Polymorphism as a Defense for Automated Attack of Websites

Xinran Wang[1], Tadayoshi Kohno[2], and Bob Blakley[3]

[1] Shape Security
xinran@shapesecurity.com
[2] University of Washington
yoshi@cs.washington.edu
[3] Citigroup
bob.blakley@citi.com

Abstract. We propose PolyRef, a method for a polymorphic defense to defeat automated attacks on web applications. Many websites are vulnerable to automated attacks. Basic anti-automation countermeasures such as Turing tests provide minimal efficacy and negatively impact the usability and the accessibility of the protected application. Motivated by the observation that many automated attacks rely on interaction with the publicly visible code transmitted to the browser, PolyRef proposes to make critical elements of the underlying webpage code polymorphic, rendering machine automation impractical to implement. We categorize the threats that rely on automation and the available anti-automation approaches. We present two techniques for using polymorphism as an anti-automation defense.

1 Introduction

A web user interface (UI) is designed for manual use. The intent is that a human interacts with a web UI in a browser, and the web browser acts as a user agent to communicate with a web server. Unfortunately, by design the source code (HTML, JavaScript, and CSS) of every web page is publicly visible, and thus can be exploited by attackers in numerous ways including subjecting the website to automated attacks.

The past decade has seen a staggering diversity and volume of automated attacks on web applications. Man-in-the-Browser (MitB) attacks, such as the notorious Zeus, seize control of the end user's browser and can modify bank transactions without possessing authentication credentials or compromising any of the bank's technology infrastructure. For example, in 2007 the online banking services of KBC Bank were compromised with MitB techniques despite two-factor transactional authentication [1]. Credential stuffing attacks test a list of authentication credentials stolen from one website on a different website to discover where users have re-used their credentials. When originating from a botnet, these attacks can be indistinguishable from legitimate traffic [11]. Business logic denial-of-service (DoS) attacks interact with a website and exercise resource-intensive business logic: these attacks knock over sites without

I. Boureanu, P. Owesarski, and S. Vaudenay (Eds.): ACNS 2014, LNCS 8479, pp. 513–530, 2014.
© Springer International Publishing Switzerland 2014

requiring a significant volume of traffic. Furthermore, these attacks are unstoppable using traditional network DoS defenses [8].

Automation by attackers is not a new problem in web security. For lack of better options, Turing tests are widely used to block automation. As attackers have become more sophisticated about solving Turing tests, either with automation or human solvers, the tests have increased in difficulty to the point that the failure rate of humans approaches the failure rate of bots. Combinations of reputation and rate thresholds are currently promoted by application delivery controller (ADC) and web application firewall (WAF) vendors, [2] but are largely rendered obsolete by the widespread availability of botnets, which reside on the same machines as the legitimate website users.

We propose PolyRef, a novel technique using polymorphism for defense, which may offer a practical path to block certain classes of automation. Our approach is driven by the observation that today's automated interaction with a website often requires interacting with page content transmitted to the browser. By dynamically re-writing the page content, PolyRef impedes two types of attack: HTTP attacks, which rely on known POST or URL parameters to directly construct HTTP requests and DOM attacks, which manipulate DOM elements.

As shown in Fig. 1, PolyRef sits between a firewall and a web server. When a web page sent by the web server arrives, PolyRef finds the target forms and then applies reference and/or field polymorphism techniques. Note that the replacement happens for each page request. When the form is submitted, PolyRef restores the field names of the form back to their original values.

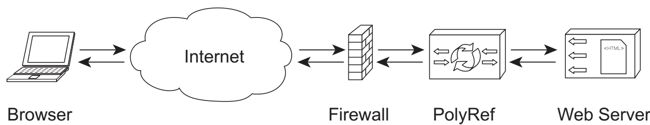

Fig. 1. PolyRef as a transparent proxy

We study several actual automated attack cases, and show how PolyRef uses reference and field polymorphism to impede unwanted automation. We also discuss potential counter attacks for PolyRef and the limitations of PolyRef.

The contributions of this Paper are summarized below.

- The paper systematically analyzes automated attacks against web applications defining a representative threat model, identifying relevant vulnerabilities, applications for automation, and implementation archetypes.
- We propose PolyRef, a new defense concept using polymorphism. We show how PolyRef deflects current generation automated attacks, and analyze impact of potential attacker evolution.
- We implement a prototype of PolyRef as a transparent proxy to protect web servers. We evaluate PolyRef in two experiments: a fake account creation attack and a Zeus MitB attack. The evaluation shows PolyRef is effective to deflect these attacks. We also evaluate PolyRef with a real world large e-commerce website and show latency is very low with caching turned on.

2 Background: Automated Attacks

We observed that many notable attacks on websites seemed to be rooted in automation, yet there are surprisingly few options for a viable defense. We set out to appreciate the extent to which automation is a current problem and look for options to mitigate the threat.

Definition 1. *Automated Attack/Automation. An interaction performed by a program on the user interface of a website where the user interface is intended exclusively for use by a human.*

2.1 Threat Model

To generalize the kinds of threats connected to automated attack of websites, we made a list of archetypical example threats. The list is derived from surveys of security practitioners for large scale websites in e-commerce, financial services, healthcare, national government, and social media, and is also informed by key threats listed in the OWASP Top 10.[1] We discarded threats and vectors not connected to automated attacks and the remainder are shown in the "Surveyed Threats" column of Table 1.

The "Attack Vectors" column of the table lists examples of the corresponding vulnerabilities that might be exploited by automation to realize a successful attack:

Credential Stuffing— The attacker tests a list of authentication credentials stolen from one website on other websites to discover where users have re-used these same credentials. Particularly useful when multiple websites can be correlated with the same credentials such as credit card and e-commerce sites.

Business Logic DoS— Denial-of-service attacks that interact with a website as if they were a human operated browser and exercise resource-intensive business logic. For example, loading a shopping cart on an e-commerce site often causes numerous writes to an underlying database.

Fake Account— Accounts used for the sole purpose of manipulation. Often created or exercised in a sufficiently large volume that they are impractical without automation.

Account Aggregation— Account aggregation services (for example, `mint.com`) collect and use login credentials to access their customer's bank accounts electronically and scrape information from the bank's website.

Carding— Small purchases used to verify the validity of stolen credit card data. Often operated on a large volume of low quality data and therefore reliant on automation. Particularly damaging to certain e-commerce sites as the chargeback fees can be much larger than the transactions.

Man-in-the-Browser (MitB)— A kind of man-in-the middle attack where the attacker controls the user's browser, and may observe or change information that is transmitted between the browser and the website.

[1] Open Web Application Security Project, `www.owasp.org`

Table 1. Relationship of threats to automated attack vectors

Surveyed Threats	Attack Vectors	Automation Application	Vulnerability Category
Account takeover Database scraping	Credential stuffing	Iteration	Inherent
Protection racketeering Hacktivism Masking of other attacks	Business logic DoS		
Comment SPAM Rating/review skewing Database scraping	Fake account		
Customer disintermediation Reduced security posture Database scraping	Account aggregation		
Chargeback fees	Carding		
Credential harvesting Account takeover Transaction manipulation	MitB	Manipulation	
Information leakage Loss of control	XSS CSRF		Inadvertent

XSS/CSRF— Non-persistent cross-site scripting and cross-site request forgery as defined by OWASP.

For the purpose of this threat model, we further narrowed the list of vulnerabilities and vectors to the cases where automation is required. For an attacker, automation is applied for at least two fundamentally different reasons which are noted in the "Automation Application" column of the table.

Iteration— A repeated interaction with a web user interface where a high number of iterations are required to realize value.

Manipulation— A one-time operation performed autonomously by a program over a specific web interface, because it is not practically accessible to a human attacker at the time of attack.

We distinguish between *Inadvertent* and *Inherent* vulnerabilities to highlight an observation[2] that the majority of concerns for large scale websites are for vulnerabilities not contemplated in the OWASP Top 10.

Inadvertent vulnerabilities Some attacks rely on vulnerabilities that are the product of implementation errors or design failures. In theory, this category of vulnerability never need exist and when discovered can be corrected without impacting user experience, business requirements, or application functionality. Many well known web application vulnerabilities such as CSRF, XSS, SQL injection, and the remainder of the OWASP Top 10 belong to this category.

Inherent vulnerabilities Many modern website attacks rely on vulnerabilities that are the byproducts of fundamental design requirements or conditions not

[2] Perhaps our survey suffered from a type of selection bias where our sample had sufficient budgets to remediate the better understood inadvertent vulnerabilities.

under the control of the solution architect. For example, a credential stuffing vulnerability stems from the requirement that sites must allow anonymous connections to attempt authentication, or they fail to meet the most fundamental business need: access from the Internet. To illustrate with a specific example, consider that an attacker could abuse the common security protocol of locking out an account after five consecutive login failures to create a denial-of-service attack. Depending on the design objectives of website, the solution to stop locking out accounts may not be an option. Unlike inadvertent vulnerability attacks, inherent vulnerabilities cannot be mitigated by "fixing" the application as the "fix" is at odds with a design requirement.

2.2 Methods of Automated Attacks

We classify the methods of automated attacks into three categories, each with fundamentally different approaches: *HTTP attack*, *DOM attack*, and *GUI attack*.

HTTP attack— This approach relies on manipulating the target of attack by transmitting GET or POST messages, but without any appreciation of how the target page would be rendered in a browser. A common example is a credential stuffing attack where a simple POST request is transmitted with the username and password key-value pairs. Another variation employed for manipulation instead of iteration is a CSRF attack where the HTTP GET is in the form of URL embedded in an HTML email message.

DOM attack— This attack operates inside a browser and uses JavaScript to feed input into DOM elements and perform submission. In DOM attacks, the target web page and all referenced content including JavaScript is loaded in a browser. The attack software now examines the DOM and feeds input into input elements of a target form. Because DOM attacks drive a real web browser, JavaScript, application state, cookies, nonces, sessions, properly set referrers, and other dependencies that arise in a complex web application are handled seamlessly. MitB attacks take this form (*e.g.*, Fig. 13). Most existing inherent vulnerability attacks, which exploit automation test tools such as Selenium and HtmlUnit, also take this form (*e.g.*, Fig. 11).

GUI attack— A more complicated option is when the attacker takes full control of a real browser to render the image of the target web page and interact with the web page by directing mouse movement/click and keystrokes. It can position the input focus by tab key press, x, y coordinates, or relative vectors, and then stream keystrokes into fields of focus. DOM manipulation is not necessary in this method. Note that GUI attacks need full control of a real browser and cannot be performed inside a web page by JavaScript. Although JavaScript in a web page can simulate a mouse event and cause browsers to fire the default action for the event (e.g., navigate to the link's href, or submit a form), browsers do not perform the default action for simulated keystroke events by JavaScript (e.g., browsers do not assign the value to an input field), and the actual mouse location cannot be changed by JavaScript. Many automated attacks presently implemented with the HTTP or DOM approaches could be adopted to

the GUI approach by implementing them in open source automation test tools like PhantomJS.

Note that CSRF is limited to the HTTP approach, as it has no possibility to control the browser or to access the DOM of the target domain due to the same-origin policy limit. Non-persistent XSS is limited to the HTTP approach or the DOM approach, as it has no control of a browser. Other attacks may choose any of the three methods. The choice of methods depends on the attack requirements, and the methods are used differently due to the required attacker resources and the properties of the targeted web application.

2.3 Scope

In this paper, we focus on HTTP and DOM attacks. GUI attacks are out of scope. As PolyRef forces an adversary to perform GUI attacks with keyboard and mouse activity, the behavioral biometric method [13,14] mentioned in Section 3, which can tell the difference between mouse and keystroke behaviors of a human and those of a bot, can be used to complement the PolyRef method.

2.4 Requirements for a Theoretical Ideal Mitigation Solution

Having defined automation as a fundamental and significant threat it seems clear a protection is needed. We were not able to identify any well accepted industry term of art for this class of solution and chose the term "botwall," a portmanteau of *botnet* and *firewall*.

Definition 2. *Botwall. A website security layer intended to mitigate programmatic or automated use of a website user interface that is intended exclusively for use by a human.*

We propose the following design objectives for an ideal botwall:
Preventive— Able to deflect automation.
Transparent— Does not impact the user experience.
Comprehensive— Broadly useful; not a point solution.
Facile— Easily applied to legacy websites.

3 Related Work

There is a wealth of research on web security, we survey the most relevant works here. Numerous protection techniques have been introduced during the last decade which create some friction for automation. However, all of them either have a low efficacy or a negative impact on usability.

Turing Test— CAPTCHAs [22] are widely used on web to mitigate some automated attacks. However, CAPTCHAs negatively impact the usability and the accessibility of the protected application [23]. In addition, CAPTCHAs do not work for MitB attacks.

Browser Detection— Examination of headers such as "user-agent" or exploration of expected browsers capabilities like running a JavaScript program that calculates the answer to a selected problem.

Reputation— Reputation methods are based on information about the historical activity of endpoints and their connection to activities of ill repute. These methods hinge on being able to establish the unique identity of the endpoint. Common approaches of creating a unique identity include IP address, cookies, and fingerprinting [10, 18]. IP address methods are not reliable because of dynamic IP addressing. Cookie methods are easily bypassed by removing tracking cookies [19]. The fingerprint algorithm collects information such as browser fonts, timezone, and installed plugin to uniquely identify a browser. Fingerprints may be used in combination with other techniques to facilitate a whitelist of known customer devices, or blacklists of problematic devices.

Honeypot— In the honeypot method, faked fields, links, and forms are inserted in the web page. They are invisible to users and only bots can perform the tasks in the honeypot. As honeypot forms are not real forms, honeypot methods cannot be used to prevent inherent vulnerability attacks such as MitB and credential stuffing. This method has been used to detect bots performing reconnaissance attacks [4]. This method can be used to complement PolyRef. PolyRef makes forms polymorphic and the "original" forms can be used as a honeypot.

Rate Threshold— Rate thresholds can be used to detect bots performing iteration attacks. Some application delivery controllers (load balancers) and web application firewalls (WAFs) detect bots by measuring volume and speed in the context of endpoint identity [2]. Often these implementations rely entirely on the IP address for endpoint identity but may also use cookies or browser fingerprinting. This solution is at best a modest barrier today given the widespread availability of botnets to distribute the traffic from rather broad selections of endpoints with a low request rate. This technique generally fails on the efficacy prong of our test as attackers may limit their request rate or generate requests from a botnet to bypass this form of detection. Furthermore, it also fails on the user impact test as well: IP rate-limiting may generate false positives in cases where multiple users are NATed through the same IP address.

Behavioral Biometrics— User keyboard and mouse activity can also be used to detect bots. The method injects a piece of JavaScript code in web pages which collects the user keyboard and mouse activity. The activity is sent back to web servers, and the web servers check if the results fit an expected human behavior distribution. This technique has been used to detect game bots [13], chat bots [14], and twitter and blog bots [6, 7].

Token—The secret validation token method [15, 17, 21] is a approach to defend against CSRF attacks. A secret validation token is attached to each HTTP request. If a request is missing a validation token or if the token does not match the expected value, the server rejects the request. Ollmann [20] proposed token-based methods to protect web applications against some malicious automated scanning tools. One disadvantage of this approach is that a website must maintain a large state table to validate the tokens.

Header Validation— Referer header checking is a common method to prevent CSRF. The header contains the URL of the site making the request, and thus can differentiate a same-site request from a cross-site request. A website can prevent CSRF by checking if a request was issued by the site itself. One problem of referer header checking is that it causes privacy leaking. Barth et al. [5] proposed a defense against CSRF by introducing an origin header with POST requests in the browser. It provides the security benefit of the referer header while responding to privacy concerns. Czeskis et al. [9] proposed a developer-friendly and complete coverage method called Allowed Referrer Lists (ARLs) to prevent CSRF. An ARL is a whitelist of referrer uniform resource locators (URLs) that allows browsers to withhold sending ambient authority credentials for websites wishing to be resilient against CSRF attacks.

Multi-Factor Authentication— It can be used to prevent password dictionary attacks and credential stuffing attacks. The approach requires the presentation of two or more of the three authentication factors: a knowledge factor ("something only the user knows"), a possession factor ("something only the user has"), and an inherent factor ("something only the user is"). However, this approach cannot stop MitB attacks [3].

Out-of-Band Verification— It is an effective method of combating MitB attacks. It overcomes MitB attacks by verifying the transaction details to the user over a channel other than the browser (for example, an automated telephone call or SMS). The downside of Out-of-Band Verification is a negative impact to the user experience from more and slower steps.

4 Proposal: PolyRef

The idea of PolyRef is motivated by the observation that all automated attacks are based on the fixed web page of a web user interface. PolyRef makes the web page of any web user interface polymorphic: the web page is different every time it is served. The variation introduced by PolyRef makes it hard for the attacker to predict how to automatically operate a future page. In this paper, we define Polymorphism as follows:

Definition 3. *Polymorphism. [As applied in this paper] Any technique which makes key elements of a web user interface (for example, HTML/JavaScript references) sufficiently varied for each request so that future constructions of the page are non-deterministic and render automated operation impractical.*

Unlike the use of polymorphism for the construction of malware, it is not our objective to protect intellectual property, obfuscate design, or even impede the manual reverse engineering of a given case, but to make the next case unpredictable or impede automatic program analysis.

We propose two types of polymorphism: reference and field polymorphism. Note that PolyRef is not limited to these two types. It can accommodate new types of polymorphism for any elements of HTML as attack evolves.

4.1 Reference Polymorphism

In reference polymorphism, HTML symbols such as form names, field names, and element identifiers are replaced with random character strings. Fig. 2 shows an example where the form name `Login`, the field name `lastname` and the element identifier `lastname_id` are randomized.

```
<form action="login.jsp" method="post" name=" Login ">
    <input type="text"  id=" lastname_id"  name="lastname">
                    ↓
<form action="login.jsp" method="post" name=" Imp0q6wNm">
    <input type="text"  id=" b24mpqdfKX "  name="aSkFjp5x1Y">
```

Fig. 2. Form Name and Element ID transformation

```
var lastname = document.getElementById("lastname_id").value;
                    ↓
var lastname = document.getElementById("b24mpqdfKX").value;
```

Fig. 3. Example JavaScript with transformed HTML reference

```
var lastname = document.getElementById("lastname_id").value;
                    ↓
var akb8pZx = document.getElementById("b24mpqdfkX").value;
```

Fig. 4. Example JavaScript with a variable name randomized

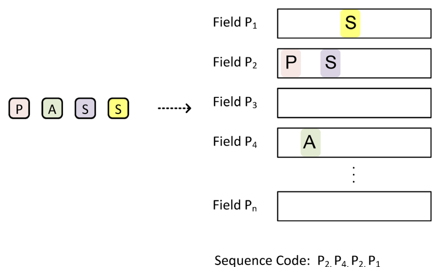

Fig. 5. Sequence code determines the order of field alternation

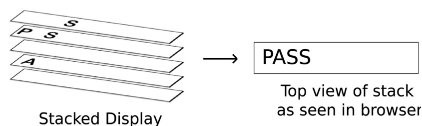

Fig. 6. Stacked display preserves user experience

Form name and element identifier randomization will prevent attackers from directly locating a field. For example, using the JavaScript statement `document.getElementByID(lastname_id)` to locate `lastname` field in Fig. 2 will not work anymore. As form name and element identifier may also be referenced in JavaScript/CSS, the randomization should be consistent. Fig. 3 shows an example of JavaScript changed with HTML symbol randomization.

Similarly, symbols in JavaScript should also be randomized. The JavaScript shown in Fig. 3 makes it clear to an attacker that the field `b24mpqdfKX` should contain a last name due to the JavaScript variable name `lastname`. A simple regular expression could allow an attacker to script the scraping of the field name from the page. Fig. 4 shows that we extend the concept of HTML polymorphism to JavaScript.

4.2 Field Polymorphism

Reference polymorphism is effective for HTTP attacks and existing DOM attacks. However, it is vulnerable to advanced DOM attacks. Advanced DOM

attacks can indirectly find fields to defeat reference polymorphism. For example, instead of looking for the field `lastname`, an adversary could refer to it as the third field of the first form in DOM or even find fields based on the page structure after page rendering.

We propose field polymorphism to impede advanced DOM attacks. In field polymorphism, a field is broken into multiple fields. Keystrokes are distributed between the multiple fields in a pattern that is unique for each page served.

As shown in Fig. 5, focus is alternated between several input fields as each keystroke is typed. The alternation sequence is defined by a constant (sequence code), and a unique code is embedded into the dynamically generated JavaScript added to each page. From the website visitor's perspective, the user experience remains largely unchanged, as all fields are stacked in the display and appear like one field. (See Fig. 6.)

Examination of the POST shows the characters of a single field split between multiple name-value pairs. The constant required to reassemble the sequence was determined in advance, encrypted by a shared key only residing in the PolyRef server. It is embedded in the return POST as a hidden field. In Fig. 7, "KYTr29y7rhKJP6" is the hidden field containing the encrypted constant.

```
POST login.jsp HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 18

P1=S&P2=PS&P3&P4=A&
KYTr29y7rhKJP6=QmFzZTY0IGVuY29kaW5nIHNjaGVtZXMgYXJlIGNvb
W1vbmx5IHVzZWQgd2hlbiB0aGVyZSBpcyBhIG5lZWQgdG8gZW5jb2Rl
```

Fig. 7. Value distributed across multiple fields

```
POST /Locator/retrieveLocation.jsp  HTTP /1.1
Host:  bank.com
Content-Type: application/x-ww-form-urlencoded
Content-Length:127

latitude=37.3986039&longitude=-121.9643745&
atm=on&branch=on&specialServiceIndex=0&
service=null&region=US&distSelected=25
```

Fig. 8. Business logic attack

5 Case Study

We study several real-world automated attack cases, and show how PolyRef defeats them with only reference polymorphism. Note that we only consider contemporary attacks (i.e., attacks that already exist today) in this Section. We will discuss future attacks in Section 8.

5.1 Cross-Site Request Forgery

Let's look at a cross-site request forgery (CSRF) attack on `bank.com`. It has a web page with a form that allows its customers to transfer money.

Assume Alice wants to transfer $50 to Bob. Although the `POST` method is used in the money transfer form, `bank.com` has accidentally allowed `GET` requests as well. Our malicious attacker, Mallet, exploits the vulnerability to automate a form submit with the URL `http://bank.com/transfer.jsp?to_account=Mallet&amount=1000` which will transfer $1000 from an unwitting victim to himself.

There are a couple of ways Mallet can trick Alice into submitting the URL. One way is to include the request as an HTML image element in an email to Alice.

Her browser will make the request automatically as if it were any other image content on a page `<img src=http://bank.com/transfer?to_account=Mallet &amount=1000>`. If Alice's bank keeps her authentication information in a cookie, and if the cookie has not expired, then the attempt by Alice's browser to load the image will submit the transfer request along with her cookie, thus authorizing a transaction without Alice's knowledge.

We can see that in this case, the CSRF attack must have fixed symbols for the forms parameters to work. Polymorphic references therefore stop these attacks.

5.2 Business Logic Denial-of-Service

Denial-of-service attacks have moved up the web stack from early Smurf attacks to syn floods to more modern socket exhaustion attacks. The next generation of DoS attacks focus on computationally intensive requests on back-end servers.

For web applications that must remain up-to-date or for content that cannot be cached for other reasons, distributing content on a worldwide content delivery network (CDN) is not an option. Thus requests must reach back to centralized back-end servers. One example of an attack that reaches back to a back-end server is a branch locator function.

An attacker could craft a POST like the one shown in Fig. 8 that asks a website for branch locations. This computationally intensive request could be made at arbitrary rates from a botnet until the server collapses under the computational load.

This attack is difficult to stop using current defenses. It bypasses CDN caching and does not rely on volume to overwhelm servers. It contains no malicious signature as it is, in fact, a perfectly valid request. However, if the site operator stops automation the attack is stopped.

It is clear to see that the attack shown in Fig. 8 will fail by applying reference polymorphism.

6 Design and Implementation

We constructed a prototype of PolyRef, implemented as a special case of a transparent HTTP proxy located adjacent to the web server. When a web page passes through this special proxy, PolyRef finds the target forms and then replaces selected content with a revised version applying reference polymorphism described in Section 4. Note that a different polymorphic variant is applied for each page request. When the form is later submitted, the same special proxy restores the key/value pairs of the form to the expected content so the protected web application continues to operate without modification. Since many websites terminate SSL at the load balancer, our special proxy would never see HTTPS and hence our implementation does not handle SSL directly.

Our implementation addresses the following scenario: A company — without modifying its own web servers — installs PolyRef as per Fig. 1. The implementation therefore needs to handle the complexity of modern web page design, including the use of CSS and JavaScript. There are two distinct phases for handling

each web page. The first is a *pre-computation* phase, in which PolyRef learns the relevant symbols. This phase is performed automatically the first time a new page is processed and then cached, and hence can employ heavyweight techniques. Following the pre-computation phase is the *application* phase, in which the polymorphic techniques are applied. Our description below focuses on the pre-computation phase.

6.1 Web Page Transformation

If an HTTP response contains a web page, PolyRef will transform it to its polymorphic version with the polymorphic techniques described in Section 4. The transformation is trivial, if the page contains only plain HTML without CSS and JavaScript. However, today almost all web pages contain CSS and JavaScript. We must make symbols consistent among JavaScript, HTML, and CSS; otherwise, the functionality may be broken.

The process of the web page transformation is as follows. In the first step, we find target forms for transformation. The target forms are configured in a profile. We use a HTML parser to parse the web page into a DOM tree. The target forms are identified in the DOM tree. To keep consistency, once we find all relevant symbols, we need to accurately identify all references to them in CSS and JavaScript. A simple regular expression match may have problems. For example, the string `username` in line 12 of Fig. 9 is a reference to the field `username`, while the one in line 16 is not. To make symbols consistent among JavaScript, HTML, and CSS, in the second step, we parsed JavaScript and CSS into abstract syntax trees (ASTs). ASTs can tell us if a symbol is a property of a form. For example, as shown in Fig. 10 (2), `username` is the property of the form; while `username` in Fig. 10 (1) is not.

There are cases where relying on ASTs alone is not enough. For example, variable `pwd` in line 11 of Fig. 9 will be used to get the reference to field `password` and should be made consistent. To handle these cases, we do a static analysis in the analyzer step. We exploit several compiler optimization techniques such as constant folding and propagation in this step. By exploiting constant folding and propagation, the value of variable in line 11 of Fig. 9 will be made consistent. After all references are identified, symbols are randomized consistently across HTML, JavaScript, and CSS, and a new web page is generated in the serializer stage.

In this implementation, two cases are handled with human assistance. First, the fields of forms may be referenced in the `eval` function. Second, forms may be dynamically generated by JavaScript. Future work could potentially address these cases via more sophisticated automation or – changing the model slightly – combining the earlier approach with annotations or support at the web server.

6.2 HTTP Request Restoration

When the transformation is made, the mapping between symbols and randomized values is encrypted and added to the target form as a hidden field. When

```
1: <html>
2: <head>
3: <meta content="text/html; charset=utf-8" http-equiv=
4:   "Content-Type">
5:  <style>
6:   input[name="username"] { background-color:red; }
7:   input[name="password"] { background-color:green; }
8:  </style>
9:  <script>
10:   function validate_input (form) {
11:     var pwd = "pass" + "word"
12:     var name = form.username.value;
13:     var pass = form[pwd].value;
14:     var tmp=" can not be empty";
15:     if (name == "") {
16:        alert("username" + tmp );
17:     }
18:      if (pass == "") {
19:        alert("password" + tmp);
20:      }
21:   }
22:  </script>
23: </head>

24: <body>
25:  <h1>This is a test page.</h1>
26:  <form action="http://test.com/web/login00.asp" id="00"
27:    method="post" name="login00">
28:    Name:<input name="username" size="20" type="text">
29:    Password:<input name="password" size="20" type="text">
30:   <input name="button_00" type="button" value="login"
31:     onclick="validate_input(this.form) >
32:  </form>
33: </body>
34:</html>
```

Fig. 9. HTML consistency example

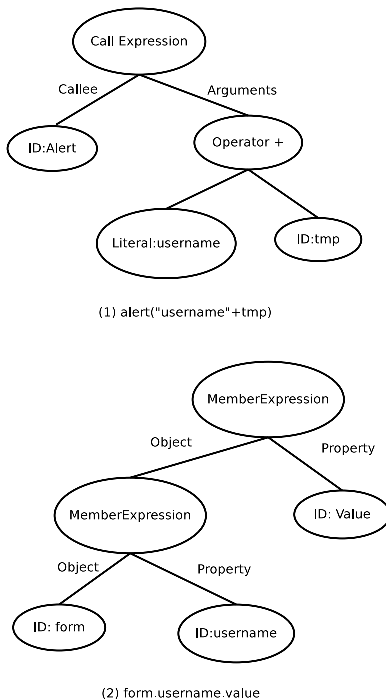

Fig. 10. AST examples

an HTTP POST request arrives, PolyRef decrypts the mapping and restores the names in the name-value pairs of the HTTP POST back to original values so the underlying web application requires no changes to work with the PolyRef. Note the encryption key is a only known by PolyRef and can be periodically rotated.

7 Evaluation

We designed three experiments to evaluate the prototype implementation of PolyRef. The first two experiments tested the effectiveness, and the third experiment tested the performance. We will discuss future attacks in Section 8.

7.1 Fake Account Creation Attack

In this Section, we first demonstrate a fake account creation attack, and then show the result after applying PolyRef. We examined the Top 10 Alexa websites, found four of them (Facebook, Yahoo, Twitter, and LinkedIn) did not require CAPTCHAs in the account creation page. We used Facebook as an example for our attack. To avoid directly attacking Facebook, we mirrored the front page (login and account creation page) of Facebook.com. We wrote a simple back-end which stored the created accounts in a database.

```
def create_fake_account(ssURL , ssFN , ssLN , ssMail , ssPass ,ssDomTag ,
    ssBm , ssBd , ssBy , ssSx ):

  ssBrowserDriver = webdriver.Firefox();
  ssBrowserDriver.get(ssURL);

  ssLoginSignin = ssBrowserDriver.find_element_by_id("u_0_2");
  ssFirstName = ssBrowserDriver.find_element_by_name("firstname");
  ssLastName = ssBrowserDriver.find_element_by_name("lastname");
  ssEmail = ssBrowserDriver.find_element_by_name("reg_email__");
  ssEmailConfirm = ssBrowserDriver.find_element_by_name("reg_email_confirmation__");
  ssPassword = ssBrowserDriver.find_element_by_name("reg_passwd__");

  ssFirstName.send_keys(ssFN);
  ssLastName.send_keys(ssLN);
  ssEmail.send_keys(ssMail);
  ssEmailConfirm.send_keys(ssMail);
  ssPassword.clear();
  ssPassword.send_keys(ssPass);
  // Birth year, month and day are removed in the code due to space limit

  ssLoginSignin.click();
```

Fig. 11. Fake account creation function. It begins by creating a Firefox driver and visiting the account creation page. It then uses the web driver API `find_element_by_id` and `find_element_by_name` to find all fields of the account creation form. It fills the fake account data into the fields by `send_keys` API and `pick_select_item` and `pick_radio_item` functions. Finally, it clicks the form submission.

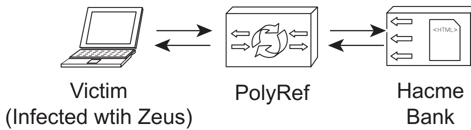

Fig. 12. Zeus MitB experiment setup

```
set_url *hacmebank*main.aspx?function=AccountTransfer* GP

data_before
value="Transfer" onclick="
data_end
data_inject
doTransfer();
data_end
data_after
data_end

data_before
  <body>
data_end
data_inject
<script>
  function doTransfer() {
    var destination_account = "5204320422040005";
    var tform = document.getElementById("WelComeForm");
    tform.ctl03$InternalOrExternalPayment[0].checked = false;
    tform.ctl03$InternalOrExternalPayment[1].checked = true;
    tform.ctl03_txtExternalPaymentAccount.value=destination_account;
  }
</script>
data_end
data_after
data_end
```

Fig. 13. Page injection Config for Hacmebank. The config tells Zeus to find the Hacme bank transfer account page to inject two pieces of code. The first one hijacks onclick function of "transfer" button. The second one performs the malicious transfer: replaces the transfer destination to the attacker's account 05. Parameter `set_url` sets the attack target; Parameter `data_before` describes the data to search for before the injection; Parameter `data_inject` is the actual script that will be injected.

The attack is written as a Python script using Selenium Webdriver API. Selenium is a software testing framework for web applications. The attack script exploits Selenium Webdriver API to drive a Firefox browser to launch the attack. Fig. 11 shows the source code of the fake account creation function. It only contains tens of lines of Python code. Note that the Python import header is ignored for brevity.

We used the attack script to attack our mirrored Facebook.com and we successfully created 1000 accounts. We deployed a PolyRef in front of the mirrored Facebook.com. We launched the attack again, and all 1000 attempts failed. We tested the account creation manually through the user interface and creation still works with PolyRef deployed.

7.2 Zeus MitB Attack

In this experiment, we show how Zeus performs a Man-in-the-Browser attack. Then, we demonstrate how our PolyRef blocks Zeus's attack.

Experiment Setup. Fig. 12 shows the setup of this experiment. We used McAfee's Hacme Bank, a security training bank application built on Microsoft

IIS/.Net framework. It is a database driven application. Hacme Bank provides a minimal representation of a financial institution such as authenticated accounts with balances and fund transfers between accounts.

Alex is a customer of Hacme Bank. Unfortunately, Alex's laptop is infected with Zeus malware. When Alex logs into Hacme Bank, the Zeus malware hijacks the session by page injection and secretly transfers money to attacker Mallet. One benefit of page injection is that Zeus is able to bypass two-factor authentication.

Victim Alex's machine is installed with Windows 7, Service pack 1, and IE 10 is installed and used as the browser. The victim machine was infected with our custom Zeus, created with a Zeus 2.0.8.9 builder. It is configured with a page injection file webinjects.txt, shown in Fig. 13.

Experiment. In the experiment, we show account transfers from a clean machine and a machine infected with Zeus. We created user Alex and two accounts 01 and 02. We also deposited $10000 and $100 into these two accounts, respectively. We started the victim machine without Zeus infection. We logged into Hacme Bank as Alex. We did an account transfer: $1000 from account 01 to 02. The account balance of account 01 and 02 became $9000 and $1100, respectively.

Then, we infected the victim machine with the Zeus sample created in our setup. We did another transfer after infection: $500 from account 01 to 02. The account balance shows $500 was deducted from account 01, but the balance of account 02 did not increase. The transaction details show the transfer destination is account 05 instead of account 02. This transaction was hijacked by Zeus.

We then deployed the PolyRef. We did a final transfer: $400 from account 01 to account 02. We checked the account balance and transaction details. There are no malicious transactions. The result shows PolyRef successfully deflected the Zeus MitB attack. The Zeus MitB attack failed to reference the account transfer form `WelComeForm` and the payment destination field `ctl03_txtExternalPaymentAccount` in the `doTransfer` function (Fig. 13) because of the reference polymorphism.

7.3 Performance

In this experiment, we showed that the additional latency to deliver the login page for a popular e-commerce website is very low—if we cache the results of our first-time analysis. We ran PolyRef on a laptop with a quad-core Intel CPU and 16G memory. We used the laptop as a proxy to visit the login page of the tested website, stubhub.com[3]. The average additional latency to load the login page is shown in Table 2. We tested the latency with 1 to 32 concurrent threads in 6 tests, and each test was performed for 60 seconds. We used Apache JMeter to measure the result. Note that we started our test after page analysis cache was created during the first visit. Although the first-time analysis of the page (particularly the JavaScript) is time-consuming, it only needs to be done once.

[3] One of the largest online ticket marketplaces.

Table 2. The latency generated by PolyRef

Concurrent threads	1	4	8	16	24	32
Average latency (ms)	4	4	6	10	13	13

8 Discussion

We view PolyRef as the first step in polymorphic defense for websites. Adversaries, once they learn about PolyRef, may be able to tailor their attack techniques to target PolyRef's current defenses. In this section we discuss potential challenges and next steps, and we encourage future research on polymorphic defenses for web security.

8.1 Attacker Response

Field polymorphism raises the bar for advanced DOM attacks. An adversary may respond to field polymorphism by automatically extracting the sequence code from the dynamically generated JavaScript for each page served. It may be not hard for a skilled adversary to manually reverse the obfuscated JavaScript code of a page and extract the sequence code. However, DOM attacks have to perform automatic static analysis, as the sequence code is unique for each page served. Automatic static analysis of JavaScript is difficult, if not impossible, due to a number of dynamic features [16] and hard-to-understand semantic features [12] of JavaScript. In addition, the dynamically generated JavaScript can be highly obfuscated which makes the automatic analysis even harder. Obfuscation techniques including, but not limited to, variable and function name replacement, dead code insertion, encryption, string and number encoding, eval hiding, and opaque predicates, can be used to impede future attacks.

Eval hiding — hides the usage of eval function. Eval is commonly used to run code stored as a string in a variable which makes static analysis hard or even impossible. To hide uses of eval function, eval functions are assigned to various randomly named variables.

Opaque predicate — is defined as an expression for which the outcome is predetermined to be always true or false. The most simple example of this is expression *if (true)*. Opaque predicates can thwart static analysis by constructing expressions that are not so simple to determine without evaluating them inside the targeted environment.

8.2 Limitations

The limitation of PolyRef is that it does not prevent GUI attacks (where the attacker controls a real browser and interacts by directing mouse movement and keystrokes) as defined in Section 2. A GUI attack is equipped with a fully functional browser and sends OS mouse and keyboard events to the browser to

simulate human interaction. It positions the input focus by x, y coordinates or relative vectors, and then streams keystrokes into the field of focus. As the attack relies on fixed x, y coordinates of web UI elements, a new type of polymorphism—view polymorphism, where view will be varied for each page served, may be used to impede such attacks. For example, locations of forms and fields can be changed slightly for each HTTP request, so that the x and y coordinates are unpredictable. In practice, moving a form slightly down in the browser may not affect the user experience. On the other hand, as mentioned in Section 2, the attack relies on simulating keyboard and mouse activity so the behavioral biometric method [13, 14] mentioned in Section 3, which can tell the difference of mouse and keystroke behaviors between human and bot, could be used to complement PolyRef. Finally, the impact of GUI attacks is limited; that method does not enable CSRF or non-persistent XSS attacks.

9 Conclusion

We propose PolyRef, a method for a polymorphic defense to defeat automated attacks on web applications. PolyRef broadly deflects many types of automated attacks. As a preventive technique, it does not have false positive or false negatives. Further, the PolyRef concept is transparent to the web server, and most importantly, it has no deleterious user impact.

References

1. Belgisch gerecht ontdekt oplichting bij internetbankieren (2010) http://www.hbvl.be/nieuws/economie/aid956766/belgisch-gerecht-ontdekt-grootschalige-bankfraude.aspx
2. BIG-IP application security manager (2013), http://www.f5.com/pdf/products/big-ip-application-security-manager-ds.pdf
3. Multi-factor authentication (2013), http://en.wikipedia.org/wiki/Multi-factor_authentication
4. Mykonos web security (2013), http://www.mykonossoftware.com
5. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 75–88. ACM, New York (2008)
6. Chu, Z., Gianvecchio, S., Koehl, A., Wang, H., Jajodia, S.: Blog or block: Detecting blog bots through behavioral biometrics. Comput. Netw. 57(3), 634–646 (2013)
7. Chu, Z., Gianvecchio, S., Wang, H., Jajodia, S.: Who is tweeting on twitter: Human, bot, or cyborg? In: Proceedings of the 26th Annual Computer Security Applications Conference. ACM, New York (2010)
8. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: Proceedings of the Usenix Security Symposium 2003, Berkeley, CA, USA, pp. 243–255. USENIX Association (2003)

9. Czeskis, A., Moshchuk, A., Kohno, T., Wang, H.J.: Lightweight server support for browser-based csrf protection. In: Proceedings of the 22nd International Conference on World Wide Web, WWW 2013 Companion, Republic and Canton of Geneva, Switzerland, pp. 273–284. International World Wide Web Conferences Steering Committee (2013)
10. Eckersley, P.: How unique is your web browser? In: Atallah, M.J., Hopper, N.J. (eds.) PETS 2010. LNCS, vol. 6205, pp. 1–18. Springer, Heidelberg (2010)
11. Fontana, J.: Password's rotten core not complexity but reuse (March 2013), http://www.zdnet.com/passwords-rotten-core-not-complexity-but-reuse-7000013019/
12. Gardner, P.A., Maffeis, S., Smith, G.D.: Towards a program logic for JavaScript. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 31–44. ACM, New York (2012)
13. Gianvecchio, S., Wu, Z., Xie, M., Wang, H.: Battle of botcraft: Fighting bots in online games with human observational proofs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 256–268. ACM, New York (2009)
14. Gianvecchio, S., Xie, M., Wu, Z., Wang, H.: Measurement and classification of humans and bots in internet chat. In: Proceedings of the 17th Conference on Security Symposium, SS 2008, pp. 155–169. USENIX Association, Berkeley (2008)
15. Heiderich, M.: Csrfx (2007), http://php-ids.org/category/csrfx/
16. Jensen, S.H., Jonsson, P.A., Møller, A.: Remedying the eval that men do. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 34–44. ACM, New York (2012)
17. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: Second IEEE Communications Society/CreateNet International Conference on Security and Privacy in Communication Networks. IEEE (2006)
18. Kee, T.: Beyond cookies: digital fingerprints may track personal devices (December 2010), http://econsultancy.com
19. Miessler, D.: Bypassing WAF anti-automation using burp's cookie jar (September 2013), http://www.danielmiessler.com
20. Ollmann, G.: Stopping automated application attack tools. Technical report, Black Hat Europe 2006 (2006)
21. Sheridan, E.: OWASP CSRFGuard project (2008), http://www.owasp.org/index.php/CSRF_Guard
22. von Ahn, L., Blum, M., Hopper, N.J., Langford, J.: CAPTCHA: using hard ai problems for security. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 294–311. Springer, Heidelberg (2003)
23. Yan, J., El Ahmad, A.S.: Usability of CAPTCHAs or usability issues in CAPTCHA design. In: Proceedings of the 4th Symposium on Usable Privacy and Security, SOUPS 2008, pp. 44–52. ACM, New York (2008)