards the FA honeypots, while only 28,632 (3.62%) directed towards 404 honeypots. Regarding the effective security of PageKnocker, out of the total of 21,935 unique IP addresses, we observe only two cases of correct page-knock guesses (0.0091%).

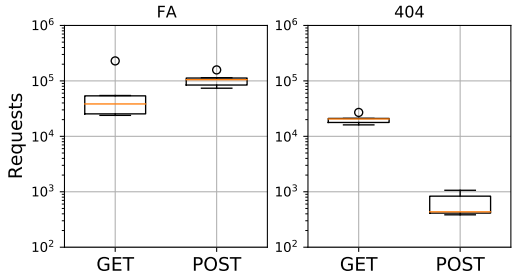

Fig. 5: *Distribution of GET and POST requests in FA and 404 honeypots in the wild*

Overall, across all deployed honeypots, we observed large discrepancies in the number of received requests between FA and 404 configurations. In order to examine these divergences, we evaluate the received requests with respect to their GET and POST distributions.

PageKnocker Authentication Robustness Regarding the two correctly guessed page knocking sequences, in both cases, the traffic was directed towards a honeypot with a predefined page knock sequence of a single request (the easiest possible scenario). These originated from two different IP addresses, namely 34.76.104.x and 34.140.32.x which sent requests to just one of our 12 honeypots exhibiting clear web-scanning behavior. They both initially made a request for a `robots.txt` file, and subsequently proceeded to consecutively request front end pages of the web application. Both were eventually redirected by the web application to the login form, when attempting to access privileged content. The fact that they managed to clear the single request page knock was due to pure chance, since they managed to knock on the correct path before landing on the login form.

Given the rarity of this event and the fact that a single request page knock represents the least secure scenario that we evaluate, we consider these cases to be statistically insignificant (two cases out of 21,935 unique IPs, which is 0.0091%). We can therefore conclude that PageKnocker successfully protected the deployed web applications from almost 800K illicit accesses.

GET and POST Requests Tables 1a and 1b show the GET and POST requests to the FA and 404 honeypots respectively. On both tables, we present the requests sent to each honeypot, along with the predefined page knocking sequence length associated with each deployed honeypot. By observing the distribution of GET and POST requests, it becomes apparent that the POST requests are significantly fewer for the 404 honeypots. This is due to the fact that the FA login environment allows the unauthenticated user to access and interact with the login page where bots can keep trying credentials (submitted via POST requests) and keep getting wrong-credentials errors. Contrastingly, in our 404 honeypots, PageKnocker makes the login page unavailable so that the vast majority of bots stop their scanning attempts much earlier. Figure 5 shows the overall distribution of GET and POST requests across all honeypots. In accordance with the measurements of Tables 1a and 1b, we observe similar distributions for GET requests, but this accumulative distribution of requests further reveals that the discrepancy for the POST requests, approximates two orders of magnitude. We note that for clients who get a 404 error on the authentication endpoint, one would expect that the number of POST requests be near zero, since there is no page to receive the POST requests. Interestingly, we observed that that is not the case, revealing faulty application logic

Table 2: *Common bot appearances across FA and 404 honeypots*

Appearances in Different Honeypots	IPs	Only FA[%]	Only 404 [%]	Both[%]
2-4	4,582	35.49	10.30	54.21
5-6	1,696	55.66	0.06	44.28
7-9	817	0	0	100
10-12	1,496	0	0	100

for a small fraction of the malicious bots that attacked our honeypots and kept on sending POST requests, even after having knowledge that the login page was not there. The tables breaking down the distribution of GET/POST requests just towards the authentication endpoint are available in Tables 5a and 5b (in the Appendix).

Common bots across honeypots Since all 12 honeypots were in operation at the same time, we can identify the specific bots that targeted more than one honeypot, with a particular focus on the ones that targeted multiple honeypots across deception types. In this way, we can compare the strategies of specific bots when encountering our failed-authentication (FA) deception honeypots vs. the 404 ones.

In Table 2, we present the number of IP addresses that make requests to multiple honeypots, along with the distribution of appearances towards only FA, only 404, or both types of honeypots. As an example, we observe that there are 1,696 bots (characterized by their unique IP addresses) which make requests towards 5 to 6 independent honeypots, with 55.66 % of them directing traffic towards only FA honeypots, and 0.06 % directing traffic only to 404 honeypots. Since we have six FA and six 404 deployed honeypots, bots making requests to more than six honeypots are, by definition, reaching both types of honeypots. This is also the reason behind the selection of presented ranges of honeypots.

Overall, from Table 2, we observe that there are significantly fewer bots that make requests only towards 404 honeypots, compared to the ones making requests only towards FA, or towards both honeypot types. This means that bot interaction with FA or both types of honeypots can potentially result in a larger accumulation of bot traffic around those honeypots. We suspect that bots visiting FA honeypots do not register any abnormal behavior on the part of the web application (covert deception), while bots that visit the 404 honeypots recognize the misbehavior of the login page (overt deception) and, in some manner, flag the web application as non-conforming for future reference. It is therefore likely that FA honeypots are added to databases of potentially vulnerable websites that can be attacked again in the future, either by the same attacker or by other attackers through data exchanging. Contrastingly, the non-conforming behavior of PageKnocker's HTTP 404 honeypots may be precluding them from lists of potential targets and thereby attracting overall fewer unique attackers, compared to the FA honeypots. Focusing on the bots (as identified by their IP addresses) that appear in both types of honeypots, we present in Fig. 6 a scatter plot depicting the GET and POST requests made from bots that visit all deployed honeypots. The vertical axis represents the requests made to 404 honeypots, whereas the horizontal axis represents the requests towards a failed-authentication (FA) honeypot. A data point above $y=x$ indicates that there are more requests to a 404 than a FA honeypot, while the opposite is true for data points below that line. First, we observe the large accumulation of GET requests at the region of fewer than 50 requests (for both the FA and the 404), whereas the POST requests are concentrated along the horizontal axis. To extract insights from this data, we provide the Linear Regressions (LRs) for both the GET and POST requests. The LR curves demonstrate the following:

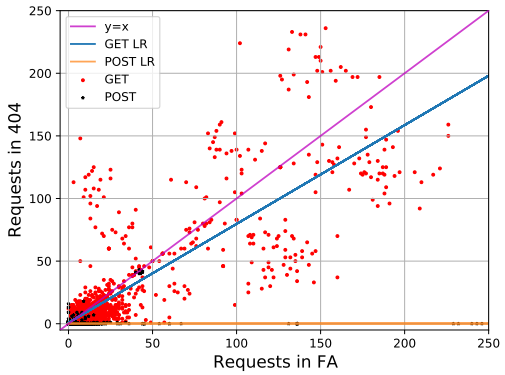

Fig. 6: *GET and POST requests of common IPs in FA and 404 honeypots in the wild*

- Since the slope of the GET-requests LR curve is relatively close to that of the $y=x$, we can conclude that the GET requests between the FA and the 404 honeypots do not present a significant difference. This is to be expected from our prior experimentation, given that GET requests essentially represent reconnaissance or discovery attempts by bots, which we do not interfere with our PageKnocker mechanism.
- On the other hand, the POST-requests LR curve is practically parallel to the horizontal axis, meaning that the overwhelming majority of POST requests is directed towards FA honeypots. This matches our previous results (see Fig. 5), where we showed that the 404 deployments receive fewer POST requests than the FA ones, to the difference of two orders of magnitude. Thus, we observe that this result holds when considering common clients that appear in both FA and 404 honeypots.

We note that in Fig. 6 we also include nine IPs that demonstrate very large numbers of POST requests, which fall outside the confines of the presented plot. This was a conscious choice for ease of viewing, and we present these data points in Table 6 (Appendix). Taken together, these observations show that, for the vast majority of bots, PageKnocker triggered different paths of their attack logic. That is, given the deceptive "wrong-credentials" message, bots keep sending POST requests with additional credentials. The same bots, however, when encountering our 404-deception honeypots stopped attacking our honeypots. System administrators can therefore choose between different deception modes of PageKnocker depending on whether they want bots to go away as fast as possible (404 deception) or keep them engaged in order to study them, without any danger of successfully authenticating to the targeted web application (failed-authentication deception).

Time-based Analysis of Requests In this section, we evaluate the behavior of bots in terms of their successive requests towards our PageKnocker honeypots. As we established in our prior analysis of GET and POST requests, bots employ GET requests as part of their reconnaissance and proceed with POST requests when attempting to brute-force the application's authentication endpoints. Since PageKnocker disrupts this sequence of events, we examine how the bots react to our mechanism in terms of what requests they make. In Fig. 7, we present the distribution of request methods (GET, POST) for the first 1,000 bot requests towards our FA and 404 honeypots. The choice of the first 1,000 requests stems from the fact that clients making at least 1,000 requests approximate the totality of our dataset, as we show in Fig. 8 (Appendix).

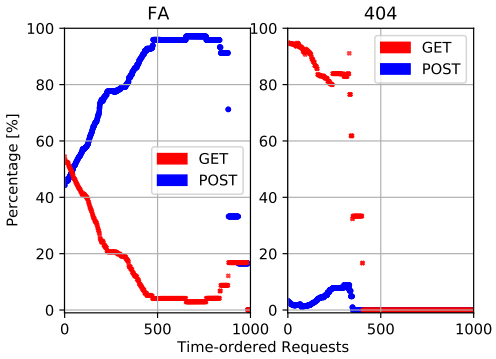

Fig. 7: *Distribution of request methods in time for FA and 404 honeypots in the wild*

In Fig. 7, we observe the average percentage of GET/POST requests to PageKnocker honeypots evaluated over the first 1,000 time-ordered requests. The horizontal axis represents the time-ordered requests, which are made by all bots interacting with the respective honeypots, meaning that a value of 200 on the horizontal axis represents the 200^{th} request that a bot makes. Thus, we observe that for FA honeypots, bots begin their interaction with slighlty more GET requests, and after the 100th request, on average, the POST requests have clearly surpassed the number GET requests. This indicates that the bots have transitioned from reconnaissance to actual attack. This POST-majority pattern continues for the better part of the first 1,000 requests, and only presents a relatively small disparity, as we approach the last requests.

Regarding the 404 honeypots, we notice diametrically opposite results. The GET requests represent the majority, being well above 80% of the total requests for the first 200 requests. Furthermore, we notice that after initially attempting relatively few POST requests, after the 400^{th} request, both GET and POST requests drop to zero. This reveals that the majority of bots are directed away from the application, with just a few bots with faulty application logic attempting to send POST requests despite not finding the actual authentication endpoints.

6 User Study

In Section 5 we evaluated PageKnocker's ability to protect web applications against non-targeted attacks by malicious bots. In this section, we report on a study involving real users to understand i) whether they can defeat PageKnocker when they are told about its existence (simulating attackers), and ii) their ability to choose, recall, and use page knocks when interacting with a PageKnocker-protected web application (simulating web application administrators).

6.1 Setup of study

In order to best approximate the roles for both scenarios (attackers and administrators), we recruited 16 users from our computer science department (graduate students and faculty) who work in the area of cyber security and are knowledgeable in the subject of attacks and authentication. In contrast with the 22K unique bots that attacked our

PageKnocker-protected honeypots in Section 5, in this user study we consciously trade attack volume in favor of attacker sophistication.

The user study setting comprises of two parts addressing the aforementioned objectives. For the first part, the participants are introduced to the PageKnocker mechanism and its functionality. Subsequently, the participants are asked to carry out the two following tasks:

1. Each participant is directed towards a deployed web application that uses a predefined page knocking sequence of a *single* request. Their task is to figure out the single request, which is necessary to match the predefined sequence, and eventually log in to the web application. The participants have 10 minutes to complete this task.
2. The second task is the same as the first, with the exception that the predefined page knocking sequence contains *three* requests.

We note that for both tasks, the deployments are exactly the same as the in-the-wild deployed honeypots, and the participant pool was equally split between FA and 404 login environment implementations. The participants are also provided with correct login credentials, which lead to a webpage informing them of a successful task completion.

For the second part of the user study, having knowledge of PageKnocker's functionality and first-hand experience of the implementation, the participants are asked to create their own page knocking sequence. Once they submit their page knocks, they are asked three times in the two subsequent weeks after their submission, to re-visit the web application, and perform their page knock. We advised the participants to act like legitimate users of the PageKnocker-protected web applications and consider security as well as usability aspects in their selection of personal page knocks, based on their theoretical understanding and empirical experience with the framework.

6.2 Ethical Considerations

After the review of our user-study protocol by our IRB, we were informed that there was no requirement for approval or exemption, since we only collected the paths visited in our deployed web applications which contain no private, user-specific information. The participants were asked to volunteer, in the context of a security seminar, and each of them was entered in two raffles for two $50 gift cards, contingent upon their participation in the respective part of the user study.

6.3 Results

Regarding the first part of the user study, out of the 16 participants, only a single one managed to break the PageKnocker-authentication for the single-request instance, while none were able to break the three request sequence instance. This clearly demonstrates that, even though attackers may be able to brute-force their way through extremely short page knocking sequences, correctly guessing longer sequences is far from trivial.

In Table 3, we present the results of the second part of our user study. As a user identifier, we use an arbitrary index number, and for each user, we provide four metrics: the completion results of their personally submitted page knocking sequence upon their three returns in the two subsequent weeks, and the respective lengths of the freely-chosen, page knocking sequences. Two of the participants (Users #10 and #11) did not return for the page knock recall part of the study, while another two (Users #8 and #9) returned only once. We note that for the latter case (Users #8 and #9), their returns, although not completed fully, were successful, indicating that the users correctly

Table 3: *User Study Usability Results*

User	Day 1	Day 2	Day 3	Knock Length
1	✗	✗	✗	2
2	✓	✓	✓	3
3	✓	✓	✓	3
4	✓	✓	✓	3
5	✓	✓	✓	3
6	✓	✓	✓	3
7	✓	✓	✓	3
8	✓	-	-	3
9	✓	-	-	3
10	-	-	-	3
11	-	-	-	3
12	✓	✓	✓	4
13	✓	✓	✓	4
14	✓	✓	✓	4
15	✓	✓	✓	4
16	✓	✓	✓	5

remembered and reproduced their page knocks at least once. Out of the 12 participants that completed the return study in its entirety, 11 (91.67%) of them consistently completed their page knocks correctly for all three return days. The one participant who did not (User #1) never managed to complete their page knocks correctly.

Regarding the choice of page knocks by the participants, we observe the majority opted for page knocking sequences of length three and four. All but one of the participants who participated in the second part of the study, managed to consistently reproduce their page knocks correctly. Given that the overwhelming majority of users managed to correctly remember and reproduce their page knocks, and considering that all users submitted longer page knocks, we attribute the failure of User #1 to potentially initial confusion upon submission of the personalized sequence. We note that this user still managed to reproduce half of their sequence in all return days.

From the two parts of the user study, we make the following observations:

- First, PageKnocker is resistant against expert users that have knowledge of its existence and its functionality. We experimentally confirm that longer page knocks are more secure, even if these are just 3 link clicks on a web application's UI.
- Observing the choices of page knocking sequence lengths, the vast majority of users recognize the necessary sequence length threshold to be between three and four requests.
- Lastly, we showed that PageKnocker exhibits high usability, since the participants remembered their knocks consistently.

7 Limitations

Given the ever-increasing sophistication of cyber attacks, PageKnocker is not a silver bullet against all possible authentication attacks. That is, while PageKnocker can protect web applications against credential brute-forcing, guessed/reused passwords, and traditional phishing, it cannot defend against the MITM phishing kits where attackers can observe all traffic from a user's browser and replay it (i.e. clicks, typed credentials, MFA codes, etc.) to the victim web application [19]. Like with any real-world defense mechanism, PageKnocker would have to be layered alongside other countermeasures (i.e., anomaly detection systems), to defend users against these types of attacks.

An attacker that has knowledge of the presence and functionality of PageKnocker, could attempt a brute-forcing strategy to find the correct page-knocking sequence. Such a strategy would consist of performing arbitrary page knocks and subsequently visiting the login page, to evaluate each page knocking sequence. Excluding the fact that such a brute force attack can only be carried out against the 404 deceptive configuration, where the attacker can evaluate the correctness of each attempted page knocking sequence based on the response of the web application (404, or login page), such back-and-forth actions would need to be carried out repeatedly, potentially hundreds of thousands of times. The repetitive nature of this attack (page knock paths to login page loop) constitutes an extremely distinct client behavior, which could easily be detected by the server.

Although adapting PageKnocker to protect existing web applications is a relatively uncomplicated procedure, as described in Section 3.2, the different in-page structures and a web application's content-update frequency, could have an impact on the efficacy of the system (i.e. a regularly updated blog or a news website with many of the webpage paths changing often). To counterbalance this, we note that PageKnocker can still protect the authentication of such web applications, using a smaller subset of the webpage's sitemap; i.e. the pages that do not change.

8 Related Work

Deception-based Defenses The field of deception-based defenses comprises of all security techniques aiming to protect digital elements through redirection, diversion, or the deflection of attacks, by deceiving the perpetrator. In this context, honeypots have been integrally linked with deceptive security, providing the aforementioned qualities, while also allowing further analysis of an attacker's behavior by monitoring and tracking intruders. Although the deceptive defenses predate them [20], honeypots as virtual entities attracting and engaging attackers have been on the forefront of network and web security for the better part of the last 20 years [20,21]. In order to tackle diverse security issues, the concept of honeypots has been extended to analogous elements, such as honeypot networks (also known as honeynets) [22], honeytokens [23], honeywords [24], and decoy documents and files [25,26,27,28], overall characterized as honey-x entities. Such honey-x approaches have been employed in different security applications, from intrusion and malware detection [25,29,30], to software and web application defenses [31,32,33,34]. Introducing believable and attractive decoy and bait elements in the system enhances the detection capability, while the different deployments and characteristics of trap-based architectures allow for ad hoc deceptive defenses addressing various attacks [30,35,26,36].

Deception & Web Applications In the context of web applications and web servers, deception-based approaches have focused on augmenting the attack surface of the web application by introducing deceptive elements. This concept is founded on the idea of using deceptive elements as detection points, where any interaction with them is labelled suspicious. More specifically, strategies such as providing falsified system version information, tampering with *robots.txt* files, generating virtual honey documents, cookie scrambling, and Javascript obfuscation have been attempted [37,38]. Furthermore, approaches generating decoy files in lieu of tripwires deployed on real systems have proven effective for intruder detection, while achieving low rates of false positives [27,26]. On the other hand, deceptive techniques have also been employed in a parallel world context, where the attackers are covertly redirected to an isolated and appropriately

curated clone of the original environment after detection [33,39,40]. Thus, the attacker is under the impression that they are still attacking the real system, while the defender is able to observe the attacker's behavior, without the risk of lateral movements or exposure of critical resources. While PageKnocker's main objective is to protect the authentication endpoint, it also serves as a detection mechanism, and may be employed as a trigger for a parallel world setting.

Regarding the authentication itself, in order to alleviate the inadequacies of password-based web authentication, deception-based approaches, such as login rituals and web tripwires have been suggested as supplementary authentication mechanisms [41]. In the case of login rituals, users are considered "truly" authenticated by performing specific actions post credential-based authentication, e.g., interact with specific elements of the web application. Tripwires are elements that authenticated users must avoid to remain authenticated. PageKnocker also uses web-application elements as deception primitives but brings these to pre-authenticated users. In this way, we can protect web applications from tens of thousands of would-be attackers (Section 5) *before* they even reach the login page. Provided the improved defensive posture afforded by both overt and covert deceptive defenses [42], PageKnocker employs two deceptive environments; one manifestly deceptive and one clandestine.

9 Conclusion

In this paper we introduced PageKnocker, a deception-based supplementary web authentication mechanism on top of passwords that can protect public-facing authentication endpoints from attackers. Inspired by the concept of port knocking, PageKnocker checks the requests that a user makes towards the web application against a predefined user-specific sequence of requests. The user is authenticated and allowed to reach the login page only if the sequence of requests they make matches their predefined sequence. The objective of this supplementary authentication scheme is to protect the login page of web applications, while allowing the administrators to detect attackers *before* they reach the credential-based authentication. Utilizing this a priori knowledge on the nature of the visitor, we construct two different deceptive login environments, towards which we direct the unauthenticated attacker; the first directs the user to a fake login page, while the second directs them to an HTTP 404 error. We evaluate the security and usability of PageKnocker using both in-the-wild experiments with honeypots as well as via a user study. From these experiments, we conclude that PageKnocker manages to effectively protect the main authentication of the web application, while being user-friendly. Furthermore, we demonstrated that overt placement of deceptive elements leads to bot traffic disengaging quicker, while concealing such elements encourages further bot interaction.

Acknowledgements We thank the anonymous reviewers for their helpful feedback. This work was supported by the Army Research Office (ARO) under grant W911NF-24-1-0051.

Availability. A video demonstration of PageKnocker along with source code are available at https://pageknocker.github.io/

References

1. Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Measuring login webpage security. In *Proceedings of the Symposium on Applied Computing*, 2017.

2. Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.
3. Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, 2007.
4. Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*.
5. Joshua Reynolds, Nikita Samarin, Joseph Barnes, Taylor Judd, Joshua Mason, Michael Bailey, and Serge Egelman. Empirical measurement of systemic {2FA} usability. In *29th USENIX Security Symposium*, 2020.
6. Thanasis Petsas, Giorgos Tsirantonakis, Elias Athanasopoulos, and Sotiris Ioannidis. Two-factor authentication: is the world ready? quantifying 2fa adoption. In *8th European Workshop on System Security*, 2015.
7. Xu Lin, Panagiotis Ilia, Saumya Solanki, and Jason Polakis. Phish in sheep's clothing: Exploring the authentication pitfalls of browser fingerprinting. In *31st USENIX Security Symposium*, 2022.
8. Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *15th Symposium on Usable Privacy and Security (SOUPS 2019)*.
9. F5 labs 2023 identity threat report: The unpatchables. https://www.f5.com/labs/articles/threat-intelligence/2023-identity-threat-report-the-unpatchables.
10. Troy Hunt - Have I Been pwned? https://haveibeenpwned.com/, 2018.
11. Xigao Li, Babak Amin Azad, Amir Rahmati, and Nick Nikiforakis. Good bot, bad bot: Characterizing automated browsing activity. In *2021 IEEE Symposium on Security and Privacy*.
12. Ionut Ilascu. Bleeping Computer: Uber Fined for Covering Up 2016 Data Breach. https://www.bleepingcomputer.com/news/security/uber-fined-for-covering-up-2016-data-breach/, 2018.
13. Rennie Degraaf, John Aycock, and Michael Jacobson. Improved port knocking with strong authentication. In *21st Annual Computer Security Applications Conference (ACSAC'05)*.
14. Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.
15. Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010.
16. Let'sEncrypt. https://letsencrypt.org/, 2023.
17. Certificate Transparency. https://certificate.transparency.dev/, 2023.
18. Brian Kondracki, Johnny So, and Nick Nikiforakis. Uninvited guests: Analyzing the identity and behavior of certificate transparency bots. In *31st USENIX Security Symposium*, 2022.
19. Brian Kondracki, Babak Amin Azad, Oleksii Starov, and Nick Nikiforakis. Catching transparent phish: analyzing and detecting mitm phishing toolkits. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.
20. Clifford Stoll. Stalking the wily hacker. *Communications of the ACM*, 1988.
21. Niels Provos et al. A virtual honeypot framework. In *USENIX Security Symposium*, 2004.
22. The Honeynet Project. https://www.honeynet.org/, 2023.
23. Maya Bercovitch, Meir Renford, Lior Hasson, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Honeygen: An automated honeytokens generator. In *Proceedings of 2011 IEEE International Conference on Intelligence and Security Informatics*.
24. Ari Juels and Ronald L Rivest. Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*.
25. Brian M Bowen, Shlomo Hershkop, Angelos D Keromytis, and Salvatore J Stolfo. Baiting inside attackers using decoy documents. In *Security and Privacy in Communication Networks: 5th International ICST Conference, SecureComm 2009*.

26. Jonathan Voris, Jill Jermyn, Nathaniel Boggs, and Salvatore Stolfo. Fox in the trap: Thwarting masqueraders via automated decoy document deployment. In *Proceedings of the Eighth European Workshop on System Security*, 2015.
27. Malek Ben Salem and Salvatore J Stolfo. Decoy document deployment for effective masquerade attack detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 8th International Conference; DIMVA 2011*.
28. Nick Nikiforakis, Marco Balduzzi, Steven Van Acker, Wouter Joosen, and Davide Balzarotti. Exposing the lack of privacy in file hosting services. In *4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 2011)*.
29. Emmanouil Vasilomanolakis, Shreyas Srinivasa, Carlos Garcia Cordero, and Max Mühlhäuser. Multi-stage attack detection and signature generation with ics honeypots. In *NOMS 2016 IEEE/IFIP Network Operations and Management Symposium*.
30. Brian M Bowen, Pratap Prabhu, Vasileios P Kemerlis, Stelios Sidiroglou, Angelos D Keromytis, and Salvatore J Stolfo. Botswindler: Tamper resistant injection of believable decoys in vm-based hosts for crimeware detection. In *International Workshop on Recent Advances in Intrusion Detection*, 2010.
31. Younghee Park and Salvatore J Stolfo. Software decoys for insider threat. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.
32. Frederico Araujo, Kevin W Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *2014 ACM SIGSAC conference on Computer and Communications Security*.
33. Merve Sahin, Cédric Hebert, and Anderson Santana De Oliveira. Lessons learned from sundew: a self defense environment for web applications. In *Proceedings of the 2020 Measurements, Attacks, and Defenses for the Web (MADWeb) Workshop in the Network and Distributed System Security Symposium (NDSS)*.
34. Merve Sahin, Cédric Hébert, and Rocio Cabrera Lozoya. An approach to generate realistic http parameters for application layer deception. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022*.
35. Brian M Bowen, Vasileios P Kemerlis, Pratap Prabhu, Angelos D Keromytis, and Salvatore J Stolfo. Automating the injection of believable decoys to detect snooping. In *Proceedings of the 3rd ACM conference on Wireless network security*, 2010.
36. Daniel Reti, Daniel Fraunholz, Karina Elzer, Daniel Schneider, and Hans Dieter Schotten. Evaluating deception and moving target defense with network attack simulation. In *Proceedings of the 9th ACM Workshop on Moving Target Defense*, 2022.
37. Daniel Fraunholz, Daniel Reti, Simon Duque Anton, and Hans Dieter Schotten. Cloxy: A context-aware deception-as-a-service reverse proxy for web services. In *Proceedings of the 5th ACM workshop on Moving Target Defense*, 2018.
38. Daniel Fraunholz and Hans D Schotten. Defending web servers with feints, distraction and obfuscation. In *2018 International Conference on Computing, Networking and Communications (ICNC)*.
39. K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *14th USENIX Security Symposium*, 2005.
40. Vincent E Urias, William MS Stout, and Han W Lin. Gathering threat intelligence through computer network deception. In *2016 IEEE Symposium on Technologies for Homeland Security (HST)*.
41. Timothy Barron, Johnny So, and Nick Nikiforakis. Click this, not that: Extending web authentication with deception. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*.
42. Kimberly Ferguson-Walter, Maxine Major, Chelsea K Johnson, and Daniel H Muhleman. Examining the efficacy of decoy-based and psychological cyber deception. In *USENIX Security Symposium*, 2021.
43. W3Techs. Usage statistics and market share of WordPress. https://w3techs.com/technologies/details/cm-wordpress, 2023.

A

Table 4: *Predefined login page paths for top 10 CMS web applications by market share [43]*

CMS	Predefined Login Path	Market Share [%]
Wordpress	/wp-admin, /wp-login.php	63.6
Joomla	/administrator, /administrator/index.php	2.7
Drupal	?q=user/login, landing page	1.8
PrestaShop	/admin	1
Magento	/admin	0.9
TYPO3	/typo3	0.6
Craft	/admin/login	0.2
Ghost	/ghost, /ghost/#/signin	0.1
Contao	/contao, /contao/login	0.1
ProcessWire	/processwire	0.1

Table 5: *Requests received to the login page*

Page Knock Length	Total	GET [%]	POST [%]
1	131,596	10.33	89.67
1	182,182	11.41	88.59
3	133,201	16.21	83.79
3	94,108	17.93	82.07
5	95,857	16.85	83.15
5	124,965	15.31	84.69
Overall	761,909	14.67	85.33

(a) *Requests to the login page of FA honeypots*

Page Knock Length	Total	GET [%]	POST [%]
1	5,056	43.73	56.27
1	4,542	49.43	50.57
3	4,602	49.35	50.65
3	4,546	48.88	51.12
5	5,409	42.91	57.09
5	4,477	49.16	50.84
Overall	28,632	47.24	52.76

(b) *Requests to the login page of 404 honeypots*

Table 6: *Common IPs appearing in both FA and 404 honeypots making large amounts of POST requests*

IP	Requests	GET	POST	Country
20.83.147.x	168,604	48	168,556	United States
79.110.62.x	16,248	519	15,729	Switzerland
31.14.75.x	10,280	144	10,136	Czech Republic
81.161.229.x	9,648	126	9,522	United States
20.162.210.x	6,132	126	6,006	United States
89.187.163.x	5,040	72	4,968	Turkey
185.207.35.x	3,572	84	3,488	United States
89.187.163.x	2,689	72	2,617	Germany
20.166.249.x	1,927	84	1,843	Czech Republic

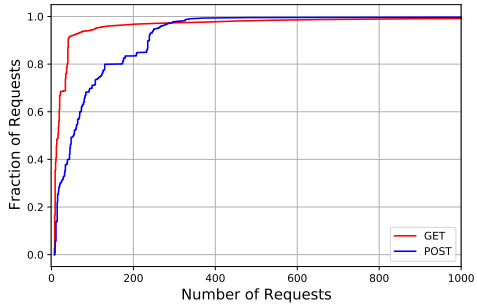

Fig. 8: *CDF of requests per number of requests FA and 404 honeypots in the wild*