NOMAD: Towards Non-Intrusive Moving-Target Defense against Web Bots

Shardul Vikram, **Chao Yang**, **Guofei Gu**
SUCCESS Lab, Texas A&M University
{*shardul.vikram, yangchao, guofei*}*@cse.tamu.edu*

***Abstract*—Web bots, such as XRumer, Magic Submitter and SENuke, have been widely used by attackers to perform illicit activities, such as massively registering accounts, sending spam, and automating web-based games. Although the technique of CAPTCHA has been widely used to defend against web bots, it requires users to solve some explicit challenges, which is typically interactive and intrusive, resulting in decreased usability.**

In this paper, we design a novel, non-intrusive moving-target defense system, NOMAD, to complement existing solutions. NOMAD prevents web bots from automating web resource access by randomizing HTML elements while not affecting normal users. Specifically, to prevent web bots uniquely identifying HTML elements for later automation, NOMAD randomizes name/id parameter values of HTML elements in each HTTP form page. We evaluate NOMAD against five powerful state-of-the-art web bots on several popular open source web platforms. According to our evaluation, NOMAD can prevent all these web bots with a relatively low overhead.

I. Introduction

Popular websites attract users' attention by providing favorable web services such as information searching, social networking, online blogs and web-based games. Masquerading as normal users, attackers have utilized web bots to launch attacks on those websites and consume precious resources like memory, bandwidth by illicitly registering accounts [23] and automatically posting web spam [11]. According to a recent report released in 2012 [20], 51% of web site traffic is "non-human" and mostly malicious from various automated hacking tools and web bots.

In fact, web bots such as XRumer, Magic Submitter, and SENuke have been developed for creation of backlinks in Blackhat Search Engine Optimization (SEO) techniques, automated content creation on web services, or bulk registration of free services through identifying meanings and functions of special HTML elements. As one of the most powerful and popular forum spam bots, XRumer [19] can be used to automatically register fake forum accounts, build appropriate browsing history, and send spam to boost search engine ranks [40], [41]. It is not only capable of posting spam on multiple popular software platforms (e.g., phpBB and Wordpress), but also can be adapted to new forum types of platforms. XRumer can also circumvent spam prevention mechanisms commonly employed by forum operators (e.g., automatically solving CAPTCHAs). Further more, it can adjust spamming patterns along the dimensions of time and message content to thwart detection [40]. In October 2008, XRumer successfully defeated CAPTCHAs of Hotmail and Gmail to massively create accounts with these free email services [12]. Popular online social networking websites such as Twitter and Facebook have also become web bots' attacking targets [13].

Most existing work utilizes CAPTCHA (Completely Automated Public Tests to tell Computers and Humans Apart) to defend against web bots [21]–[23], [26], [38]. However, CAPTCHA requires users to solve *explicit* challenges, which is typically interactive and intrusive. Also, the robustness of CAPTCHA relies on intrinsic difficulties of artificial intelligence challenges, which could be compromised through achieving artificial intelligence breakthroughs [21], [22]. As those challenges become more complex to defend against evolved web bots, they have become more difficult as well for legitimate users to solve, resulting in a decreased usability. Other approaches, relying on some machine learning techniques to detect bots, are usually time-consuming and error-prone. Such approaches identify web bots mainly through investigating bot behaviors to design detection features [41]. However, designing an effective feature set typically requires considerable time and human efforts. The trained features/models are typically site- or content-specific, not easily applicable to generic websites. False positives are still difficult to avoid, leading to bad user experience. Also, bots could evolve to evade those detection features by better mimicing real human users' behavior.

In this paper, we design a novel, first-of-its-kind, *non-intrusive* moving-target defense system, named **NOMAD**, to *complement* existing solutions. NOMAD explores a new perspective that could be easily combined with existing CAPTCHA and machine learning techniques to form more effective defense. Specifically, NOMAD targets on defending against web bots that can automatically submit bulk requests to remote servers by imitating real users' actions of filling out HTML forms and clicking submission buttons. The basic intuition of NOMAD is based on the fact that web bots need to obtain semantic meanings of corresponding parameters to fabricate normal users' requests through identifying HTML elements. If the web server randomizes HTML form element parameters in each session, web bots would fail in sending bulk automated requests. NOMAD defends against web bots by preventing them from identifying HTML elements for later automation. Unlike CAPTCHA, NOMAD is designed to be totally transparent to end users (i.e., it does *not* require normal users to validate themselves as human via explicit actions). In addition, we believe no security systems are complete without discussing the weakness. To this end, we analyze

possible advanced attacks on NOMAD, propose and implement several extensions to enhance NOMAD, and discuss insights, limitations, and future work.

The main contributions of this paper are as follows:

- We propose NOMAD, a novel, *non-intrusive* moving-target defense system against web bots. While certainly not perfect, it is a new, further step towards a moving-target, defense-in-depth architecture and can greatly complement existing defense solutions.
- We implement a prototype system and evaluate it against five real-world state-of-the-art web bots on four popular open source web applications/platforms. According to our evaluation, NOMAD can prevent all these web bots with a reasonably low overhead.
- We thoroughly discuss possible evolving techniques that can be utilized by more powerful future web bots. We implement several extensions to enhance NOMAD and discuss insights to defeat possible evolving techniques.

II. Problem Statement

Web bots are unwanted automation programs targeting specific websites to automate browsing behaviors, aiming at gaining profits for bot owners. They have become a great hazard to many websites: causing security concerns (phishing and spamming), consuming server resources (bandwidth and storage space), and degrading user experience. Next, we introduce our targeted web bots and their major threats.

A. Threat Model

Current web bots could be mainly divided into two categories: fixed bots and self-learning bots. Fixed bots are typically written in fixed scripting languages, and send simple and fixed HTTP requests. Since the functions of these bots are fixed, they may not work well once their target web applications change their web content. Many cross-site scripting worms on the social network platform, aiming at posting duplicated message spam on victims' profiles, belong to this type.

Self-learning bots are more advanced, intelligent, and complex. They can evolve through analyzing the changes of their target web applications. Also, they can imitate real human users' browsing behavior to automatically send data submission requests. Here, we define data submission requests as those requests that contain users' submitted data through filling out HTML *forms* (e.g., a user authentication request containing values of username and password). XRumer, is a typical self-learning bot, collects inputs with their names, data-types and label-text, which appears next to input HTML forms. Then, XRumer sends specific responses back to the server by filling out these input forms and submitting corresponding HTTP requests. In our work, we target both two types of bots, with a focus on the second one, because if we can defend against advanced bots, we can defeat simple bots.

We next briefly introduce the operating mechanism of self-learning web bots. To fabricate meaningful data processing requests to remote servers, web bots need to have an semantic understanding of the HTML web forms on the web page. Particularly, each input HTML element (e.g., textbox, textarea, checkbox and submit button) in an HTML form, is uniquely implemented in the web page by assigning a unique "name/id" attribute. When a real user submits data through filling out a form, the browser generates HTTP requests containing parameters with key-value pairs, which will be processed by remote servers. The keys in the parameters are "name/id" attributes of input elements, and the values are users' input data. Thus, through parsing and identifying those "name/id" attributes in input HTML elements, web bots can be programmed to automatically fabricate and submit requests with customized content.

B. Web Bots' Threats

Web bots are widely used for the following attacks. (1) *Massive Account Registration/Login.* Attackers could utilize web bots to massively register web application accounts to more effectively launch illicit actions (e.g., sending spam); (2) *Comment Form Spamming.* Attackers can utilize web bots to post customized information (e.g., spam or malicious links) on the comment sections in forums, blogs, or online social networks [3], [16], [37], [41]; (3) *Email Field Spamming.* Attackers can send spam to victims through exploiting mail servers, and utilizing web bots to automatically submit content in the email field of website forms; (4) *Online Vote Cheating.* Poll bots, a special type of web bots, can skew online poll results [8]; (5) *Web-based Game Automation.* Web game bots are developed to automate certain repetitive tasks in the games, which essentially violates the policy of fair competition of online games.

III. System Design

A. System Overview

As described in Section I, the intuition behind NOMAD is that web bots need to pre-identify those unique id/name parameter values of the HTML form elements to automatically interact with remote servers. Because the name/id parameters in an HTML element are used in the server-side logic, these parameters are generally designed as constant. Accordingly, these constant values are utilized by web bots to automatically fabricate massive requests. With this observation, NOMAD is designed to defeat web bots by hiding correct name/id parameter values of those HTML elements. Specifically, NOMAD hides those values by randomizing them in the web source.

NOMAD can be naturally implemented at the server-side by modifying the source code of the web applications. Also, NOMAD could be implemented as middleware between the server and client, to avoid adding the complexity to the server-side logic of the web applications. Implementing NOMAD as a middleware allows it to be independent and universally applicable to different web applications (without directly modifying the source code) and client-side technologies(e.g., different browsers and plugins). Thus, the middleware solution will be transparent to both servers and end users. Furthermore, it is worth noting that NOMAD does not need to randomize all regular web pages. Instead, NOMAD only needs to randomize

a very small percent of HTML *form* pages on which users can submit data. Thus, the workload of NOMAD is relatively lightweight.

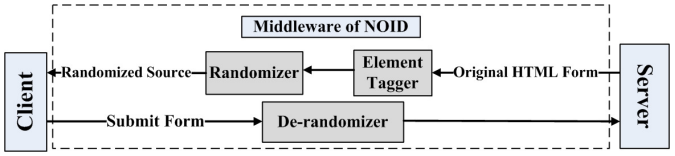


Fig. 1. The system architecture of NOMAD

With this intuition, the middleware version of NOMAD is designed with three major components: *Element Tagger*, *Randomizer*, and *De-randomizer*. As illustrated in Figure 1, once the remote server replies to the client any response of webpage source code that is related to HTML forms such as HTML, Javascript, and CSS, NOMAD will intercept the response and utilize the component of *Element Tagger* to tag those HTML elements in the source for (de-)randomizer to recognize. These elements can be all HTML elements in the webpage or specific types of elements used to achieve web service functions, such as posting content on specific webpages (comments, posts, blogs etc), collecting users' email addresses and so on. Then, *the randomizer* replaces parameter values of those tagged elements in the source with newly randomized values, and relays the randomized HTTP form page to the client. Once the client receives the randomized form page and submits back their form, *the de-randomizer* restores original parameter values of those R-tagged elements and relays to the server. We next describe the design of the middleware version of NOMAD in details.[1]

B. Tagging Elements

The goal of element tagger is to tag those elements that should be protected and randomized by NOMAD. By default, NOMAD randomizes all HTML form elements in the webpage source to defend against web bots. Alternatively, NOMAD can randomize a subset of critical HTML elements depending on the need, and ignore some other elements (e.g., those navigation elements such as buttons and links that are difficult to be exploited by attackers). Some examples of critical HTML elements include the following: a) those allow users to post content in the webpages; b) those used by remote servers to form email headers; c) those used by the server to modify/update other server resources such as a database, a file and in-memory states; d) the values of those elements, which are a part of the parameters in a request.

After knowing which elements need to be randomized, we annotate the parameter values of those elements in the webpage source for the randomizer to recognize. This could be done by either adding a unique and specific prefix with those parameters or specifying totally new parameter values. In our work, we choose the first approach, i.e., we annotate "name/id" parameter values of the elements by adding a unique and specific prefix string, named as "R-tag". Specifically, we use the unique and special string of $\_RaNmE\_$ as R-tag (web developers can easily choose their own unique ones), which is not normally used in the webpage source, aiming at preventing any unwarranted errors. For example, if a submission button with an id value of "submit" (e.g., "$< \ button \ type = button \ \ id = submit \ >$") needs to be randomized, Element Tagger will annotate this element as "$< \ button \ type = button \ \ id = \_RaNmE\_submit \ >$". Since the parameters of those elements can also be used in the presentation code such as javascript and CSS, NOMAD will also annotate elements in those presentation code with R-tags.

[1]The major difference between the server-side version and the middleware version of NOMAD is that the server-side version does not need Element Tagger, which is directly hard-coded by the web developers. However, the principles of (de-)randomizer are still the same.

C. Randomization

The randomizer will randomize those elements with R-tags. In order to guarantee NOMAD to be stateless (i.e., NOMAD does not need to save original values to de-randomize), the randomizer uses a symmetric encryption scheme to randomize the parameters. As seen in Figure 2, the entire process of randomization can be divided into four main steps.

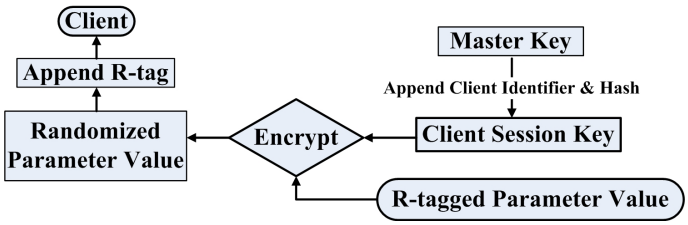


Fig. 2. System flow of the Randomizer

Generate Master Key. The system periodically generates a master key in every "T" time slots[2], which is used to encrypt those annotated parameters. The master key generation should be fast and efficient, since a busy web server may serve millions of requests per day. Thus, we adopt Xorshift random number generator [35], which generates sequences of random numbers basically by applying the "xor" operation on a seed number with a bit shifted version. The seed number could be a random counter or a specific system state value of the server such as system time and number of current processes. Since the change of the master key during each session may cause errors for the de-randomization, once a session is built between the server and client, NOMAD will load current master key and use it to protect web resource for the entire session. In this way, even though a new master key has been generated during a session, the current session will not be broken.

Generate Client Session Key. After generating the master key, for each session, the randomizer transforms the master key to a specific client session key. This client session key is generated by hashing the string of the combination of the master key and some specific client-side identifiers such as IP address.

[2]In our experiment, we choose T=30min, which could be tuned easily depending on the need.

The hashing algorithm can be SHA1 or MD5. The usage of a constant client identifier (e.g., IP address) could guarantee that web pages and javascripts are (de-)randomized by the same client session key.

Generate Randomized Parameter Values. After obtaining the client session key, the randomized parameter values can be calculated by using any encryption cipher (e.g., AES) on the R-tagged parameter value with the client session key. Particularly, although the sever may transfer some javascripts to the client after transferring HTML pages, NOMAD will record the relationships between the javascripts and corresponding webpage requests. Then, it will use the same client and master key to (de-)randomize the javascripts.

Append R-tag. To indicate those elements that should be de-randomized, the randomizer appends R-tags again with randomized parameter values. Finally, NOMAD will relay randomized web source to the client.

At the end of this phase, all (R-tagged) HTML elements are obfuscated thus making web bots unable to recognize correct parameters. As a result, they can not craft massive, correct HTTP requests to successfully send to the servers.

D. De-randomization

Once receiving a submission request[3] to the server according to their received randomized web page source, De-randomizer reverses those "name/id" parameter values with R-tag prefix back to original parameter values in the following four steps as illustrated in Figure 3.

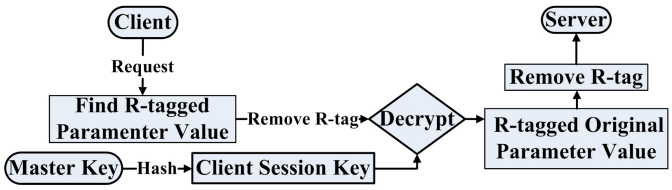


Fig. 3. System flow of De-randomizer

Find R-tagged Parameter Values. When De-randomizer receives users' requests, it will find out all parameter values of the HTML elements that have R-tag prefix. These values will be de-randomized later.

Remove R-tags. Since the randomizer previously appends R-tag prefix to those randomized parameter values before sending them to the client, De-randomizer removes the R-tag prefix in the parameter values.

Calculate Client Session Key. In this step, De-randomizer will retrieve current master key and use client identifier information to calculate client session key. The way of using hash functions to calculate this client session key is similar to the one that used in the Randomizer. Thus, since both the master key and those identifier information will not change during the session, we can obtain the same client session key that is used to randomize parameter values.

[3]The request can be an ajax call, a submitted form, or an HTTP GET request with randomized parameters.

Generate Original Parameter Values. De-randomizer generates original parameter values through decrypting the randomized parameter values with the usage of the client session key. Then, De-randomizer will remove R-tag prefix to generate original parameter values and send them back to the sever.

IV. Evaluation

In this section, we evaluate our techniques of defending against web bots by implementing a prototype system of NOMAD. We next present our implementation, experiment setup, and evaluation results.

A. Implementation

As described in Section III, NOMAD can be implemented as either a server-side approach by directly modifying the web source or a middleware solution between the server and the client. In this paper, we choose the latter approach to implement our prototype NOMAD as a proxy to provide a generic way to defeat web bots without modifying server-side code. As most of the websites behind a proxy can be stacked on the proxy without making any substantial infrastructure changes, NOMAD under such an implementation will be transparent to both the servers and the clients.

We modify the source code of a powerful web proxy called Privoxy [9] to create a NOMAD-enabled proxy through adding a NOMAD module. Privoxy is an open source and non-caching web proxy with advanced filtering capabilities, allowing for modifying web pages and HTTP headers. It can also run in an intercepting mode. We use Pcre [7] module in the Privoxy to search, replace and substitute content, through implementing perl compatible regular expressions in C. The Pcre module allows to use regular expressions to search for R-tag annotated parameters and to replace them with randomized values. We then hook NOMAD in Privoxy's request/response processing module. The HTTP form pages are buffered, modified and then sent to the client.

B. Experiment Setup

We evaluate both security effectiveness and performance overhead of NOMAD on four popular web platforms: phpBB, Simple Machine Forums (SMF), WordPress, and Buddypress [1] (See Table I). PhpBB and SMF are two of the most popular open-source forum/discussion platforms. Wordpress starts as a blogging service but has evolved to be a powerful platform to design websites and content management system [15], [17], [18]. BuddyPress is an open source endeavour to provide easy setup of websites with social networking features, which is built upon WordPress. It has been adopted by a number of websites such as Solo Practice University and hMag [11]. However, through misusing those free services offered by these open-source platforms, attackers have also designed web bots to launch attacks on victim websites built upon these platforms, which caused great displeasure to legitimate users [3]. These four web platforms are chosen with an aim at representing most popular and state-of-the-art systems, which have been plagued by web bots.

TABLE I
EXPERIMENT SETUP OF WEB BOTS AND THEIR TARGETED WEB PLATFORMS

Platform	Platform Type	Web Bots
PHPBB	Forum, Content Management	XRumer, Magic Submitter, SENuke
SMF	Forum, Content Management	XRumer, Magic Submitter, SENuke
WordPress	Blogs, Websites	XRumer, Magic Submitter, SENuke, UWCS, Comment Blaster
BuddyPress	Social Networking Addon	XRumer, Magic Submitter, SENuke, UWCS, Comment Blaster

We evaluate our prototype of NOMAD using five state-of-the-art web bots targeting on four web platforms (See Table I): XRumer, Magic Submitter, Ultimate Wordpress Comment Submitter (UWCS), SENuke, and Comment Blaster. XRumer is one of the most widely used web bots [40]. Magic Submitter allows users to automatically submit bulk messages on blogs and social networks such as Facebook and Twitter [6]. Ultimate Wordpress Comment Submitter and SENuke are two commercial web bots mainly designed for the purpose of SEO. Comment Blaster allows users to automatically send bulk comments or messages on the web platforms.

C. Security Effectiveness

As described in Section IV-B, NOMAD is evaluated on forum and blogging instances of popular open source systems against five state-of-the-art web bots. The settings on these created website instances are liberal, allowing guest/unregistered users to create threads and to post comments. The web bots XRumer, Magic Submitter, UWCS, Comment Blaster, and SENuke can succeed in creating new posts/comments or filling web forms automatically on the website instances.

Thus, we run NOMAD to protect website instances' important webpages such as Login page, Thread Posting page, Comment page, and Registration page. Then, we run those five bots to register, login, and post threads/comments on NOMAD enabled instances of the websites. As seen in Table II, NOMAD can defend against all of the five bots on different platforms. (UWCS and Comment Blaster target blogs, and hence could not be evaluated on the forum platforms.)

We then present an illustration of NOMAD to defend against a Xrumer instance, which attempts to automatically create a new thread on phpBB3 platform. As seen in Figure 4(a), when we do not use basic NOMAD, XRumer can successfully post a phpBB thread. Specifically, XRumer first issues a request (see the lower inset) and verifies that the post is successful (see the upper inset). Then, as shown in Figure 4(b), when NOMAD is turned on and R-tag is appended to the thread creation form elements, XRumer fails to identify those HTML elements. Essentially, this results in an empty request body (see the lower inset). The "unknown" status signifies that XRumer reaches an unknown page, which it cannot identify.

D. Performance Overhead of NOMAD

We evaluate the performance overhead of NOMAD based on the following two metrics: page loading time and webpage size. The former one reflects the time overhead and the latter one reflects the page size overhead. We evaluate NOMAD on three web platforms (phpBB, Wordpress, Buddypress) and five different types of webpages: New Thread page (NT), Post Reply page (Post), Account Login page (Log), Comment page (Comt), and Register page (Reg). The summary of those web platforms and types of web pages can be seen in Table III. ("R-tags" denotes the number of HTML elements that are randomized in the web page.)

TABLE III
SETUP OF WEB PLATFORMS, AND TYPES OF WEB PAGES

Platform	phpBB			Wordpress			Buddypress	
Web Page	NT	Post	Log	Comt	Reg	Log	Comt	Log
R-tags	6	6	5	5	3	4	5	4

Overhead of Page Loading Time. We examine the time overhead of NOMAD by measuring the metric of "Page Loading Time (PLT)", which is the time interval between the timestamp of sending the request and the timestamp of completely loading a webpage by the browser. Specifically, we use Mozilla Firefox's addon LORI [5] to calculate the time used to load and display a web page. Thus, a higher PLT increased by using NOMAD implies a higher overhead. Particularly, this value will be affected by the number of R-tagged elements in the web page, because this number affects the time used to randomize parameter values. Table IV shows the overhead of PLT on different types of webpages in three web platforms. T_{page} denotes the overhead of PLT (in second) to protect the whole page; T_{R-tag} denotes the average overhead of PLT (in second) to protect one R-tag element.

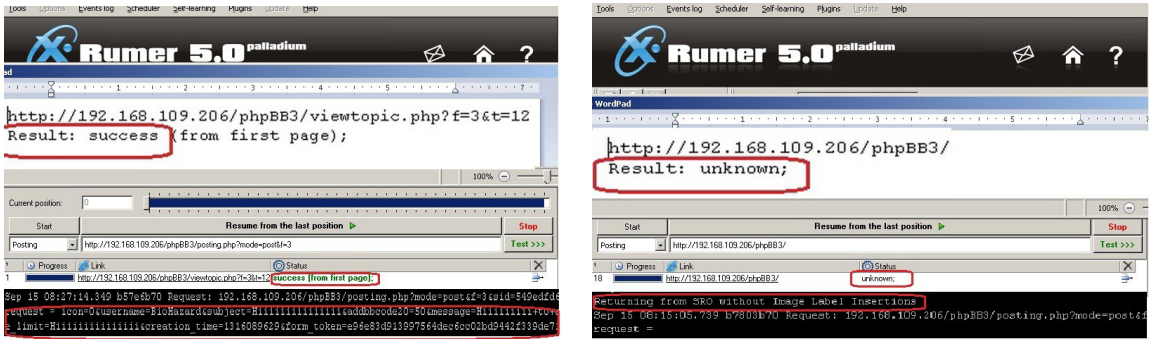
As seen in Table IV, under all of the cases, the overhead of PLT increased by using NOMAD to protect a webpage is less than 0.2 seconds. In terms of the percentage, the average overhead is 14.29% and the highest overhead is still less than 30%. We believe such a short delay is reasonably low in practice, and note that such a delay is only generated on very few HTML form pages instead of most regular webpages browsed by users. In addition, since our evaluation is based on a virtual machine environment with limited processing capability, we expect the performance to be better on a real web server and with code optimization of NOMAD implementation. Also, since each PLT is collected using the browser with empty cache, the overhead can decease even more in a real world communication session with cache storage.

Page Size. We next examine the increased sizes of web pages by using NOMAD. Specifically, we also use Mozilla Firefox's addon LORI [5] to measure the web page size. Table V shows original sizes (in KB) of different types of pages on phpBB3 before using NOMAD (labeled as "Ori") and after using NOMAD (labeled as "NOM"). All of these webpages have six R-tagged elements.

TABLE II

THE EFFECTIVENESS OF USING NOMAD TO DEFEND AGAINST WEB BOTS ON DIFFERENT WEB PLATFORMS. "YES" IMPLIES THAT NOMAD CAN SUCCESSFULLY DEFEND AGAINST THE BOTS ON THAT PLATFORM. "N/A" IMPLIES THAT THE BOT DOES NOT TARGET ON THE PLATFORM.

Web Platform	XRumer	Magic Submitter	SENuke	UWCS	Comment Blaster
phpBB	Yes	Yes	Yes	N/A	N/A
SMF	Yes	Yes	Yes	N/A	N/A
WordPress	Yes	Yes	Yes	Yes	Yes
BuddyPress	Yes	Yes	Yes	Yes	Yes



(a) Success (b) Fail

Fig. 4. Snapshots of requests from XRumer sent to the server to post a new phpBB thread. (a) shows a successful post when NOMAD is OFF, (b) shows a failed attempt when NOMAD is ON.

TABLE IV

OVERHEAD OF PLT (MEASURED IN SECONDS) ON DIFFERENT TYPES OF WEBPAGES IN THREE WEB PLATFORMS

phpBB				Wordpress				Buddypress			
Type	T_{page}	Percentage	T_{R-tag}	Type	T_{page}	Percentage	T_{R-tag}	Type	T_{page}	Percentage	T_{R-tag}
NT	0.196	7.67%	0.033	Comt	0.22	27.5%	0.044	Comt	0.18	24%	0.026
Post	0.192	7.57%	0.032	Reg	0.141	17.65%	0.047	Log	0.129	14.48%	0.045
Log	0.015	1.47%	0.003	Log	0.11	14.01%	0.028				

TABLE V

OVERHEAD OF PLT ON DIFFERENT TYPES OF WEBPAGES/PLATFORMS

NT		Post		Log		Reg	
Ori	NOM	Ori	NOM	Ori	NOM	Ori	NOM
60.94	61.02	62.24	62.36	29.69	29.8	22.57	22.67

As seen in Table V, the overhead of page size by using NOMAD is relatively small. (The average increased size of these web pages is only 0.103 KB.) That is because this overhead mainly comes from the prefix of those R-tagged element values/parameters. According to our implementation, for each element, the randomizer only needs to increase a small number of characters from the original string.

From the previous evaluation, we can see that NOMAD could be used to successfully defend existing web bots with a relatively low overhead of page loading time and page size.

V. ATTACKING AND ENHANCING NOMAD FOR FUTURE THREATS

A. Attacking NOMAD with Hypothetical Future Bots

As described in Section III, the intuition of designing our basic NOMAD is based on the fact that bots use hard-coded name/id parameters of HTML elements to build HTTP requests. We believe our moving-target design of the basic version of NOMAD has already raised the bar for web bots and can be an effective line in the defense-in-depth architecture. In this section, we will think as an attacker to design more advanced (hypothetical) future bots in attempt to evade NOMAD.

Specifically, we envision more advanced bots may evade NOMAD by using the following two approaches: (1) Analyzing label context of elements. Each HTML form element has a label to inform end users about the its semantic meaning. A bot can identify correct input HTML elements through parsing the content description in the label associated with the input element; (2) Analyzing the locations of elements. A bot may achieve the relative location of specific HTML elements with respect to other elements in the page. Hence, identifying and retrieving the correct name/id parameter boils down to moving through the DOM structure along a pre-identified path.

Although to our best knowledge, none of the current bots utilize such intelligent techniques yet, however, we believe it is necessary to think ahead to mitigate such bots when they catch up. Thus, we create a proof-of-concept bot that has advanced capabilities to automatically fill a simple user login form on phpBB3 as shown in Figure 7. This advanced bot first uses hard-coded parameters to submit the form. If it fails to do that, it parses element label context and relative positions of input elements in the page to identify HTML elements. This advanced bot was able to defeat NOMAD and bypass it to successfully submit data requests to the login page. (Due to

the page limitation, we skip our implementation details of this hypothetical advanced bot.)

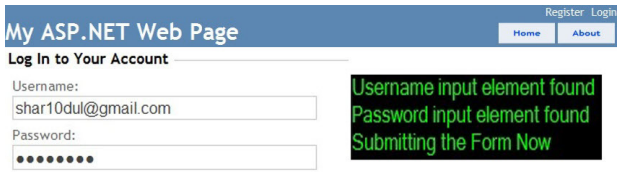


Fig. 5. Snapshot of the success of our proof-of-concept bot on NOMAD

B. Enhancing NOMAD

To defeat our proposed hypothetical advanced web bots, we further strengthen NOMAD by adding two enhanced components: Label Concealer and Element Trapper. The basic intuition is to incorporate some ideas from existing techniques such as CAPTCHA, while still keeping our scheme as implicit as possible (e.g., not asking users to explicitly solve CAPTCHA).

Label Concealer. Since more advanced web bots may learn semantic meanings of HTML elements through analyzing label context, to defeat such web bots' intelligence, the goal of Label concealer is to replace important label text to random images. A label text is a string inside "<label></label>" HTML tags associated with an HTML element, where the string specifies the purpose of the element to users. The connection between the text and the element is built either by setting the attribute of "for" in the label tag "<label for=id>", or by placing the element between the label start and end tags. For example, in a typical login form, the label text of "Username" and "Password" are placed near the corresponding textbox to facilitate users to fill up the login form. A more advanced bot could use the meanings of label text to figure out the name/id attributes of the original HTML element.

To stop bots recognizing label text, Label Concealer will *randomly* substitute label text with images in real time (as illustrated in Figure 6(a)). To avoid even more intelligent bots using the OCG (Optical Character Recognition) technique to identify the context in the images, those images could be generated with noise in a pre-processing step. More discussion could be seen in Section VII.

(a) Image Labels (b) Decoy Elements

Fig. 6. Illustration of label concealer and element trapper

Element Trapper. Since more advanced web bots may identify elements through analyzing elements' relative locations in the webpage, the goal of Element Trapper is to insert decoy elements to stop bots from learning elements' location information. Decoy elements are duplicates of original HTML elements with a random name/id attribute. Also, those decoy elements are randomly inserted nearby the original element (as illustrated in Figure 6(b)).

Although adding decoy elements could prevent web bots' from learning locations of important elements in the webpages, it may also confuse normal users. Thus, to solve such a limitation, we suggest two types of decoy elements: hidden elements and user-notification elements. Hidden elements are implemented to be hidden from normal users' eyes, through setting the "display" or "position" property via css statements in the source. In this way, web bots may obtain wrong HTML elements, by automatically parsing keywords in the HTML source. However, normal users could not see such decoy elements.

To further stop advanced bots from identifying decoy elements through parsing css/html, Element Trapper could also use user-notification images with specific tags, indicating to human eyes that they are decoy only. (The tag could be "Do not fill this", "invalid", "decoy", or a simple notification symbol according to web developers' design. To make sure that decoy elements fit the style (size, color, etc.) of the webpage, the developer could choose a similar design style as used in the webpage to create decoy elements.) In this way, identifying decoy elements is an easy and cognitive task for humans, but a tough AI challenge for web bots. Moreover, enhanced NOMAD could increase the bar even higher for web bots to analyze decoy elements by randomly changing decoy elements' source, locations, and tags.

It is worth noting that the addition of decoy elements needs to balance the usability and security. It is an arms race between web bots and defenders, and there is always a tradeoff between security and usability. Fortunately, one only needs to use them in very few selected HTML form pages (e.g., registration page), which could help minimize the effect to the usability.

We also note that it might be impossible to design a perfect approach to defend against all web bots. Our NOMAD represents a new perspective to incorporate moving-target defense to complement existing solutions. While not perfect, we believe it is a very useful component to form a better defense-in-depth architecture in the battle against web bots. More discussions could be seen in Section VII.

C. Evaluation of Enhanced NOMAD

We next evaluate the security effectiveness of our enhanced NOMAD. Based on the previous design, we implement a prototype of enhanced NOMAD and evaluate it on our proof-of-concept advanced bot mentioned before. As illustrated in Figure 7), once NOMAD is enhanced with two enhanced elements, the advanced bot fails to recognize HTML elements of "Username" and "Password".

Similar to the evaluation on the overhead of NOMAD, we also evaluate the overhead of page loading time and page size on Enhanced NOMAD. With the same configuration on evaluating the basic NOMAD, the average overhead of page loading time by using enhanced NOAMD to protect one webpage and

Fig. 7. Illustration of the failure of the advanced bot under enhanced NOMAD

one R-tag element is 1.37 seconds and 0.29 seconds, respectively. Compared with NOMAD, the relatively high overhead of Enhanced NOAMD is mainly because Element Tagger needs to add decoy images. Accordingly, more time is needed to create/process images at the server side, to download additional image bits, and to load them into the browser at the client. However, this overhead is mainly affected by the number of added decoy elements, which will not increase much as the size of webpages increases. Also, since Enhanced NOMAD will only protect a small number of HTML *form* pages, this overhead will not affect users' web surfing experience (on most of regular webpages).

The average overhead of page size by using enhanced NOMAD is 67.99 KB, which is mainly generated by inserting decoy images. With the increasing network capacity and speed, we believe that a $60 - 70$ KB overhead on very few specific web pages is acceptable.

VI. Related Work

We next introduce current work in the area of bot prevention and mitigation.

A. Analysis of Input

Existing studies have analyzed submitted content and HTTP requests to identify bots [24], [30], [36], [41], [42]. Shin *et. al.* [41] examined the characteristics of forum spam posted at a research blog and developed light-weight features to detect those spam. Mishne *et. al.* [36] proposed detecting comment spam by comparing the language models used in the blog post, the comment, and pages linked by the comments. Bhattarai *et. al.* [24] used content analysis to characterize spam in blogs. Schluessler *et. al.* [39] analyzed input data events such as keystrokes and mouse clicks to detect bots. Brewer *et. al.* [25] used link obfuscation to detect and counter web bots. Gianvecchio *et. al.* proposed HOP-based (Human Observable Proof) approaches to detect web bots [31], [32].

Noncespaces [34] also utilizes the technique of source randomization technique that enables web clients to distinguish between untrusted content to prevent exploitation of XSS vulnerabilities. Although sharing the same spirit of moving-target defense, our NOMAD is a different design and application to defeat web bots.

B. CAPTCHA Systems

CAPTCHA systems have been widely used to defend against web bots. Security is not the only concern of a good CAPTCHA design. All CAPTCHA systems are a form of HIP (Human Interactional Proofs) and require users' involvement. Current CAPTCHA systems could be divided into the following two types: Text-based and Image-based CAPTCHA.

Text-based CAPTCHA. Text-based CAPTCHA systems ask the user to identify letters or numbers [4], [10], [21], [22]. Such systems face one inevitable situation: humans find the CAPTCHA challenge unpleasant as CAPTCHA gets more complicated. While building a high bar to prevent bots identifying content, Text-based CAPTCHA also makes trouble for human users [43], [44].

Image-based CAPTCHA. Since intelligent web bots could defeat text-based CAPTCHA systems, image-based solutions are designed to replace text-based CAPTCHA systems, which are even more complex for human users to solve [2], [14], [27]–[29]. Most current image-based systems (e.g., ESP-PIX [2]) suffer from the limitation of creating and maintaining a large, constantly evolving image database. Although Google provides a new image-based CAPTCHA system [33], which asks the user to adjust the orientation of an image, it is restricted by the requirement of correctly identify subtle mouse (or other hardware) movement.

VII. Limitations and Future Work

Since our prototype of NOMAD is implemented as an intermediate proxy, it could not protect webpages, if the connection between the client and server is encrypted with SSL. However, it is just the limitation of our specific implementation, instead of the limitation of the fundamental solution. The server-side version of NOMAD can easily solve this problem through directly modifying web applications.

Attackers could try to locate R-tagged HTML elements by analyzing the differences among multiple retrieved contents of the same webpage. This approach could be effective when there are few R-tagged elements (out of all elements) in one page, i.e., when the entropy of randomization space is not high. In this way, attackers may somehow guess semantic meanings of some critical elements through checking whether they are changing each time. However, we could increase the bar for guessing critical HTML elements under such situations through R-tagging more randomly selected non-critical (or simply all) HTML elements in the page. Then, this significantly increased size will make it more difficult for attackers to correctly guess out those critical elements. In addition, our enhanced NOMAD with multiple insertions of decoy elements could further increase the entropy of randomization space and decrease the success probability of this kind of guessing or brute-forcing attacks.

More advanced bots may use advanced image recognition capabilities to identify decoy images and recognize semantic meanings of the label images. However, this brings a high cost for attackers, especially when the images are added with random noise (which can be easily implemented). This is essentially as hard as breaking current CAPTCHA-based approaches.

We admit that our enhanced NOMAD, requiring users to identify visible decoy elements, may affect usability. However, these elements are only inserted on the selective pages (e.g.,

registration page). Also, since users at least do not need to pay more efforts on recognizing those decoy elements than solving CAPTCHA problems, it is still an implicit defense approach. We also admit that the insertion of decoy elements and image labels reduces the accessibility of the web site for blind users. This limitation exists for many current web sites. To examine possible effect to users, we plan to conduct a deep user study, and try to create a more accessible design leveraging audio elements in our future work.

Enhanced NOMAD generates a relatively higher overhead than basic NOMAD, because it requires more image insertions. In our future work, we plan to reduce such overhead. For example, to improve the time performance, NOMAD can be designed to create and load images in advance, and then randomly adding different noises to the images. In addition, through analyzing users' HTTP request histories, NOMAD can add decoy elements only when the requests are suspicious.

VIII. Conclusion

In this paper, we present a novel, first-of-its-kind, non-intrusive moving-target defense system against web bots, named NOMAD. It is designed to pose *implicit* challenges to web bots (i.e., unlike CAPTCHA, it does not require legitimate users to explicitly solve challenges.). We also provide enhanced NOMAD by adding two additional components aiming at defeating more powerful future web bots. Our evaluation results show that NOMAD can defeat state-of-the-art web bots on different popular web platforms, with an reasonably low performance overhead for better security protection.

IX. Acknowledgment

This research is partially supported by a NPRP grant (5-648-2-264) from the Qatar National Research Fund (QNRF). All opinions, findings and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of QNRF.

References

[1] Buddypress - social networking in a box. http://buddypress.org/.
[2] Esp-pix. http://server251.theory.cs.cmu.edu/cgi-bin/esp-pix/esp-pix.
[3] Flooded by spam/splog registrations. http://wordpress.org/support/topic/.
[4] Gimpy project. http://www.captcha.net/captchas/gimpy/.
[5] Lori: Life-of-request info. https://addons.mozilla.org/en-US/firefox/addon/lori-life-of-request-info/.
[6] Magic submitter. http://www.magicsubmitter.com/.
[7] Pcre. http://www.pcre.org/.
[8] Poll bots. http://www.scriptlance.com/projects/1290407615.shtml.
[9] Privoxy. http://www.privoxy.org/.
[10] recaptcha. http://www.google.com/reCAPTCHA.
[11] Spam injection. http://www.web-form-buddy.com/html-wfb/spam-injection.html.
[12] Spam swine break next-gen captchas. http://www.theregister.co.uk/2008/10/03/captcha_break/.
[13] The spambots on twitter are completely out of control. http://wilwheaton.typepad.com/wwdnbackup/2009/08/the-spambots-on-twitter-are-completely-out-of-control.html.
[14] Sq-pix. http://server251.theory.cs.cmu.edu/cgi-bin/sq-pix.
[15] Top powered by wordpress sites. http://www.tripwiremagazine.com/2009/11/20-remarkable-examples-of-websites-powered-by-wordpress.html.
[16] Web form spam alive and kicking. http://blog.trendmicro.com/web-form-spam-alive-and-kicking/.
[17] Wordpress - wordpress is web software you can use to create a beautiful website or blog. http://wordpress.org/.
[18] Wordpress powers 14.7 percent of world's sites. http://www.h-online.com/open/news/item/.
[19] Xrumer. http://www.botmasterlabs.net/.
[20] Web bots vs humans: We're losing. http://www.sitepoint.com/web-bots-vs-humans/, 2012.
[21] L. Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using hard ai problems for security. In *In Proceedings of Eurocrypt*, 2003.
[22] L. Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Commun. ACM*.
[23] H. Baird and K. Popat. Human interactive proofs and document image analysis. In *Proceedings of the 5th International Workshop on Document Analysis Systems V*, DAS '02.
[24] A. Bhattarai, V. Rus, and D. Dasgupta. Characterizing comment spam in the blogosphere through content analysis. In *IEEE Symposium on Computational Intelligence in Cyber Security, (CICS'09)*.
[25] D. Brewer, K. Li, L. Ramaswamy, and C. Pu. A link obfuscation service to detect webbots. *International Conference on Services Computing*, 2010.
[26] K. Chellapilla, K. Larson, P. Simard, and M. Czerwinski. Designing human friendly human interaction proofs (hips). In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '05.
[27] M. Chew and J. Tygar. Image recognition captchas. In *In Proceedings of the 7th International Information Security Conference (ISC'04)*.
[28] R. Datta, J. Li, and J. Wang. Imagination: a robust image-based captcha generation system. In *Proceedings of the 13th annual ACM international conference on Multimedia (MULTIMEDIA '05)*.
[29] J. Elson, J. Doucerur, J. Howell, and J. Saul. Asirra: A captcha that exploits interest-aligned manual image categorization. In *Proceedings of the 14th ACM CCS*, 2007.
[30] S. Gianvecchio, Zhenyu Wu Mengjun Xie, and Haining Wang. Detecting Blog Bots through Behavioral Biometrics. 2013.
[31] S. Gianvecchio, M. Xie, Z. Wu, and H. Wang. Measurement and Classification of Humans and Bots in Internet Chat. In *USENIX Security Symposium*, 2008.
[32] Steven Gianvecchio, Z. Wu, M. Xie, and H. Wang. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. In *USENIX Security Symposium*, 2008.
[33] R. Gossweiler, M. Kamvar, and S. Baluja. What's up captcha?: a captcha based on image orientation. In *Proceedings of the 18th international conference on World wide web (WWW'09)*.
[34] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information tracking and thwart crosssite scripting attacks. In *16th Annual Network and Distributed System Security Symposium*, 2009.
[35] P. Matthews and C. Zou. Scene tagging: image-based captcha using image composition and object relationships. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10.
[36] G. Mishne. Blocking blog spam with language model disagreement. In *In Proceedings of the First International Workshop on Adversarial Information Retrieval on the Web (AIRWeb'05)*.
[37] Y. Niu, Y. Wang, H. Chen, M. Ma, and F. Hsu. A quantitative study of forum spamming using contextbased analysis. In *In Proc. Network and Distributed System Security Symposium (NDSS'07)*.
[38] Y. Rui and Z. Liu. Excuse but are you human? In *11th ACM international conference on Multimedia*, 2003.
[39] T. Schluessler, S. Goglin, and E. Johnson. Is a bot at the controls? detecting input data attacks. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games (NetGames '07)*.
[40] Y. Shin, M. Gupta, and S. Myers. The nuts and bolts of a forum spam automator. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, LEET'11.
[41] Y. Shin, M. Gupta, and S. Myers. Prevalence and mitigation of forum spamming. In *2011 Proceedings of IEEE INFOCOM*.
[42] S. Webb. Characterizing web spam using content and http session analysis. In *4th Conference on Email and Anti-Spam (CEAS'07)*.
[43] J. Yan, E. Ahmad, and A. Salah. A low-cost attack on a microsoft captcha. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*.
[44] J. Yan, E. Ahmad, and A. Salah. Usability of captchas or usability issues in captcha design. In *Proceedings of the 4th symposium on Usable privacy and security (SOUPS'08)*.