Hyp3rArmor

Reducing Web Application Exposure to Automated Attacks

William Koch
Department of Computer Science
Boston University
Boston, MA 02215
wfkoch@bu.edu

Azer Bestavros
Department of Computer Science
Boston University
Boston, MA 02215
best@bu.edu

***Abstract*—Web applications (webapps) are subjected constantly to automated, opportunistic attacks from autonomous robots (bots) engaged in reconnaissance to discover victims that may be vulnerable to specific exploits. This is a typical behavior found in botnet recruitment, worm propagation, large-scale fingerprinting and vulnerability scanners. Most anti-bot techniques are deployed at the application layer, thus leaving the network stack of the webapp's server exposed. In this paper we present a mechanism called Hyp3rArmor, that addresses this vulnerability by minimizing the webapp's attack surface exposed to automated opportunistic attackers, for JavaScript-enabled web browser clients. Our solution uses port knocking to eliminate the webapp's visible network footprint. Clients of the webapp are directed to a visible static web server to obtain JavaScript that authenticates the client to the webapp server (using port knocking) before making any requests to the webapp. Our implementation of Hyp3rArmor, which is compatible with all webapp architectures, has been deployed and used to defend single and multi-page websites on the Internet for 114 days. During this time period the static web server observed 964 attempted attacks that were deflected from the webapp, which was only accessed by authenticated clients. Our evaluation shows that in most cases client-side overheads were negligible and that server-side overheads were minimal. Hyp3rArmor is ideal for critical systems and legacy applications that must be accessible on the Internet. Additionally Hyp3rArmor is composable with other security tools, adding an additional layer to a defense in depth approach.**

I. Introduction

The World Wide Web (Web) is the foundation of the Information Age, providing billions of people with a tool to discover, learn, and share information. The Web is comprised of a vast number of web servers that handle requests for information from hosted websites. Web servers have the largest visible footprint on the Internet [1], yet a recent report has found that almost all websites have serious, exploitable security vulnerabilities [2].

In order for attackers to capitalize on these vulnerabilities, they must first identify potential victim web servers. This is done through the use of automated, opportunistic agents (a.k.a., robots or bots) that scour the Internet for potential victims based on some search criteria. Typically, bots search for a vulnerable web application (webapp) through *network scans* or through web *application fingerprinting*.

Network scans send a probe to each destination to identify if it qualifies as a target. The probe may be as simple as checking if a web server is running on the destination computer, by checking if particular ports (e.g., port 80, 443, 8080, etc.) are open, or by determining the type and version of the operating system or the software running on the host. A popular strategy used by bots is IP address network scanning which provides a fast, efficient way to identify potential victims. For example, a new ransomware strain, known as SamSam, includes worm capabilities allowing it to spread through a network, encrypting and holding users files hostage for money [3]. SamSam scans the Internet for vulnerable Jboss web servers to compromise. Currently, the IPv4 address space can be scanned in under 5 minutes [4].

Bots may also acquire targets through web application fingerprinting. Probes are sent to a web server to elicit information about exploitable application-layer software used at the server. In some cases the bot doesn't even need to search for exploitable applications; instead it could leverage search engines to locate targets using application frameworks with exploitable vulnerabilities (which may not yet be known publicly, e.g., a zero-day vulnerability). Such search engine query strings are known as *dorks*[5]. For example the query `allinurl: "wordspew-rss.php"` will locate a SQL injection vulnerability in a Wordpress plugin [6]. In 2013, Google dorks were used to compromise 35,000 websites running vulnerable message board software [7].

Typically, web applications use a cocktail of defenses to protect themselves from automated attacks. Network firewalls, intrusion detection systems (IDS), and intrusion prevention systems (IPS) protect the server at the network layer, while web application firewalls (WAF) provide protection at the application layer. Additionally, there are several defensive strategies that are specific to automated attacks, including honeypot links and CAPTCHAs [8]. While these defenses reduce the risks associated with autonomous attacks, they are by no means satisfactory, as they typically exhibit a high-rate of false positives, and are altogether ineffective against zero-day attacks.

We posit that as long as a server maintains a visible footprint on the Internet (accepting packets from unauthenticated clients), it remains susceptible to automated scans. Faced with such risks, the only truly effective defense is to minimize the server's attack surface *at the network layer* by only accepting packets from vetted clients (i.e., legitimate users and not bots).

Towards that end, a lesser-known and seldom-deployed defense is *port knocking*, which conceals a network service on the Internet, making it invisible to unauthenticated clients [9]. Generally speaking, port knocking allows a client to let a server open an otherwise-closed port (i.e., ”white-list” the client) through special one-way communication, or secret knocks. Any client communication that does not start with a valid knock elicits no response from the server, rendering the server invisible to such client. More specifically, port knocking is implemented by having the server's firewall monitor incoming network traffic. If a client ”knocks” by sending a packet encoded with an authentication token (AT) that is associated with a particular service, the client's IP address is white-listed for that service.[1] If the client sent the correct sequences of ”knocks” to the server (i.e., by sending a sequence of TCP-SYN packets with the correct destination port numbers) the client's IP address would be white-listed. Any other traffic is summarily dropped by the firewall.

Port knocking, which is typically used to hide private services (e.g., SSH), has not been widely adopted in other protocols due to (1) the lack of a standardized key distribution mechanism, and (2) the need for the client to generate, and send packets encoded with the ATs. Thus, without modifying the client to install client-side port knocking software, a webapp's attack surface exposed to automated opportunistic attacks will exist. In this paper, we introduce Hyp3rArmor, a new security regime that overcomes this limitation.

Hyp3rArmor uses port knocking to make the webapp's attack surface virtually non-existent, and thus appears hidden to network scans, without requiring alteration to the client.[2] A visible web server, residing on a separate physical machine, is introduced to replace the webapp's visible footprint: Rather than accessing the webapp's server directly, a client is directed to the visibile web server, which provides the client with JavaScript code that induces the client's browser to retrieve and send an AT to the webapp's hidden server for authentication. Once authenticated, the JavaScript sends the original request to the webapp. An AT provider, residing on the hidden web server, publishes ATs to the visibile web server (e.g., using scp or rsync). Alternatively, the AT provider can be hosted as a separate service that distributes ATs directly to clients. A simplified overview of Hyp3rArmor applied to a webapp is illustrated in Figure 1.

In order to deny bots the ability to retrieve ATs, Hyp3rArmor differentiates between legitimate clients and bots by relying on one of two assumptions about the operation of automated bots: (a) Bots search for servers to victimize by scanning large blocks of the IP address space looking for specific services *without knowledge of the domain name* under which the service is offered, or (b) To be effective at scale, bots must operate autonomously *without human intervention or assistance*. Under the first, Hyp3rArmor uses knowledge of the domain name as the secret needed by legitimate clients to generate valid ATs. Under the second, for services offered under a domain name for which the reverse mapping is (or must be) discoverable, Hyp3rArmor forces clients to solve a one-time challenge that requires assistance from a human in order to obtain a valid AT.

[1]Traditionally, an AT was represented by a set of destination port numbers that client requests will target.

[2]The attack surface is not totally eliminated because the operating system and port-knocking monitoring software must process packets, leaving the lower layers of the network stack (transport and below) vulnerable. The risk of exploits at those typically-hardened layers is low.

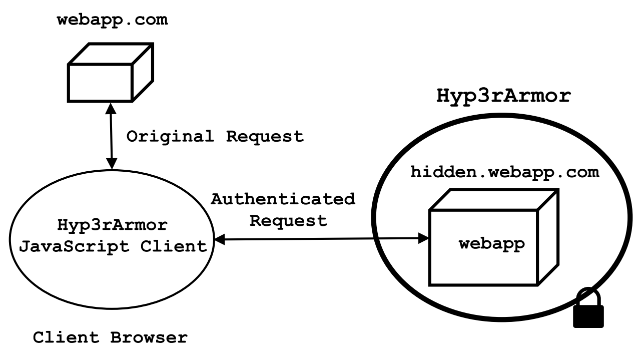


Fig. 1: High level overview of our Hyp3rArmor defense applied to a web application.

We have developed an open-source implementation of Hyp3rArmor, and have deployed it as a defense for single-page and multi-page web applications. Our evaluation has shown that our Hyp3rArmor prototype imposes minimal overheads on both the client and server. With over 114 days of being deployed on the Internet, our Hyp3rArmor prototype has demonstrated its ability to protect a webapp from bots, ensuring that the only packets that reached hidden web server were from authenticated clients. As such, Hyp3rArmor adds a valuable layer of protection for defense-in-depth approaches, allowing it to be composed with other defenses such as IDS/IPS and WAF.

The remainder of this paper is organized as follows. We start in Section II by introducing the threat model and an assessment of associated risks. We follow that with an overview of our proposed mechanism for minimizing a web server's attack surface in Section III and a detailed description of the various components of our design in Section IV. In Section V we present a security analysis of our mechanism. In Section VI we present our Hyp3rArmor prototype and evaluate its performance in Section VII. We provide additional observations and discuss design choices and alternatives in Section VIII. We present related work in Section IX and conclude the paper with a summary of on-going and future work in Section X.

II. Threat Analysis

A. Threat Modeling

Our adversary is software that operates automatically, without human assistance or intervention. This includes autonomous systems such as botnets and worms, as well as automated scanners. In the remainder of this paper, for simplicity we refer to these adversaries as bots. Bots are opportunistic attackers in the sense that they do not know their victims *a priori*. Instead, they perform reconnaissance to search for potential victims. Reconnaissance may include network scanning and application fingerprinting (e.g., Google dorks). Additionally, bots may fetch pre-compiled “hit lists” such as the Alexa top 500 sites on the web.[3] Once targets are acquired, the bot performs the attack autonomously.

We consider attacks on the webapp's entire software stack, including operating system, web server, and application software. Attacks may include, but are not limited to, exploitation to compromise the machine, scraping, and vulnerability scanning. We require that the target acquisition and attack be performed automatically by bots, without human assistance. We consider bots to be end points in a network. As such, they do not engage in network traffic monitoring, and they do not engage in other adversarial behaviors. Thus, for the purposes of this paper, other types of attacks that may be mounted using botnets, such as denial of service (DoS) attacks, are out of scope. Finally, while we expect that bots will exhibit a high level of sophistication and autonomy, we assume that there exists a class of challenges (solvable by humans) that these bots cannot solve.

We distinguish between two categories of bots: those knowledgeable and those not knowledgeable of their target's identity, specifically its domain name. We refer to the former as a Domain Name Aware Bot (DN-Bot), and refer to the latter as a IP Aware Bot (IP-Bot). This distinction is based on the reconnaissance method adopted by the attacker. For example, if victims are found using Google Dorks, then knowledge of the domain name comes for free, if it exists. However, if the bot performs reconnaissance by IP scanning, additional work is required to determine if the target's domain name can be resolved.

If the bot is knowledgeable of its target's domain name, then it readily has knowledge of the server's IP address through a forward DNS lookup. However, the inverse is not necessarily true. IP addresses are not required to have a reverse DNS lookup, and even if a record exists, it does not necessarily reflect the domain name of the web application at the IP address (e.g., virtual name-based hosting allows multiple web applications with different domains to be assigned to a single IP address). Although there may be other ways to derive the domain name from an IP address, this can be made hard for the attacker.

B. Risk Assessment

The effectiveness of our Hyp3rArmor mechanism depends on how well the ATs are secured from bots. Otherwise, if the bot obtains the AT, it only needs to authenticate itself to the hidden web server and perform its attack as normal. Recall, in Section II-A we identified bots as Domain Name Aware Bot (DN-Bot) and IP Aware Bot (IP-Bot). Hyp3rArmor must know (or otherwise assume) the bot type in order to defend the webapp from the threat posed by the bot. DN-Bots encompass the knowledge of IP-Bots, therefore using Hyp3rArmor for DN-Bots also protects the webapp from IP-Bots.

The basis of our Hyp3rArmor defense from IP-Bots is the use of virtual-name based hosting, which requires the client to specify the webapp's domain name in the `Host` HTTP header when requesting web resources. Therefore if the bot does not know its target's domain, it will be unable to obtain the AT. Hyp3rArmor defense from DN-Bots is based on the client solving a challenge (i.e., CAPTCHA). Hyp3rArmor for DN-Bots requires human intervention, while Hyp3rArmor for IP-Bots is completely transparent to the user. Therefore, for an improved user experience, if the webapp is not vulnerable to DN-Bots, the webapp should use a Hyp3rArmor for IP-Bots.

To determine which bot type a webapp is vulnerable to, we developed the flow chart illustrated in Figure 2 to determine if the webapp's domain name is at risk of being targeted, and if a domain name can be found given an IP address. We elaborate on this risk assessment process (which determines the appropriate defense) below.

If the webapp is accessible by an IP address as opposed to a domain name, virtual name-based hosting is not an option and Hyp3rArmor for DN-Bots should be used.

Next, the webapp must be analyzed to determine if it is at risk of a bot searching for it by its domain. This is done by cross-referencing the webapp's software with existing Google dorks, found in the Google Hacking Database [10]. If dorks exist they should be removed.[4]

The webapp's domain may exist in a list, which DN-Bots may target. The nature and popularity of such a list greatly affects the level of risk. On the one hand, inclusion in a list such as the Centralized Zone Data Service (CZDS)[5] list of participating Top Level Domains carries a low risk as the domain is not necessarily hosting a web server. On the other hand, inclusion in a list such as Alexa is considered high risk as evidenced by the use of these lists by automated software: After the HeartBleed bug was disclosed, the top 1 million sites on Alexa were regularly scanned to determine if they were vulnerable [11]. Removal from such lists may be impossible. For this reason we mark these decisions in the flow chart with dotted lines.

If a bot cannot find the webapp by domain, bots may exist that search for targets by IP scans and then attempt to find the webapp's domain given its IP address. Many techniques exist to find this reverse mapping, e.g., Web sites storing IP-to-domain mappings could be queried by a bot. Other sources include checking if a DNS PTR record exists, and if a Whois record contains values that may leak the domain name, e.g., an email address.

Lastly, if the web application is served over HTTPS, the server must not respond with its certificate to a client that does not first provide the domain name. Currently OpenSLL will readily reveal its identity to anyone who requests its certificate. We have made a patch to provide this functionality, details are discussed in Section VI.

If the webapp domain is not a target, and an IP-to-domain reverse mapping does not exist, Hyp3rArmor for IP-Bots is sufficient, otherwise Hyp3rArmor for DN-Bots should be used.

III. Hyp3rArmor Mechanism Overview

To minimize the webapp's attack surface exposed to opportunistic bots, our Hyp3rArmor mechanism introduces three key components: a port-knocking daemon, a visible web server, and an AT provider. These components, along with their

[3]http://www.alexa.com/topsites

[4]Google provides tools to remove entries from its index [7].

[5]https://czds.icann.org/

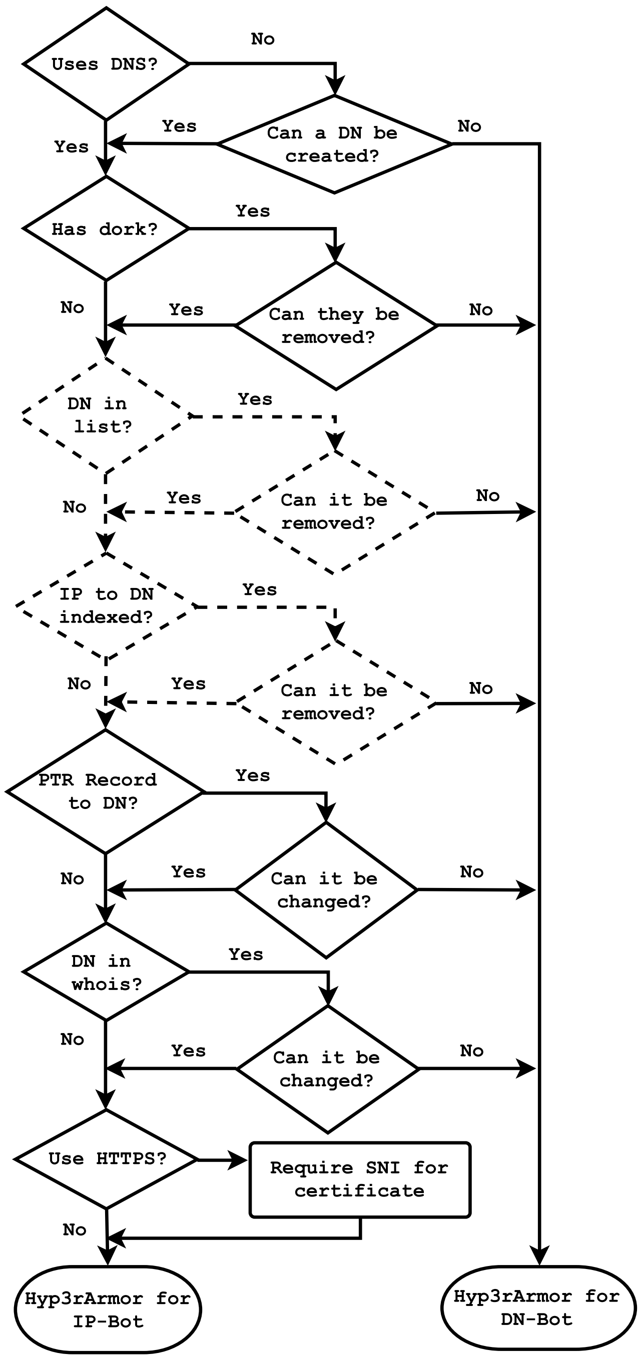


Fig. 2: Threat analysis to determine which bot the webapp is vulnerable to, so Hyp3rArmor can be properly configured.

interactions with the webapp server and client browser, are shown in Figure 3. The remainder of this section provides an overview of these components and how these components might be instantiated in order to deploy Hyp3rArmor on existing web architectures.

A. *Hyp3rArmor's Port-Knocking Daemon*

Our end goal is to protect the webapp from attack, under the realistic and justified assumption that any webapp's software stack is exploitable (e.g., using zero-day attacks). As software stacks continue to grow and increase in complexity, so does the webapp's attack surface. To virtually eliminate the attack surface exposed to unauthorized clients, a port-knocking daemon is installed on the web server. The daemon continuously monitors attempted connections to the server, looking for clients that exhibit a behavior consistent with a valid "knock" – e.g., demonstrating possession of the correct AT. If such a client is identified, then the client's IP address is temporarily white listed, and hence allowed to access the web service. Traffic from non-white-listed clients is summarily dropped.

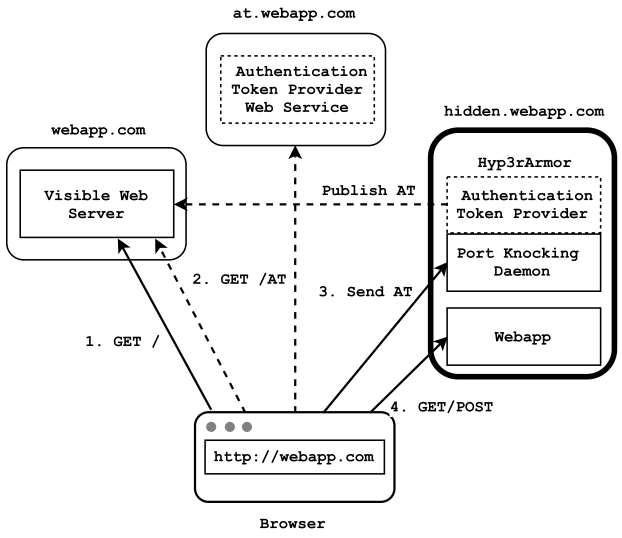


Fig. 3: An overview of Hyp3rArmor. Only a single AT provider is used.

Since clients will no longer be able to access the web application without first obtaining the means to generate a valid AT, a visible web server is used as the new access point for clients. Therefore, the original webapp's server (to be hidden) is assigned to a new domain name, e.g., by appending a domain label `hidden` to the original domain name, which is then assigned to the visibile web server.

B. *Hyp3rArmor's Visible Web Server*

Using Hyp3rArmor, the visibile web server only serves static content (HTML, CSS and JavaScript), thus drastically reducing the webapp's reachable attack surface. The visible web server is never modified based on any client requests, nor does it keep state.[6]

When a client requests the root web page, the visibile web server returns JavaScript code that provides the functionality for the client to retrieve and send ATs to the port-knocking daemon. By using JavaScript, the client does not require any specially-installed software to send the ATs; it only requires a JavaScript-enabled browser.

JavaScripts network privileges are limited when executed within a browser; low level packet creation is not permitted.

[6]If using Hyp3rArmor for IP-Bots this server must adhere to all changes in Section II-B.

However JavaScript can make HTTP requests to arbitrary destination ports. We leverage this functionality as a way to communicate to the hidden web server that the client possesses a valid AT by representing the AT as a multiset of k destination port numbers. We describe this next.

For each destination port in the multiset representing the AT, an Asynchronous JavaScript and XML (AJAX) GET request is sent with the URL format `scheme://host:port`, where `scheme` is HTTP or HTTPS, `host` is the domain of the hidden web server and `port` is the destination port.[7] While these requests will fail (because the client is not currently white-listed), the port-knocking daemon on the hidden web server will be able to record the set of port numbers targeted by the TCP-SYN packet (the first packet of the 3-way TCP handshake), and thus establish that the client possesses the AT.

To reduce the delay associated our JavaScript's implementation of port knocking above, the AJAX requests are sent all at once, and the port-knocking daemon is designed to reconstruct out-of-order packets. After the AT has been sent, the JavaScript code probes the hidden web server to check when the clients IP address has been white listed. Once white listed, the JavaScript makes the application request to the webapp.

C. Hyp3rArmor's Authentication Token Provider

The AT provider generates and provides the ATs that are necessary for client authentication using the port knocking daemon. New ATs are generated periodically (every predefined time step) requiring clients to re-authenticate to maintain access to the webapp.[8]

In this paper, we present two AT constructions each providing a different scope. universal ATs are shared among all clients, whereas IP-bound AT are bound to client IP addresses and thus not shareable. Each of these AT constructions requires a different design. We describe these next.

A universal AT provider runs on the hidden web server. At a pre-configured time step, the universal AT provider generates a random AT to be shared among all clients, publishing it securely to the AT provider (requiring the provider to hold special privileges to be able to do so, e.g SCP).

An IP-bound AT provider runs as a separate web service and generates ATs individually for each client, using a *secret* and the clients IP address, thus preventing the AT from being shared with clients of different IP addresses. The port-knocking daemon shares the secret with the IP-bound AT provider, allowing the port-knocking daemon to correctly verify the knocks.[6]

D. Website Architecture Modifications

Hyp3rArmor can be applied to any existing webapp, whether the webapp is designed using a single-page or a multi-page wsebsite architecture.

[7]From the perspective of the network, each knock is a TCP-SYN packet targeting the AT destination port.

[8]Similar to DNS, the expiration time allows the AT to be revoked if needed. Shorter lived ATs provide the defender with more control and also can limit the time an attack may attempt to brute force the AT, but will also introduce additional processing overheads due to increased number of requests.

A *single-page website architecture* provides an experience similar to a desktop application. This architecture cleanly separates the roles of the user interface (UI) from the application business logic. When a website is designed as a single-page application, client-side static content that defines the user interface (i.e., HTML, CSS and JavaScript) is loaded. Through AJAX requests, JavaScript code provides a bridge to a back-end server hosting the application logic to dynamically respond with content. The back-end server exposes an application programming interface (API) defining how the JavaScript can communicate with it. The back-end server is usually connected to at least one database where it stores and retrieves data. For performance reasons, it is common for the static content and the servers accessible through the API to be hosted on physically-separate machines (e.g., static content is commonly hosted on a content delivery network (CDN) servers).

Applying Hyp3rArmor to single-page website architectures is straightforward since the architecture already provides separation of static and dynamic content. The back-end web server (producing the dynamic content) poses a greater attack surface and stores valuable data, and hence must be protected. The only change is the inclusion of an additional JavaScript file that implements the logic to retrieve and send the ATs to the hidden web server. The back-end application server is installed with the port knocking daemon and AT provider.

Multi-page applications are comprised of two or more pages; requested content is defined by the resource at the URL, and is retrieved through a page load. These pages are commonly generated dynamically by the server.

Applying Hyp3rArmor to a multi-page applications involves the JavaScript code redirecting requests to the webapp loaded in an Iframe after authentication occurs. First, the multi-page applications domain must be altered, appending a new domain label (e.g., `hidden.webapp.com`). Next, a visibile web server is created to host the JavaScript file and is assigned the applications original domain. The JavaScript code must keep the browser state and multi-page application hosted on the hidden web server synchronized. Changes to the browser input, such as page loads and navigation, must be reflected in the Iframe. Changes to the Iframe must be reflected in the browser (i.e., navigation and page titles).

The visibile web server must be prevented from responding with a Resource Not Found error (i.e., 404). This can be achieved by one of two methods: resource stub pages or configuring the visibile web server to rewrite URLs and always load the document root, in effect making the occurrence of a 404 impossible. Resource stub pages are HTML files named after resources existing on the webapp which include the JavaScript authentication code. Stub files can be automatically generated if given the webapp's site map. However this approach does not scale. An alternative, and preferred approach, is to configure the visibile web server to always return the document root (i.e. index.html) containing the JavaScript code.

To synchronize the browser with the Iframe, when the client requests a resource (e.g., `webapp.com/login`), the JavaScript extracts the relative URL (`/login`), and sets the Iframe source to a URL with the host set to the webapp's new domain and the path of the relative URL (`hidden.webapp.com/login`).

To synchronize the Iframe with the browser, we subscribe to the Iframe's `onload` event, which is fired every time the Iframes location changes. When this event is fired, document properties such as the title, and most importantly navigation and history are updated. Although the child document loaded in the Iframe resides at a subdomain of visibile web server, the parent document can still access its Document Object Model (DOM) when the child sets the `document.domain` property to the parent domain name.

IV. Hyp3rArmor Mechanism Details

In this section, we provide additional details for each Hyp3rArmor component.

A. Hyp3rArmor's Visible Web Server

The visible web server hosts the Hyp3rArmor static web files to provide the functionality to authenticate to the hidden web server. The details of this process are described in Algorithm 1.

Recall from Section III, Hyp3rArmor provides defenses for IP Aware Bots and Domain Name Aware Bots. The defense to use is specified as the BotDefense input. Additionally the ATSrc specifies the URL where the AT can be loaded. If the AT scope is universal, the universal AT provider publishes the AT directly to the visibile web server thus the ATSrc is a local URL. Otherwise, the ATSrc is the URL of the IP-bound AT provider web service.

If Hyp3rArmor is for DN-Bots the client must solve a challenge to obtain the AT. The challenge is obtained by calling the `getChallenge` function. The challenge is then displayed to the user with a countdown timer indicating the amount of time available for the user to solve the challenge. If the timer expires and an answer is not returned, the algorithm returns false. Otherwise the answer is used as the key to decrypt the cipher text to obtain the AT. If the Hyp3rArmor is for IP-Bots, the AT is simply queried and returned without human intervention. Once the AT is obtained, it is sent to the hidden web server via the `send` function.

Until clients are authenticated, they cannot communicate with the hidden web server. Therefore a client must probe hidden web server to determine when it is granted access (making the server visible to it). Ideally, this probe is as small as possible such as an AJAX HTTP HEAD request, which will not include the response body, for a service running on the hidden web server. Alternatively, the JavaScript can probe for an image such as a favicon to check whether or not it can load. If the hidden web server does not become visible after a pre-defined number of tries, the authentication fails. For example, this could occur if the user incorrectly solves the challenge.

The client is defined by an authentication policy detailing when and how often the client should authenticate. We have defined two policies: automated, and on-demand. The automated authentication policy states that the client should automatically re-authenticate when the AT expires, whereas the on-demand policy states that the client should only re-authenticate if AT has expired and a web request has been initialized. Automated renewals allow the client to remain in an authenticated state with the hidden web server, providing a proactive approach so requests can be made without any delay.

Algorithm 1 Authenticate

```
Input: The type of bot to defend, BotDefense and the location
    of the AT, ATSrc.
Output: True if authentication is successful, false otherwise.
 1: if BotDefense is DN-Bot then
 2:     ($C_{AT}$, challenge, ATExpire)= getChallenge(ATSrc)
 3:     $sk$ = promptChallenge(challenge, ATExpire)
 4:     if answer then
 5:         AT = decrypt$_{sk}(C_{AT})$
 6:     else
 7:         return false
 8:     end if
 9: else
10:     (AT, ATExpire) = getAT(ATSrc)
11: end if
12: send(AT)
13: wait = 0
14: while not isServerVisible() do
15:     if wait $\geq$ MAX_WAIT then
16:         return false
17:     end if
18:     sleep(TIMEOUT)
19:     wait += 1
20: end while
21: return true
```

B. Hyp3rArmor's AT

ATs are generated by the AT provider. Algorithm 2 describes this process, which we discuss below.

An AT is a multiset of k destination ports, which has a maximum life specified by its time to live (TTL). All AT expire at the same time. Duplicates are allowed in the AT, therefore we represent it as an associative array data structure, where the key is the destination port mapped to the frequency count of each destination port.

ATs with universal scope are generated and published by the hidden web server every AT TTL time units. Each destination port number is randomly sampled from a uniform distribution over the integer interval $(0, 2^{16})$.

To bind ATs to a client's IP address, we introduce a visible server hosting an AT provider web service. The hidden web server must be able to verify ATs generated autonomously by the visible server (which resides on a physical separate machine). To this end we utilize the time-based one-time password (TOTP) algorithm [12]. The TOTP algorithm computes one-time passwords from a shared secret, current time and time step defining the life time of the password before a new one is generated. To keep the hidden web server and IP-bound AT provider synchronized, a master secret S is shared between the two, while the AT TTL time defines the life time of the one-time password, thus at any moment both will be able to compute the same password.

The destination port is then constructed from a truncated Hash-based message authentication code (HMAC) such that its key is a one-time password derived from the TOTP, and the message is a concatenation of the clients IP address and an incremented counter.

Algorithm 2 Generate

```
Input: The size k of the AT, the master secret S, the type
    of bot to defend, BotDefense, the scope of the AT,
    ATScope and the clients IP address, IP.
Output: A tuple including the AT for the defence type.
 1: $AT = \{\}$
 2: for $c = 0$ to $k$ do
 3:     if ATScope is universal-AT then
 4:         dport = getRandomInt($2^{16} - 1$)
 5:     else
 6:         pw = TOTP(S, timeNow(), AT_TTL)
 7:         dport = HMAC(pw, IP $\circ$ c) & 0xFFFF
 8:     end if
 9:     if dport $\in$ AT then
10:         $AT$[dport] += 1
11:     else
12:         $AT$[dport] $= 1$
13:     end if
14: end for
15: ATExpire $= -1 \times$ timeNow() $\bmod$ AT_TTL
16: if BotDefense is DN-Bot then
17:     (challenge, sk) = genChallenge()
18:     $c_{AT} = \text{encrypt}_{sk}(AT)$
19:     return ($c_{AT}$, challenge, ATExpire)
20: else
21:     return (AT, ATExpire)
22: end if
```

If an DN-Bot-based defense is used, the AT provider generates a random (challenge, secret key) pair. Solving the challenge produces the secret key which is then used to encrypt the AT using a symmetric encryption algorithm. Finally the Generate algorithm (shown in Algorithm 2) returns the cipher text, challenge and time when the current AT will expire. Otherwise the AT and its expiration time are returned for IP-Bot defense.

C. Hyp3rArmor's Port-Knocking Daemon

The port-knocking daemon continuously monitors incoming TCP-SYN packets to detect valid ATs as described in Algorithm 3. For IP-bound ATs, each client has a unique AT thus requiring the daemon to first compute the clients AT before it can verify that the received knock is valid. To increase performance, the client's AT is computed when the first valid knock is received and cached until it expires. The port knocking daemon maintains state of each client's AT progress in a cache, keyed using the clients IP address.

To account for network latency during validation, the daemon is able to accept ATs within a window of time, using a similar method used by the TOTP specification. This solves the problem in situations where the client sends an AT at the end of its life, but the verifier attempts to verify the knock for the following time step due to the network delay imposed on sending the knock. The client's state is initialized with ATs that would occur during a defined time window specified by `ATWindow`. The daemon will attempt to match the received destination port to an AT in the window to determine which AT the client is sending. As suggested by [12] the window should be no more than one time step in the past. Once initialized,

Algorithm 3 Verify

```
Require: The verification state, STATE.
Input: The destination port, dport, and source IP address,
    IP.
Output: True if dport was verified, False otherwise.
 1: if IP $\notin$ STATE or STATE[IP][expire] $\geq$ timeNow() then
 2:     STATE[IP] = {}
 3:     STATE[IP][ATWindow] = getATWindow()
 4:     STATE[IP][ATBuild] = {}
 5:     STATE[IP][M] = 0
 6:     STATE[IP][expire]= timeNow() + windowSize
 7: end if
 8: client = STATE[IP]
 9: matchedWindow = []
10: for AT in client[ATWindow] do
11:     if dport $\in$ AT and
12:     (dport $\notin$ client[ATBuild]) or
13:     (client[ATBuild][dport] + 1 $\leq$ AT[dport]) then
14:         matchedWindow.append(AT)
15:     end if
16: end for
17: if len(matchedWindow) $> 0$ then
18:     client[ATWindow] = matchedWindow
19:     if dport $\notin$ client[ATBuild] then
20:         client[ATBuild] = 1
21:     else
22:         client[ATBuild] += 1
23:     end if
24:     client[M] +=1
25:     if client[M] $\geq$ M then
26:         timeLeft $= -1 \times$ timeNow() $\bmod$ AT_TTL
27:         expireTime = timeNow() + timeLeft + AT_TTL
28:         addToWhiteListQueue(IP, expireTime)
29:         delete IP from STATE
30:     end if
31:     return True
32: else
33:     delete IP from STATE
34:     return False
35: end if
```

the client has as long as the AT window size to finish sending the rest of the knocks, specified by `expire`. As the daemon receives correct knocks from the client they are added to the clients `ATBuild` state. The number of correct knocks are tracked by `M`.

Every time a TCP-SYN packet is received, the destination port is checked if it exists in the AT window. If it does, it is checked to ensure that it doesn't exceed the expected count since an AT may contain duplicates. The client's `ATWindow` is then reassigned to all matched ATs in the window, thereby decreasing the size of the window until the true AT is left.

When at least M valid knocks have been received, the client will be granted access to the webapp: The client's IP address will be white-listed for the remaining and the next AT cycle. This allows the client to proactively renew their access ahead of time, thus minimizing any client side delay sending requests to the webapp.Therefore, the maximum time that a client will be white listed for is $2 \times$ AT_TTL. For white-listed clients, the IP address and access expiration is tracked allowing the daemon to periodically revoke all expired clients.

If an invalid knock is received, the server's security access control policy specifies how the client will be handled. For example, a strict policy may be to black list the client's IP address.

V. Security Analysis

The security of Hyp3rArmor is defined by its ability to guard the AT and its ability to resist brute-force attacks. As outlined in Section II-B, the AT is guarded differently depending on the type of bot. Given the rapidly evolving landscape of automated threats, we provide multiple options to protect from current and future threats. Protection from IP-Bots is reliant on it being hard to obtain a domain name given an IP address. While protection from DN-Bots is based on the security of the challenge and the symmetric encryption used.

The scope of the AT (whether it is universal or bound to the client's IP address) determines who is able to possess the AT. Any client possessing a valid universal AT can be authenticated to the hidden web server. IP-bound ATs limit the AT's use to only the IP address for which it was assigned. This renders inconsequential the leakage of an AT to bots. The security of IP-bound ATs is dependent on the secrecy of the master secret for the time-based one-time password (TOTP).

Assuming that the bot is unable to obtain the AT, the only other available option for the bot to gain access to the hidden web server is to brute-force attack the AT. As we explained earlier, the AT is a multiset of destination port numbers to allow out-of-order processing. However by doing so, the set X of all ATs decreases in size in comparison to if the destination port numbers were sent and received in order allowing an encoding as a single string. The size of X for ATs of length k is $|X| = \left(\!\!\binom{2^{16}}{k}\!\!\right) = \binom{2^{16}+k-1}{k} < 2^{16k}$.

A comparsion of the entropy H of each encoding scheme is shown in Figure 4, specifying the number of bits required to encode an AT for each scheme. Although there is a noticeable decrease in entropy encoding ATs using a multiset (as opposed to an ordered set), it is at most an addition of an extra destination port for when entropy is 100 bits or less.

We model the execution of a brute-force attack as a bot randomly sampling an AT from X, without replacement, and sending it, along with a reconnaissance probe, to the server. The bot succeeds if the AT is accepted and the probe signifies that a web service is running, otherwise they fail. We give the bot an advantage by assuming that the bot knows the size k of the AT. We are interested in the expected time, T, it takes to brute-force an AT with a rate μ.

$$E[T] = \frac{(k+1)}{\mu}\frac{1}{|X|}\sum_{i=1}^{|X|} i = \frac{(k+1)}{\mu}\frac{|X|+1}{2} \tag{1}$$

The bot's attack rate potential is limited by their hardware, network access and ability to collude with others (i.e., a botnet). Currently off-the-shelf scanning software such as ZMap can scan near 15 million probes per second (MPPs) with a 10 gigabit Ethernet link [4]. At these speeds, an AT with $k = 1$ has an expected brute-forced time of 4.4ms, with

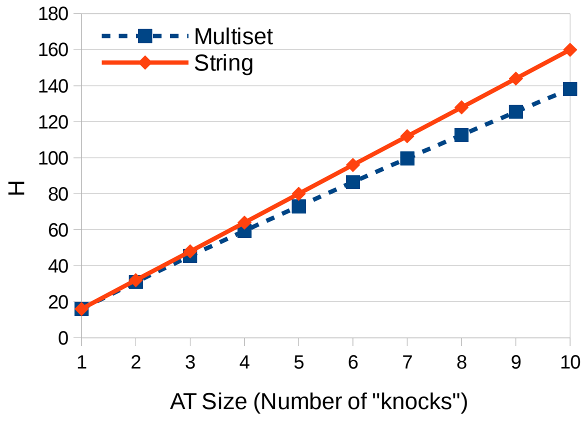


Fig. 4: Comparison of AT encoding scheme entropy.

$k = 3$ this increases to over two months. We note, however, that an attack on a single target at these rates is more of a denial of service (DoS) attack, which would be presumably addressed using other deployed mechanisms (and is outside the scope of this paper).

More importantly, Hyp3rArmor is for opportunistic attackers who are resource-bound in regards to funding and time. Finding exploitable systems is a race against time, especially in the case of newly disclosed vulnerabilities as witnessed in the past [7], [11]. Due to these constraints, the AT size can be substantially less than that used for symmetric encryption, which NIST currently recommends to be 128 bits for AES [13].

VI. Hyp3rArmor Prototype Implementation

We have developed and deployed a fully-functional open-source prototype implementation of the Hyp3rArmor defense. Our prototype consists of three separate packages, corresponding to client-side functionality, server-side functionality, and the AT provider functionality.

The client-side functionality is implemented as JavaScript code for authentication. The JavaScript is configured with the AT source URL and the subdomain of the hidden web server. The JavaScript code does not introduce any client-side dependencies. However, to provide a seamless transition for single-page applications built using this framework, we developed a shim for jQuery.[9] The only modifications required is to include an additional script tag pointing to the JavaScript source and its initial configuration.

The server-side functionality is implemented as a daemon to authenticate clients and generate universal ATs. The daemon is written in Python 2.7 and follows a producer/consumer design pattern for AT verification and client white listing.

The producer process continuously monitors incoming TCP-SYN packets via the Python library Pcapy,[10] which interfaces with the libpcap packet capture library, or optionally through firewall logs. Cryptographic operations used to

[9]https://jquery.com/
[10]https://coresecurity.com/corelabs-research/open-source-tools/pcapy

generate and verify IP-bound ATs are computed with the Cryptography Python library.[11] When a valid AT is received, the clients IP address is added to the producer-to-consumer queue.

The consumer process pulls from the queue and dynamically white lists the IP address which is granted access the webapp port. A rule is appended to iptables using the Python-iptables library.[12] The expiration time is placed in the rule's comment. An additional thread periodically revokes access for expired entries.

Our implementation of the universal AT provider is embedded in the Hyp3rArmor server daemon. An internal clock, with a period equal to the AT TTL, generates an AT as defined in Algorithm 2. Once generated it is exported to the file system allowing the system administrator to configure how the AT is published to the visibile web server. ATs are exported in JSON format, containing an array of destination ports and the time when the AT will expire. Hyp3rArmor for DN-Bots uses the Python Imaging Library (PIL)[13] to randomly generate CAPTCHAs. The answer to the CAPTCHA is used as the key to encrypt the AT using an XOR cipher.

Our Hyp3rArmor implementation also contains a standalone IP-bound AT web service to allow ATs to be bound to the clients IP address. The web services is also written in Python 2.7 using the Twisted Web 13.2.0 framework.[14] The web server is enabled with virtual name-based hosting to defend from IP-Bots. When a client makes a request, the source IP address is used to generate the AT as defined in Algorithm 2. In cases where the web service is sitting behind a proxy (e.g., Nginx), the web service first checks the `X-Real-IP` HTTP header to check if it has been written by the proxy. To allow the JavaScript to request resources from this web services, Cross-Origin Resource Sharing (CORS) is enabled for the configured domain. The AT output is equivalent to the universal AT provider.

VII. Performance and Efficiency of Hyp3rArmor

In this section, we evaluate the performance and effectiveness of our Hyp3rArmor prototype implementation.

A. Evaluation Set-up

We applied Hyp3rArmor to two web sites on the Internet. While the websites provide the same functionality (a simple clock), they are built using the two design architectures discussed earlier: One is designed as a single-page application and the other is designed as a multi-page application, reachable at `spa.example.com` and `mpa.example.com`, respectively.

Our setup consists of three Amazon EC2 micro instances each running a software stack of Ubuntu 14.04, Nginx 1.4.6 and a patched OpenSSL 1.0.2 (required by Section II-B). These servers contain a single core Intel CPU operating at 2.40GHz 1GB memory. The first instance is used for the visibile web server. The second instance is used for the hidden web server with the Hyp3rArmor server-side software installed. The third instance is used for the AT provider web service when IP-bound ATs are used. When using universal ATs, the hidden web server uses `rsync` to publish ATs to the visibile web server. The domains `spa.example.com`, and `mpa.example.com` are mapped to the IP address of the visibile web server, while the hidden web server is assigned the domain `hidden.sample.com`. Additionally the AT provider web services is mapped to `at.example.com`.

Measures were taken to remove fingerprints from the visibile web server such as the version and name in HTTP headers, configuring the firewall to only expose our web servers, and modifications to OpenSSL to require the client to supply the domain name in the `ClientHello` message.

We use a separate EC2 instance as a *jump server* to administer visibile web server and hidden web server from a single static IP address. The firewall rules on the servers are set to only allow SSH connections from the jump server, thus removing the visibility of this service.[15] The jump server hides its SSH port using KnockKnock [14].

The websites protected using our Hyp3rArmor prototype implementation were created using TwistedWeb. The single-page application contains a single RESTful web services running on the hidden web server with an endpoint `GET hidden.example.com/time` to obtain the time. An HTML file hosted on the visibile web server contains JavaScript to make AJAX requests to the web service to update the time on the web page, as well as the Hyp3rArmor JavaScript code. The multi-page application hosted on the hidden web server uses HTML templating to generate a page dynamically with the time for each request. A stub HTML page is hosted on visibile web server to include Hyp3rArmor JavaScript code providing redirection capabilities. Each website is configurable to retrieve ATs from either the AT provider web service or at a local URL where the published universal AT can be retrieved.

B. Authentication Time

Our first set of experimental results report on the time it takes a browser to authenticate to the hidden web server, and consequently the overhead imposed by Hyp3rArmor on client-side perceived performance. More specifically, we measured the time it takes to perform Algorithm 1, i.e., fetch the AT, send it to the hidden web server and verify the client has been authenticated.

We measure the delay for universal ATs, and IP-bound ATs and compare that to the delay in responding to a request with our Hyp3rArmor defense disabled. Our client is Firefox 48.0 to simulate typical browsing performance. The client authenticates every 30 seconds, while the AT TTL is 10 seconds, thus ensuring that each authentication requires a new AT to be retrieved.

Results shown in Figure 5 are from an average of 10 experiments for AT sizes one to eight. An AT of size zero is the delay with Hyp3rArmor disabled. Error bars indicate

[11]https://cryptography.io
[12]http://ldx.github.io/python-iptables/
[13]http://www.pythonware.com/products/pil/
[14]http://twistedmatrix.com/trac/wiki/TwistedWeb

[15]There are other methods to accomplish the same goal without a jump server such as installing a separate port knocking system for SSH on each server, however this provided us with the most flexibility.

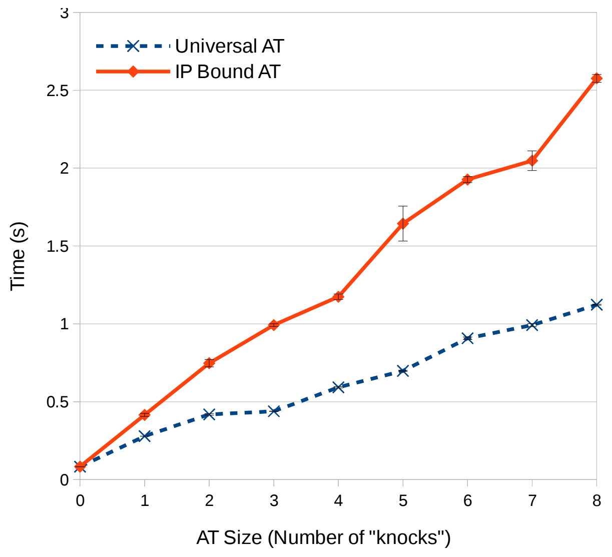


Fig. 5: Comparison of authentication time for universal ATs and IP-bound ATs.

the variance across trials. The results show that authentication time using IP-bound AT is considerably higher (and also grows more rapidly) than that using universal AT. For a single knock, compared to the response time with Hyp3rArmor disabled, we observed more than a two-fold increase in authentication time using universal ATs and about a four-fold increase using IP-bound ATs.

In terms of impact on the user experience, as a general rule of thumb, response times less than one second will be unnoticeable by users [15] and larger response times will be perceived as a degradation in quality of service.

With Hyp3rArmor's port-knocking enabled and using a reactive authentication policy (i.e., one in which the client authenticates every single per request), the user will experience a 3-second delay for IP-bound AT and a 7-second delay for universal AT. The longer delay for universal AT (compared to IP-bound AT) comes with a gain of more than 54 bits of entropy as shown in Figure 4.

We note that with a Hyp3rArmor IP-bound AT implementation, an automated renewal policy with AT_TTL $>$ $(2 \times k)$seconds for an AT size k would result in no perceived delays by the client.

C. *Verification and Rejection Throughput*

Our second set of experiments aim to evaluate the maximum load that our Hyp3rArmor prototype can withstand. We do so by measuring the verification throughput (rate of successful authentication) and rejection throughput (rate of subverted requests) for each AT construction. This is also helpful for the purpose of comparison with a web server's throughput to determine if the adoption of a Hyp3rArmor defense would introduce a bottleneck.

We focus our throughput analysis specifically on the verification code block as specified in Algorithm 3, and execute the experiment on the same server to remove the network as a possible bottleneck. Recall this server has a single core with 1GB memory. For verification throughput we simulate one million unique clients sending valid ATs of size k. For rejection throughput we simulate one million invalid packets sent to the server, each originating from a unique IP address. Results from this experiment are reported in Figure 6.

These results show that processing universal ATs is drastically faster than processing IP-bound ATs. On average, universal ATs are verified 16 times faster than IP-bound ATs; the rejection rate for universal ATs is 137 times faster than that for IP-bound ATs. This comes as no surprise since the AT must be generated individually for every new client when using IP-bound ATs to be able to verify if the received destination port is valid. This consists of computing a TOTP and an HMAC to bind the IP address to the port. A TOTP is constructed with an HMAC therefore for each received port this requires two HMAC calculations or four hashes. Using universal ATs, the processing overhead is minimal since ATs are pre-computed for the duration of the ATs TTL.

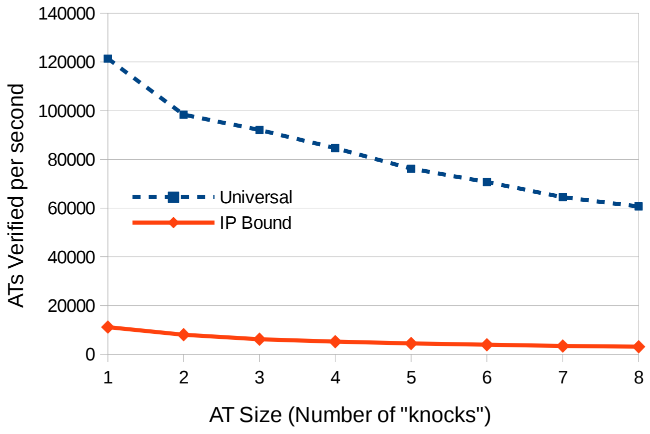
Although processing IP-bound ATs incurs large overheads, the achievable throughput compares favorably to that of an Apache server's throughput, when the AT size is less than four (6K actions/sec)[16]. Verifying universal ATs for the max tested AT size of 8 has a throughput close to twice that of Nginx Stable (60,658 verifications per second compared to 30,487 requests per second), which is listed as the fastest benchmarked web server.

D. *Hyp3rArmor Effectiveness*

Our last set of experimental results aim to evaluate the effectiveness of Hyp3rArmor in minimizing the exposed attack surface of the webapp's server. We perform this evaluation by first conducting our own attack surface analysis of the deployed websites, and then by analyzing measurements obtained from the in-the-wild deployment of our Hyp3rArmor prototype.

An attack surface is defined by [17] as the set of entry/exit points, channels and untrusted data items of a system. To assess the attack surface of the deployed websites from the context of a bot, we performed vertical port scans (i.e., ports 1 to 65535) to identify the visible entry and exit points. The network scanner, Nmap[16], found the hidden web server to have all ports closed, while the visibile web server has ports 80 and 443 open (the minimal for an HTTP/S server). The application layer of the visibile web server does not contain any entry/exit points as this server only hosts static content. However the visibile web server has two ports open, allowing anyone on the Internet to send packets to be processed all the way up the network stack (e.g., it is susceptible to Heartbleed [11] and ShellShock [18] attacks). Furthermore the port-knocking daemon is a communication channel which can be used to invoke methods on the hidden web server and is therefore part of the attack surface. The only untrusted data we receive is from libpcap, or we read from the firewall logs, in which IP addresses and destination port numbers are extracted from packets and sanitized. Given the visibile web server and the hidden web server have identical software stacks, Hyp3rArmor reduces the overall attack surface and isolates compromises to the visibile web server.

[16]https://nmap.org/



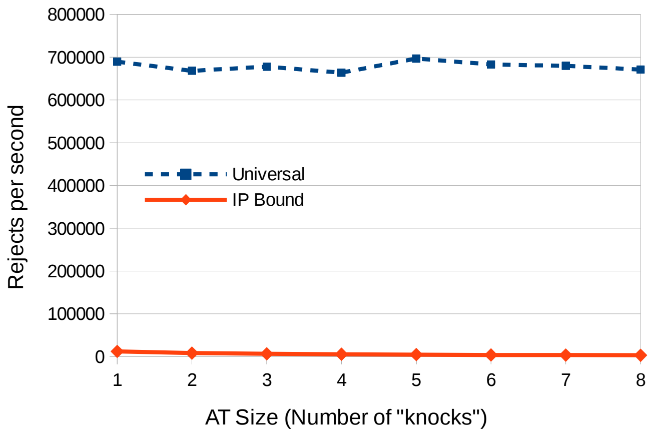


Fig. 6: AT Verification (top) and Rejection (bottom) Throughput of Hyp3rArmor for both AT constructions.

For a DN-Bot defense, compromising the visibile web server does not put the hidden web server at any greater risk. The bot still needs to solve the challenge to decrypt the AT to obtain authorization to hidden web server. For a IP-Bot defense, compromising the visibile web server would gain them access to the AT and thus the hidden web server. This risk could be decreased with additional hardening, such as the use of an intrusion prevention system (IPS) to detect system compromises and respond by locking down the system and destroying the AT to prevent escalation to the hidden web server. The visibile web server would be an ideal candidate for defenses such as a N-variant systems [19] due to the minimal load compared to the hidden web server.

A threat analysis, as outlined in Section II-B, was conducted to determine the risk of an IP-Bot obtaining the AT, results are displayed in Table I. The DNS PTR record and Whois identifies our servers as Amazon which is a benefit of using third-party hosting. We use Zgrab[17] to obtain the TLS banner. Due to our OpenSSL modifications, TLS banner grab fails since the server name indication (SNI) is not specified. Our results show there is not a risk of a bot identifying the domain from the aforementioned identifier methods.

[17]https://github.com/zmap/zgrab

To evaluate the effectiveness of a deployment of Hyp3rArmor to protect against threats "in the wild", we collected access logs from the visibile web server and hidden web server web servers for 114 days. During this time, *not a single unauthorized web request* was made to the hidden web server. In contrast, the logs of the visibile web server show significant number of unsolicited in-bound traffic. A detailed breakdown of the servers activity is summarized in Table II.

We categorized each request as either benign or malicious, and we also recorded whether the request was made using the server IP address or using its domain name. Our classification considered search engine crawlers and scanners grabbing the document root (i.e., "GET /") as benign traffic. Search engine crawlers are identified by IP address and user agent. Furthermore, any traffic/activity due to our own experiments have been excluded from these results. In addition to categorizing requests as benign or malicious, we also characterized malicious requests based on their behavior, e.g., requests attempting to use our server as a proxy, requests probing our server to identify if a particular resources exists, and requests attempting code injection and ShellShock attacks.

Our results show that all malicious requests are made to the server's IP address and not to the domain name, suggesting that the bot source is enumerating IP address to search for victims. The majority of the attacks we observed originated from a bot known as Muieblackcat, targeting vulnerabilities in PHP applications [20]. We obtained 6 requests for the document root by domain name for which we cannot establish the intended purpose: the user agents indicate a standard browser as the origin, and the requests were all for the document root. These very well could be legitimate users.

VIII. Discussion

In this section, we we discuss considerations that are important for deploying Hyp3rArmor, as well as alternative design and implementation choices and open questions.

A. *Hyp3rArmor Defense versus Proxy Server Defense*

A reverse proxy server is a proxy server that responds to requests on behalf of an origin web server. Requests made to the origin web server first flow through the proxy, which is then forwarded to the origin server. The use of a proxy server provides many security benefits, including hiding the location of the origin server and deployment of web application firewalls (WAF). As such, proxy servers may be seen as an alternative to a Hyp3rArmor defense.

Many third-party providers offer reverse proxies as a service (e.g., Akamai, Cloudflare, and Amazon). Cloudflare's WAF even has a setting to present clients with a challenge that must be solved before the request is forwarded to the origin server, similar to our Hyp3rArmor for DN-Bots [21].

When used to proxy for an HTTPS website, the website administrator must share the private key used for encrypting HTTPS traffic with the third-party service provider so that the provider is able to act on behalf of the website[22]. Given that private key, the third-party is able to inspect the website's inbound traffic. For websites that involve the exchange of sensitive data this is a serious issue.

Identifier Method	Tool	Result	Risk Level
IP address, domain name pair indexed	Google search	None	None
PTR Record	dig	ec2-server IP.compute-1.amazonaws.com	None
Domain in Whois	whois	Amazon Technologies Inc.	None
Get certificate by IP	Zgrab TLS banner grab	Conversation error with remote host	None
		Overall Risk	None

TABLE I: Risk analysis of an IP-Bot identifying the server's domain name given only the server's IP address

		Domain	IP
Benign	Crawler	169	0
	Scan	0	482
	Unknown	6	0
Malicious	Proxy	0	93
	Probe	0	781
	Code Injection	0	78
	ShellShock	0	12
	Sub Total	175	1446

TABLE II: Summary of requests made to visible web server

Our Hyp3rArmor mechanism does not suffer from this vulnerability since the visibile web server acts more as a redirect than a proxy, and as such the webapp's traffic does not flow through it. Using Hyp3rArmor, the website administrator can create two separate certificates: one for the visibile web server and the other for the hidden web server. The administrator never needs to share their private key for the hidden web server hosting the webapp.

Another disadvantage of using proxy servers is the necessity for proxies to act as forwarding/relay agents for *all* the website's inbound and outbound traffic. This is typically done through the use of line-speed hardware acceleration. While such a solution allows proxies to sustain high throughput to/from the an origin server, it tends to introduce delays since *all* traffic must first pass through it before going to the origin server. Our Hyp3rArmor mechanism does not suffer from this "relaying" overhead: once a client is authenticated, all traffic flows directly to the webapp's server without the need for an intermediate forwarding/relay agent. The processing requirements of a visibile web server compared to its proxy counterpart is minuscule.

Interestingly, for these same reasons, using redirection (as opposed to proxying) has been the norm used by botnets for a long time [23]. Botnets use a layer of bots for redirection to hide the identity of the command and control server (a.k.a., mothership). Due to the limited resource capabilities of most bots (which tend to be compromised home computers), redirection is preferred to relaying.

B. Suggested Hyp3rArmor Configurations

To maintain secure Hyp3rArmor configurations, threat monitoring will play an important role to observe evolving opportunistic attack behaviors. Honeypots, or decoy servers, can provide us with this intelligence [24], [25].

Evidence from our evaluation suggests that current opportunistic bot attacks are not sophisticated. Given the current threat landscape, universal ATs are sufficient for preventing automated attacks from opportunistic attacks which provide minimal overhead. Determining the AT size should take into consideration the attackers search space. Given current network link capacities and scanning rates, an AT of size three would be acceptable for Internet facing webapps. If the search space is decreased (e.g., due to the use of an internal network) the attacker could potentially spend more time attempting to brute-force the AT. However, sending a large number of requests of this nature would certainly trigger IDS/IPS sensors. Configuring a firewall to rate limit request and ban malicious uses could further decrease the size of the AT.

C. Deployment Considerations

Hyp3rArmor stops all bots from accessing the hidden web server, including search engine bots (e.g., GoogleBot). In fact, this is necessary because if the webapp hosted on hidden web server was indexed, a user trying to access the link would not be able to do so since they would not have necessarily been white-listed, and therefore the website would appear down.

Without the ability to crawl a website, search engine optimization (SEO) will be affected. Only the static content hosted on the visibile web server will be crawled and indexed. This may be a deciding factor on whether or not to deploy this defense. For sensitive, security conscious websites, SEO is a secondary consideration and the use advantage from using Hyp3rArmor far outweighs the benefit from SEO.

Some organizations use egress filter at network edges to control out going traffic. This would cause problems for Hyp3rArmor given its current AT generation algorithm. This can be addressed by having system administrators modify egress filters to allow sending TCP-SYN packets to any destination ports for pre-approved domains and IP address using Hyp3rArmor. It may also be possible to dynamically change egress policies. For example, the edge device could perform a reverse DNS lookup on the IP address and look for a domain label indicating the server at the IP address using Hyp3rArmor. Additionally, other AT encodings may able to coexist with strict egress policies. Introducing a temporal dimension and only targeting HTTP/S ports 80 and 443 could be an option to encode ATs. This will have to be investigated further to determine how practical this would be.

Challenges other than CAPTCHAs may be favorable to organizations to defeat DN-Bots. Disconnected tokens, e.g., RSA

SecurID[18], are commonly used for two-factor authentication and could also be used to generate ATs.

As we mentioned in our threat model, denial of service (DoS) is out of scope as it involves a specific target and this paper is focused on opportunistic attackers. Nevertheless, DoS is always a concern when deploying any new solution or services. An adversary, familiar with the operational characteristics of Hyp3rArmor, could mount a DoS attack by forging the source IP addresses of TCP-SYN packets, and thus denying legitimate users originating from the forged IP address block from authenticating. The effectiveness such a DoS attack will greatly depend on the servers access policy (e.g., ignoring these packets versus black listing the source IP). Existing techniques to detect spoofed packets [26] could mitigate this risk.

D. Miscellaneous Optimizations

The performance of our Hyp3rArmor prototype implementation could be further improved using a set of miscellaneous optimizations. We discuss a few of these below.

Browser cookies can improve the users' experience and reduce unnecessary verification overhead on the hidden web server. Once the client is authenticated, a permanent cookie could be created and set to expire when the AT expires. This allows the authentication state to be saved even when the user closes their browser. The next time the user accesses the website, the cookie is first checked before re-authenticating to the hidden web server. These cookies would be created and read by JavaScript and thus the HTTP-only flag would not be set. The HTTP-only flag is used to prevent cross-site scripting (XSS) attacks from occurring by disallowing JavaScript from accessing these cookie. However XSS is not a threat to the visibile web server as it only serves static content therefore not allowing the XSS attack to persist.

To detect mobile users switching between networks, the IP address at the time of authentication would also have to be saved. If using the IP-bound AT provider web service it could provide additional functionality to tell the client of its public IP address. Alternatively, there is an HTML5 extension, WebRTC, able to obtain the clients local IP address [27].

Finally, we note that our current implementation is in Python. Thus verification and rejection throughput could be increased if Hyp3rArmor is implemented in a lower level language such as C.

IX. Related Work

From the moment criminals began using software to automate malicious tasks, researchers have responded with defenses. In the context of opportunistic attacks, potential victims are found through network reconnaissance and application fingerprinting.

Increasing the difficultly of information gathering has been a popular research area in Moving Target Defense (MTD). A survey of the different dynamic networks was presented in [28] which describes techniques for frequently change network configurations to confuse the attacker. For example, IP hopping is a defense to make reconnaissance difficult for an attacker by frequently changing the IP address of the end hosts [29], [30]. This type of defense is effective against hit-list based worms, which first create a list of targets before moving onto the attack phase. An analysis of the effectiveness of these MTD strategies has been presented in [31].

Other approaches, such as port knocking, resist network reconnaissance by attempting to make a server/service entirely invisible to network scanners. Port knocking has been around for some time with many different proposed schemes. In 2001, Christian Borss, in a Linux User Group Mailing List [32] proposed a method equivalent to port knocking. Shortly after, [33] proposed a scheme to conceal a service from non-authenticated users. In their work, authentication tokens are encoded as a sequence of destination port numbers which they refer to as Spread-Spectrum TCP (SSTCP). SSTCP was constructed with stateful protocols in mind, such as SSH, in which authentication only needs to occur once at the beginning of a session. This approach bears resemblance to the port-knocking implementation in Hyp3rArmor.

Vasserman et al. [34] defines a formal security model for port knocking. Although there is no port knocking standard, there is a draft for TCP Stealth, [35] which encodes an AT in a TCP initial sequence number. Other strategies such as single packet authentication, have a similar goal in mind [36]. Single packet authentication decreases delay time of multi-packet knocking schemes by including access and authentication information in a single packet.

Koch et al. proposed a defense to prevent bots from attacking servers by IP address using port knocking[37]. Their approach can be seen as a generalization of our risk assessment for IP-Bots.

Fingerprinting of system and application software has been used for reconnaissance purposes. For example, the network scanner Nmap has the ability to determine the operating system a host is running. Many techniques to prevent operating system fingerprinting have been outlined by [38]. These methods make changes to the operating system to remove or modify characteristics that would otherwise make them unique.

Defenses against automated attackers searching for potential victims using application fingerprinting have also been proposed. For example, Toffalin et al. proposed defenses against the use of Google dorks by obfuscating URLs and by inserting invisible characters into dork words [5].

Once an attacker acquires its targets, the attack is executed. Intrusion detection systems (IDSs) attempt to detect such attacks and resulting intrusions using monitoring sensors deployed in the environment (e.g., firewall logs) [39]. Attacks are typically detected from anomalies and misuse. However classifying events as malicious can be a difficult task. Efforts to introduce artificial intelligence to build more robust IDSs have also been made [40].

IDSs do not understand protocols at the application layer. Thus, defenses have been proposed that focus on attack prevention at the application layer. Scott et al. proposed an application level firewall (referred to as a security gateway) with user defined validation constraints and transformation rules [41]. The security gateway provides a customized defense for the web application. However, these solutions are primarily

[18]https://www.rsa.com/en-us/products-services/identity-access-management/securid

based on pattern matching against predefined signatures, and therefore are ineffective against zero-day vulnerabilities or other unknown malicious requests.

Defenses using anomaly detection have also been developed to protect against web based attacks [42]. In addition to an anomaly-based web application firewall, TokDoc [43] has the ability to react to malicious tokens found in a request and render them benign.

In some cases the request may seem benign, for example a bot scrapping or mirroring a website. A survey of frequently used defenses to weed out automated bot traffic was conducted by [8]. These techniques are primarily focused on the web application layer, including the use of graphical and audio turing tests. CAPTCHAs, the most well-known of these techniques, uses problems that can be solved by humans but not by current computer programs [44]. Advances in artificial intelligence (AI) have made text-based turing tests ineffective. Mosaic-based Human Interactive Proofs (MosaHIP) exploit current computer vision challenges by requiring the user to distinguish between real and fake images [45].

Polymorphism techniques [46], [47] have a similar goal to CAPTCHAs but attempt to prevent automated web attacks without human intervention. This defense alters the underlying client-side web code to make it difficult for automated attackers to interact with the web page.

X. Conclusion

In this paper we have introduced Hyp3rArmor, a new security system to minimize a web application's attack surface from opportunistic automated attackers. Our contributions include (1) characterization – threat modeling and risk assessment – of automated opportunistic attacks to which a web applications are exposed; (2) Hyp3rArmor, a defense mechanism which minimizes a web application's attack surface exposed to automated opportunistic attacks; (3) an open-source prototype implementation of Hyp3rArmor; and (4) an extensive evaluation of the performance and effectiveness of a Hyp3rArmor prototype implementation which we deployed in the wild.

Our experimental evaluation has demonstrated the benefits from protecting a web application's network stack from unauthorized access when exposed to real-world threats. Attacks attempted (on an hourly basis) on websites protected by Hyp3rArmor is evidence of how active these threats are. Not only does Hyp3rArmor provide protection from threats caused by bots engaged in reconnaissance, but perhaps more importantly, Hyp3rArmor provides an additional layer of security from future unknown attacks, including zero-day attacks.

Our work so far has focused on eliminating opportunistic attacks and reducing application attack surfaces. While our Hyp3rArmor mechanism is totally transparent to clients, it does require architectural changes in the way a web application is deployed. With the increased interest in the use of cloud platforms, we are currently developing Hyp3rArmor "as a service" to allow existing webapps to easily adopt this defense as an add-on service, which can be seamlessly composed with other defenses.

Our realization of Hyp3rArmor used specific AT encoding options that work well in the context of traditional client-sever internet settings. In future work, we intend to investigate other AT encoding options that would be better suited for different deployments, in particular deployments for native mobile applications which will allow for a greater degree of control over AT encoding.

Acknowledgment

This work has been supported by the National Science Foundation (NSF) awards #1430145, #1414119, and #1012798.

References

[1] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *Proceedings of the 22nd USENIX Security Symposium*, Aug. 2013.

[2] Acunetix, "Acunetix web application vulnerability report 2015," Tech. Rep., 2015.

[3] "Meet the cryptoworm, the future of ransomware," https://threatpost.com/meet-the-cryptoworm-the-future-of-ransomware/117330/.

[4] D. Adrian, Z. Durumeric, G. Singh, and J. A. Halderman, "Zippier zmap: internet-wide scanning at 10 gbps," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[5] F. Toffalini, M. Abbí, D. Carra, and D. Balzarotti, "Google dorks: Analysis, creation, and new defenses," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment-Volume 9721*. Springer-Verlag New York, Inc., 2016, pp. 255–275.

[6] "Cve-2008-0682," November 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0682

[7] "Malicious cyber actors use advanced search techniques," November 2016. [Online]. Available: https://info.publicintelligence.net/DHS-FBI-NCTC-GoogleDorking.pdf

[8] G. Ollmann, "Stopping automated attack tools," *Whitepaper–NGS software insight security research*, 2005.

[9] M. Krzywinski, "Port knocking from the inside out," *SysAdmin Magazine*, vol. 12, no. 6, pp. 12–17, 2003.

[10] "Google hacking database (ghdb)," November 2016. [Online]. Available: https://www.exploit-db.com/google-hacking-database/

[11] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[12] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "Totp: Time-based one-time password algorithm," Tech. Rep., 2011.

[13] E. Barker and A. Roginsky, "Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths," *NIST Special Publication*, vol. 800, p. 131A, 2015.

[14] M. Marlinspike, "knockknock," November 2016. [Online]. Available: http://www.thoughtcrime.org/software/knockknock/

[15] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 267–277.

[16] "Linux web server performance benchmark 2016 results," November 2016. [Online]. Available: https://www.rootusers.com/linux-web-server-performance-benchmark-2016-results/

[17] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2011.

[18] "Shellshock: All you need to know about the bash bug vulnerability," November 2016. [Online]. Available: https://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability

[19] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity." in *Usenix Security*, vol. 6, 2006, pp. 105–120.

[20] "Suc027 : Muieblackcat setup.php web scanner/robot," November 2016. [Online]. Available: http://eromang.zataz.com/2011/08/14/suc027-muieblackcat-setup-php-web-scanner-robot/

[21] CloudFlare, "Web application firewall," Tech. Rep., 2016.

[22] F. Cangialosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "Measurement and analysis of private key sharing in the https ecosystem," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 628–640.

[23] X. Hu, M. Knysz, and K. G. Shin, "Rb-seeker: Auto-detection of redirection botnets." in *NDSS*, 2009.

[24] N. Provos, "Honeyd-a virtual honeypot daemon," in *10th DFN-CERT Workshop, Hamburg, Germany*, vol. 2, 2003, p. 4.

[25] S. Antonatos, K. Anagnostakis, and E. Markatos, "Honey@ home: a new approach to large-scale threat monitoring," in *Proceedings of the 2007 ACM workshop on recurring malcode*. ACM, 2007, pp. 38–45.

[26] S. J. Templeton and K. E. Levitt, "Detecting spoofed packets," in *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, vol. 1. IEEE, 2003, pp. 164–175.

[27] "Webrtc 1.0: Real-time communication between browsers," November 2016. [Online]. Available: http://w3c.github.io/webrtc-pc/

[28] H. Okhravi, M. Rabe, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein, "Survey of cyber moving targets," *Massachusetts Inst of Technology Lexington Lincoln Lab, No. MIT/LL-TR-1166*, 2013.

[29] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 127–132.

[30] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Computer Networks*, vol. 51, no. 12, pp. 3471–3490, 2007.

[31] H. Maleki, S. Valizadeh, W. Koch, A. Bestavros, and M. van Dijk, "Markov modeling of moving target defense games," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*. ACM, 2016, pp. 81–92.

[32] C. Borss, "Drop/deny vs. reject," *Listserv post to Braunschweiger Linux User Group (lug-bs@ lk. etc. tu-bs. de). Available at: http://web.archive.org/web/20060618092902/http://www.lk.etc.tu-bs.de/lists/archiv/lug-bs/2001/msg05734.html*, 2001.

[33] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe, "Techniques for lightweight concealment and authentication in ip networks," *Intel Research Berkeley. July*, 2002.

[34] E. Y. Vasserman, N. Hopper, and J. Tyra, "Silentknock: practical, provably undetectable authentication," *International Journal of Information Security*, vol. 8, no. 2, pp. 121–135, 2009.

[35] J. A. H. K. J. Kirsch, C. Grothoff, "Tcp stealth," accessed: 2016-05-17.

[36] M. Rash, "Single packet authorization with fwknop," *login: The USENIX Magazine*, vol. 31, no. 1, pp. 63–69, 2006.

[37] W. Koch and A. Bestavros, "Provide: Hiding from automated network scans with proofs of identity," in *Hot Topics in Web Systems and Technologies (HotWeb), 2016 Forth IEEE Workshop on*. IEEE, 2016.

[38] November 2016. [Online]. Available: https://nmap.org/misc/defeat-nmap-osdetect.html

[39] R. Kemmerer and G. Vigna, "Intrusion detection: a brief history and overview," *Computer*, vol. 35, no. 4, pp. 27–30, 2002.

[40] J. Frank, "Artificial intelligence and intrusion detection: Current and future directions," in *Proceedings of the 17th national computer security conference*, vol. 10. Baltimore, USA, 1994, pp. 1–12.

[41] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 396–407.

[42] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 251–261.

[43] T. Krueger, C. Gehl, K. Rieck, and P. Laskov, "Tokdoc: A self-healing web application firewall," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 1846–1853.

[44] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *Advances in CryptologyEUROCRYPT 2003*. Springer, 2003, pp. 294–311.

[45] A. Basso and S. Sicco, "Preventing massive automated access to web resources," *computers & security*, vol. 28, no. 3, pp. 174–188, 2009.

[46] X. Wang, T. Kohno, and B. Blakley, "Polymorphism as a defense for automated attack of websites," in *International Conference on Applied Cryptography and Network Security*. Springer, 2014, pp. 513–530.

[47] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster, "Webranz: web page randomization for better advertisement delivery and web-bot prevention," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 205–216.