TamperProof: A Server-Agnostic Defense for Parameter Tampering Attacks on Web Applications

Nazari Skrupsky
University of Illinois at Chicago

Prithvi Bisht
University of Illinois at Chicago

Timothy Hinrichs
University of Illinois at Chicago

V. N. Venkatakrishnan
University of Illinois at Chicago

Lenore Zuck
University of Illinois at Chicago

ABSTRACT

Parameter tampering attacks are dangerous to a web application whose server performs weaker data sanitization than its client. This paper presents TamperProof, a methodology and tool that offers a novel and efficient mechanism to protect Web applications from parameter tampering attacks. TamperProof is an online defense deployed in a trusted environment between the client and server and requires no access to, or knowledge of, the server side codebase, making it effective for both new and legacy applications. The paper reports on experiments that demonstrate TamperProof's power in efficiently preventing all known parameter tampering vulnerabilities on ten different applications.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Verification
; K.4.4 [**Electronic Commerce**]: Security
; K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Languages, Security

Keywords

Parameter Tampering; Prevention

1. INTRODUCTION

Interactive processing and validation of user input are increasingly becoming the de-facto standard for Web applications. With the advent of client-side scripting there has been a rapid transition in recent years to validate user input in the browser itself, before it is submitted to the server. This client-side validation offers numerous advantages, among which are faster response time for users and reduction of load on servers. Yet, client-side validation exposes new vulnerabilities as malicious clients can circumvent it and supply invalid data to the server. A server that accepts such invalid data is vulnerable to *parameter tampering* attacks.

Several recent studies [5, 6, 18, 3] have presented automated techniques to *detect* parameter tampering and have uncovered parameter tampering vulnerabilities in both open source and commercial websites, most notably in websites for banking and on-line shopping, as well as those accepting payments through third party cashiers (such as `PayPal` and `AmazonPayments`.) These vulnerabilities enable takeovers of accounts and allow a malicious user to perform unauthorized financial transactions.

Defense against parameter tampering attacks for new applications is conceptually straightforward: Ensure that the input validation of the server is at least as strong as that of the client. Legacy applications are more challenging since manually editing source code to ensure the server's validation is as strong as the client's is error prone, expensive, and must be performed each time the code is modified. We therefore seek automated solutions to defend against parameter-tampering attacks for both new and legacy applications.

A major hurdle to an automated solution is that for most web applications the client's code changes from one HTTP request to the next. Each time the server receives a request, it responds with a client program that is generated based on the data the server received, the server's session state, and the database state. Each time a server protected against parameter tampering receives an HTTP request, it must therefore not only perform validation on the data it receives but also determine which of the potentially infinitely many possible clients sent the data so as to perform the proper validation.

The existing defense literature has focused on preventing popular attacks on web applications such as SQL injection, Cross-site scripting, etc., and does not address challenges posed by parameter tampering vulnerabilities. A large body of work exists for preventing client-side cheating in online games e.g., [4, 10]. These techniques cannot be applied to web application setup: unlike game clients which have fixed code, web forms are dynamically generated based on server-side state. Other relevant works include Guha et al., [11] and Ripley [17]. The work in [11] computes a model of expected flow of requests by analyzing client side code of web applications. As with the case of online game client code, properties such as the sequence of requests do not change over time and hence this technique cannot be applied to prevent parameter tampering attacks. Ripley [17] relies on executing another copy of the client code in a trusted environment to detect

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

tampering attempts from the actual client. Ripley seems to require complex engineering in keeping replicated client executions in sync. As it does not analyze client-side validation it may fail to prevent certain class of parameter tampering attacks (See Section 7). Given the severity of parameter tampering vulnerabilities, further research is warranted to prevent them.

The goal of this work is to offer an efficient defense against parameter tampering attacks that addresses the challenges above. We describe TamperProof, a transparent, automatic solution for the *prevention* of parameter tampering attacks. TamperProof protects both legacy and new applications without requiring access to, knowledge of, or changes to existing code. TamperProof is deployed in a trusted environment between client and server and intercepts all communication between them. We demonstrate that the overheads of TamperProof are similar to those of a standard web proxy. (For a live demonstration, see the anonymous website [2].)

This paper makes the following contributions:

- A method to determine the validation that ought to be performed on the server for any given HTTP request;
- Design and implementation of a tool for preventing parameter tampering attacks;
- A study of resiliency and efficiency issues of the method to ensure robustness of the defense and acceptable overheads;
- An evaluation over a large corpus of medium to large web applications.

The rest of the paper is organized as follows. Section 2 provides an overview of parameter tampering attacks and uses a simple web application to illustrate challenges in preventing them. Section 3 provides an overview of TamperProof. Section 4 elaborates on TamperProof's design and discusses challenges that were addressed to ensure security and acceptable performance. Section 5 describes implementation of TamperProof and our experiences in evaluating it over a large corpus of vulnerable web applications. Section 6 is a discussion of Web 2.0 applications. We compare TamperProof to relevant literature in Section 7 and conclude in Section 8.

2. BACKGROUND

A Running Example. Figure 1 shows three forms that are part of a typical online purchase process. (This example is based on a real-world parameter tampering exploit found on a shopping website, which was reported in [5].) The Shopping Cart form (first on the left) shows two products selected by a user for purchase and solicits a quantity for each product, a credit card to be charged (displayed in a drop down list of previously used cards) and any special delivery instructions. When the user submits this form, the client-side JavaScript verifies that the specified quantities for the selected products are positive and delivery instructions contain 30 characters or less. If any of these conditions are violated, the JavaScript code cancels the submission and prompts the user to correct the mistakes; otherwise, the user inputs $\texttt{quantity}_1$, $\texttt{quantity}_2$, `card` and `delivery` are sent to the server. The server then computes the total cost of the requested products and generates the Shipping Details form. This form asks the user where and how the products should be shipped. When the user submits this form her shipping information is sent to the server, who computes the shipping cost, adds it to the total cost, and generates a (read-only) Confirmation form. Once the user confirms her purchase by submitting the Confirmation form, the server places her order and the transaction is complete.

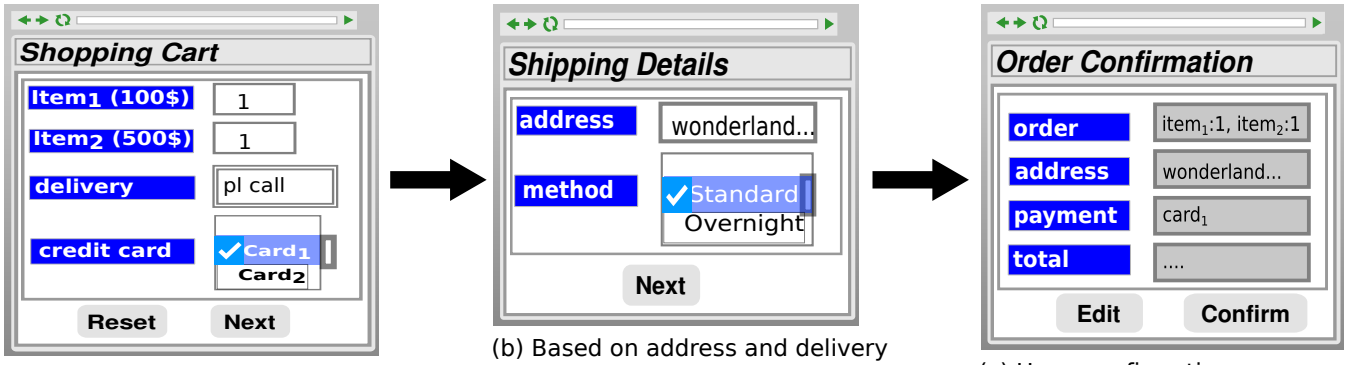
Basic Parameter Tampering Attack. Suppose the server-side code that processes the shopping cart submission fails to check if the values of the inputs $\texttt{quantity}_1$ and $\texttt{quantity}_2$ are positive numbers. In this form, a malicious user can bypass client side restrictions (by disabling JavaScript) and submit a negative number for one or both products. It is possible that submitting a negative number for both products would result in the user's account being credited; however, that attack will likely be thwarted because of differences in credit card transactions on the banking server responsible for debit and credit operations. However, if a negative quantity is submitted for one product and a positive quantity is submitted for the other product so that the resulting total is positive, the negative quantity acts as a discount on the total price. For instance, in Figure 1, if the values for $\texttt{quantity}_1$ and $\texttt{quantity}_2$ were -4 and 1 respectively, the end result would be an unauthorized "discount" of $400.

Negative Parameter Tampering Attack. Now suppose that the web application is designed to give all employees a 10% reduction in their total costs. One (poor) way to implement this feature is to include on every employee form a hidden field $\texttt{ediscount} = 1$ that when present causes the server to subtract 10% from the total price. A malicious user (non-employee) can launch a negative parameter tampering attack to gain the employee discount by modifying her form to include $\texttt{ediscount} = 1$ (a field not originally present on the form).

Tampering based Sequencing Attack. Finally consider the sequence of forms the server intends the user to navigate: the Shopping Cart, the Shipping Details, and the Confirmation. If the server does not force the user to follow this sequence, it is vulnerable to a *sequencing* attack, wherein the user can skip one or more steps in the above sequence. For example, a malicious user could manually submit the Confirmation form and choose the products, quantities, shipping costs, and total cost. This attack (a generalization of the attack on quantities described earlier) enables a malicious user to drive the total cost to a value close to zero.

2.1 Challenges in Preventing Parameter Tampering Attacks

To prevent parameter tampering attacks in legacy applications, one must augment the server-side input validation such that it is as strong as the corresponding client-side validation. In our running example, parameter tampering vulnerability of the parameter `quantity` can be patched by checking and rejecting negative values of the parameter `quantity`. Intuitively, this check can be performed after the form is submitted and before the parameter `quantity` is used. More specifically, the following check must be performed before `quantity` field is used in any server-side computation: exit if ($\texttt{quantity} < 0$ AND $\texttt{op} ==$ "purchase")



(a) Server computes total cost of selected items

(b) Based on address and delivery method, server adds shipping cost to the total cost

(c) Upon confirmation, server adds this order for processing

Figure 1: Web forms in a typical order processing functionality: (a) first form solicits items to be purchased, delivery instructions and credit card to be charged, (b) second form solicits shipping address and shipping method, and (c) third form seeks user confirmation, displaying all order details in a read-only manner.

i.e., forbid execution if value of the parameter `quantity` is negative and the requested operation is "purchase".

The process of generating and enforcing the above checks (intuitively the patch) faces two main challenges: (a) as web forms may change over time, this computation must factor in dynamism in form generation (challenge $\mathcal{C}_1$), and (b) as parameter tampering vulnerabilities may only be present in specific executions of the server-side code, patches should *only* be triggered in applicable executions (challenge $\mathcal{C}_2$). We discuss these challenges in detail below.

Challenge 1: Factoring in Dynamism in Form Generation ($\mathcal{C}_1$). Perhaps the biggest challenge for generating correct patches stems from the dynamic nature of web applications. Quite frequently, web applications use server-side state (database, files, etc.) to create web forms. In our running example, the `card` drop down menu is populated from a database that stores credit cards used by a user in the past. As the server-side state changes, such form fields change and in turn may imply a different set of constraints. In our running example, if the user makes use of a new credit card say $\texttt{card}_3$, the constraint implied by the drop down menu changes from $\boxed{\texttt{card} \in \{card_1, card_2\}}$ to $\boxed{\texttt{card} \in \{card_1, card_2, card_3\}}$. In general, a form field may encode state information that is different across user sessions but could also change within a session over time (for example, different users will likely have different credit cards, but even a single user may use new cards during the lifetime of a session). Consequently, the challenge is in computing patches that also evolve with changes in server-side state.

Challenge 2: Patching Only Vulnerable Server-side Executions ($\mathcal{C}_2$). A parameter tampering vulnerability may be specific to certain executions of the server-side code. It is possible that some or all of the other executions either don't use the vulnerable parameter (and hence aren't vulnerable) or use it in security insensitive operations. Hence even when a patch is computed, it is necessary to compute the exact conditions that must be met to enforce the patch. In the running example the server-side execution may only use parameter `quantity` when the requested operation is purchase. Intuitively, such conditions identify vulnerable executions of the server code and hence when to enforce missing checks (or patch) e.g., before requiring `quantity` field to have positive values, the patch must ensure that the requested operation is purchase. Without the knowledge of vulnerable executions, a patch may result in false positives e.g., by always demanding a positive value for the parameter `quantity`. The key challenge is in finding mechanisms that can identify vulnerable executions of the server to precisely enforce patches.

3. OVERVIEW

3.1 Problem Description

The above discussion illustrates the basic nature of parameter tampering attacks. We now formalize these attacks as violations of constraints the server intended to impose on the submission and processing of user inputs.

DEFINITION 1 (INPUT). *We define an* input *received by a server from a client as a set of field name/value pairs:* $\mathcal{I} = \{\langle \mathcal{N}_1, \mathcal{V}_1 \rangle, \langle \mathcal{N}_2, \mathcal{V}_2 \rangle, \ldots, \langle \mathcal{N}_n, \mathcal{V}_n \rangle\}$

A server receives a sequence

$$\mathcal{I}_1, \quad \mathcal{I}_2, \quad \mathcal{I}_3, \quad \ldots, \quad \mathcal{I}_k, \ldots$$

of such inputs, and each time a new input arrives the server must decide whether or not the sequence up to that point in time constitutes an attack on the server and accept or reject the new input accordingly.

Intuitively, a malicious user can launch an attack in three conceptually separate ways: tamper with the field *names* in the inputs, tamper with the field *values* in the inputs, or tamper with the *order* in which inputs arrive. Each type of attack violates a constraint the server intended to enforce on its inputs—a constraint defined by the form used to submit each input (if one exists). Thus, we associate with every input $\mathcal{I}$ the form $\mathcal{F}_\mathcal{I}$ used to submit it. Below we describe the three constraints corresponding to each attack: the Field Constraints, the Value Constraints, and the Sequencing constraints, respectively.

Field Constraints. Field Constraints dictate which field names an input is allowed to include. The Field Constraints for input $\mathcal{I}$ are usually implicit in the form $\mathcal{F}_\mathcal{I}$ be-

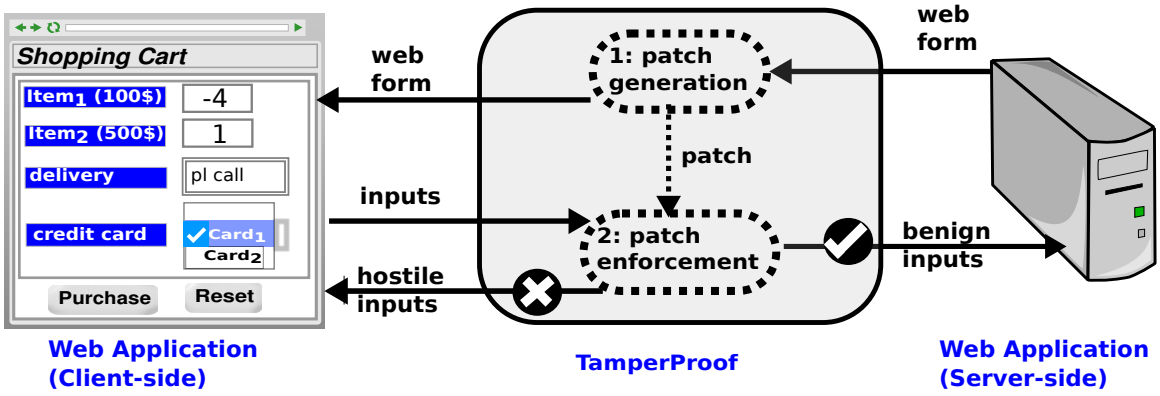


Figure 2: Overview of TamperProof: Per form patch generation and enforcement: 1) Each form is analyzed to generate a patch, and 2) form submissions conforming to this patch are submitted to the server for further processing; otherwise rejected and sent to the client.

cause the only field names the form submits are those included in it.

> Given a form $\mathcal{F}$ and input $\mathcal{I}$, we say that $\mathcal{I}$ satisfies the Field Constraints of $\mathcal{F}$ if the set of field names in $\mathcal{I}$ is a subset of fields present in $\mathcal{F}$.

Enforcing Field Constraints on our running example prevents the employee discount attack.

Value Constraints. Value constraints dictate which data values can be assigned to which field names in an input. The Value Constraints for input $\mathcal{I}$ are enforced by form $\mathcal{F}_\mathcal{I}$ either through its JavaScript (e.g., the product quantities in the Shopping Cart form) or its HTML (e.g., the length restriction on the delivery instructions).

> Given a form $\mathcal{F}$ and input $\mathcal{I}$, we say that $\mathcal{I}$ satisfies the Value Constraints of $\mathcal{F}$ if when $\mathcal{I}$'s values are inserted into $\mathcal{F}$'s fields, the form allows the input to be submitted to the server.

Enforcing Value Constraints in our running example prevents the product quantity attack described earlier.

Sequencing Constraints. These constraints ensure that inputs are sent to the server in a plausible order. By plausible we mean that at the time input $\mathcal{I}$ is submitted, form $\mathcal{F}_\mathcal{I}$ has already been generated by the server. This simple condition ensures that many of the server's intended sequence of requests are respected. If the server intends for form A to precede form B, then it is likely that form B will only be generated in response to the inputs from form A.

> We say input sequence $\mathcal{I}_1, \mathcal{I}_2, \ldots$ satisfies the Sequencing Constraints if for every input $\mathcal{I}_j$, $\mathcal{F}_{\mathcal{I}_j}$ was generated before $\mathcal{I}_j$ was submitted.

Enforcing the Sequencing Constraints prevents the sequencing attack in our running example where the Confirmation form is submitted before the Shipping Details form. The only way to generate the Confirmation form is by submitting the Shipping Details form and thus at the time the attacker submits the Confirmation form's inputs, that form has not yet been generated.

3.2 Approach Overview

The key idea in our approach is to dynamically infer and enforce Sequencing, Field, and Value constraints on each input submitted to the server. This avoids the precision issues associated with static analysis and learning approaches. Dynamic approaches come at the cost of some performance, and we develop techniques to improve performance in Section 4.2.

Once constraints are learnt, they are placed as patches (i.e., filters) at a web application proxies that enforce these constraints on incoming inputs from clients. Enforcing these patches in a proxy simplifies our implementation and has the added benefit that our approach is applicable regardless of the platform used to implement the server.

To infer and enforce constraints on server inputs, our approach uses the following ideas.

Enforcing Sequencing Constraints. To enforce these constraints, we ensure that every non-malicious input is mapped to the form used to submit the input. To implement this idea, we dynamically instrument every form generated by the server to include a hidden field with that form's identifier. Any input that arrives at the server without such an identifier (or with a spurious identifier) is rejected. Further, each such identifier is used only once i.e., a form's identifier expires when the form is submitted with inputs that satisfy client-side validation.

Enforcing Field and Value Constraints. To enforce these constraints, we verify that every non-malicious input could have been submitted by the form associated with that input (as described above). To implement this idea, we dynamically analyze each form generated by the server to extract the constraints enforced by HTML and JavaScript and record which form identifier corresponds to which constraints. Any input that arrives at the server and does not satisfy the constraints corresponding to the form used to submit the input, is rejected.

The above observations are developed in our approach and the tool TamperProof that prevents parameter tampering attacks in existing applications. In essence, TamperProof generates *a per form patch* by analyzing client side code of each form generated by a web application and then uses it to forbid parameter tampering attempts when the form is

submitted. Figure 2 presents a functional overview of these two steps, which we discuss in more detail below.

1. Patch Generation. In the first step, TamperProof intercepts web pages being sent to a client and locates any web forms in them. In each form TamperProof inserts a (randomly generated) unique form identifier, which for brevity we call the patchID. TamperProof then extracts the Field and Value constraints enforced by this form and associates them with the patchID. More precisely, TamperProof first analyzes the HTML to extract the Field constraints as well as a few Value constraints. It also analyzes JavaScript (using standard symbolic evaluation techniques e.g., [15, 5]) to extract the remaining Value constraints. Intuitively the combination of the patchID, the Field, and the Value constraints represents the patch for this form.

For the Shopping Cart form in our running example, the Field constraints TamperProof extracts is a simple set of field names: $\{quantity_1, quantity_2, card, delivery\}$. The Value constraints are captured by the formula:

$$\bigwedge \begin{array}{l} quantity_1 \geq 0 \wedge quantity_2 \geq 0 \\ delivery \in [\texttt{a} - \texttt{zA} - \texttt{Z}]* \\ card \in (card_1 \mid card_2) \end{array}$$

Note that the permitted credit card selections depend on the application's backend database, which means that if the user requests the Shopping Cart form twice her credit card options may be different each time. Because patch generation is performed each time a form is generated, the Value constraints will always properly reflect the user's actual options at the time she submits her data. As the patches are generated for each web form (which capture the server-side state at the time of their generation and hence the most up-to-date constraints), the above mechanism effectively addresses the challenge posed by dynamism in form generation (challenge $\mathcal{C}_1$).

We also note that the patchID is a nonce similar to *tokens* used in Cross-site request forgery (XSRF) prevention solutions such as [13, 12]. Such tokens are used to identify and reject unsolicited HTTP requests. patchID is used for a different purpose – to associate each generated web form with its submission. We discuss implications of similarities between patchID and XSRF tokens in Section 7.

2. Patch Enforcement. In the second step, TamperProof intercepts form submissions before they reach the server. TamperProof checks that the supplied patchID exists (thus preventing sequencing attacks) and that the supplied inputs satisfy the Field and Value constraints extracted in the previous step. A form submission is rejected if it fails any of these checks; otherwise, it is forwarded to the server for normal processing.

The following two form submissions show values assigned to various fields in the Shopping Cart form. TamperProof forwards the first submission to the server because it satisfies all the Field and Value constraints, but it rejects the second submission because $\texttt{quantity}_1$ is negative, violating the Value constraints.

1. $\{quantity_1 \rightarrow 1, quantity_2 \rightarrow 1, delivery \rightarrow$ "*call*", $card \rightarrow card_2\}$
2. $\{quantity_1 \rightarrow -1, quantity_2 \rightarrow 1, delivery \rightarrow$ "", $card \rightarrow card_1\}$

The addition of patchID in forms, affords TamperProof an effective mechanism to tie web forms (and constraints they imply) with their submissions (and hence constraints they must satisfy). This mechanism enables TamperProof to precisely check constraints specific to a web form that tampered form submissions will fail to satisfy, thus providing an effective mechanism to safeguard *only* the vulnerable executions of server code (challenge $\mathcal{C}_2$).

4. SECURITY AND PERFORMANCE

The practical effectiveness of TamperProof depends crucially on it being secure against attacks and it performing well enough for real-world web applications.

4.1 Security

The previous section gave a conceptual description of what a patch for a form is, how it is generated, and how it is enforced. In this section we expand on these ideas to ensure that TamperProof is robust against a series of attacks that attempt to violate the Field, Value, and Sequencing constraints of the application as well as the mechanisms TamperProof uses to enforce those constraints.

The key insight is that inserting a patchID into a form gives a malicious user another field to tamper with. For TamperProof to be secure, it must ensure that a form's patchID field is itself tamperproof. Tampering with a patchID is useful because different forms have different constraints, and some constraints are more permissive (i.e., weaker) than others: weaker constraints reject fewer inputs and hence are better for attackers. By submitting a patchID for a form with weaker constraints, an attacker can try to fool TamperProof into accepting data that violate the constraints on her actual form.

It turns out that making a form's patchID tamperproof only requires expanding our notion of a patch to include one additional piece of information about each form: the URL to which that form submits its data. Thus, the patch generation phase must extract the target URL for each form and tie it to the form's patchID, and the patch enforcement phase becomes the following sequence of checks.

1. patchID exists
2. server has a record of patchID
3. the data fields are a subset of those for patchID
4. the URL is the same as that for patchID
5. the data satisfies the constraints for patchID

If any of the above check fails, TamperProof rejects the submission as a parameter tampering attack; otherwise, it forwards the submission to the web application as usual and deletes the patchID entry from memory. Below we describe how this enforcement algorithm defends against attacks.

Basic parameter tampering. Basic parameter tampering attacks are those where an attacker submits data to the server that violate the form's Value constraints. In our running example, a submission where $\texttt{quantity}_1$ is negative (to obtain an unauthorized discount) constitutes a basic parameter tampering attack. TamperProof rejects such an attack at Check 5.

Negative parameter tampering. Negative parameter tampering attacks are those that violate the form's Field

constraints (the set of permitted field names). In our running example, a non-employee submission that includes the `ediscount` field (to gain a 10% discount), would constitute a negative parameter tampering attack. TamperProof rejects such attacks at Check 3.

Sequencing attacks. A sequencing attack is one that violates the form's Sequencing Constraints. In our running example, submitting the Order Confirmation form before the Shipping Details form constitutes a sequencing attack. TamperProof rejects such attacks at Checks 1 and 2 because the out-of-order submission could not have been submitted from a properly generated form, all of which have patchIDs.

Replay. Replay attacks are those where two different submissions include the same patchID. TamperProof rejects such attacks with high probability because once it receives the first submission, it deletes that patchID from memory, and hence unless the patchID was regenerated (a low probability event) the second submission will be rejected because the patchID does not exist.

patchID Spoofing. patchID spoofing attacks are those where the attacker generates a brand new patchID, attempting to forge a valid submission. TamperProof rejects such attacks with high probability because patchIDs are randomly generated and are therefore unpredictable for an attacker (i.e., Check 2 fails). This defense has the added benefit of protecting against cross-site request forgery attacks (XSRF), since the patchID is effectively a one-time use XSRF token (i.e., Check 1 fails for XSRF attacks).

patchID Swapping. patchID swapping attacks are those where a malicious user has two legitimate forms, A and B, and then submits data for form A using the patchID from form B. In our running example, a user could request a Shopping Cart form and a Shipping Details form. Then she could choose her own shipping costs by submitting the Shopping Cart form with the field `shippingCosts` and the Shipping form's patchID.

For patchID swapping attacks, TamperProof either identifies the request as an attack and rejects it or forwards the request on to the application because the attacker could have generated exactly the same request without any tampering. The cases where the attack is rejected are simple: (i) the request includes fields that form B does not (and hence Check 3 fails), (ii) the request's URL differs from that of form B (and hence Check 4 fails), (iii) the data submitted violates B's constraints (and hence Check 5 fails). Note however that if the request is not rejected, this "attack" uses exactly the same fields and URL as form B, and satisfies the constraints on form B. The attacker can therefore generate exactly the same request without any parameter tampering by simply filling out form B directly. TamperProof therefore should and does forward the request on to the application. In our running example, if a user attempts to skip the Shipping Details form by supplying too low (or high) a shipping cost, she fails, but if she supplies exactly the right shipping costs, she succeeds.

If, however, the attacker cannot legitimately checkout form B, she can fool TamperProof into treating her submission of form A as if it came from B—a form the attacker may not be permitted to submit. (Of course, the server itself still treats the submission as if it came from A.) For example, suppose there are two versions of a single form: one displayed to authenticated users with less stringent client-side validation and one to the general public with more stringent validation. If an authenticated user leaks her form's patchID before she submits it, an attacker can convince TamperProof to apply the less stringent validation to her submission, even if she is an unauthenticated user. The effectiveness of TamperProof's defense therefore relies on the secrecy of "active" patchIDs, i.e., patchIDs of forms that have been sent to client but have yet not been submitted. Admittedly, requiring patchIDs be kept secret is a limitation of TamperProof; however, it is no more a limitation than today's mainstream defense against XSRF attacks: embedding tokens into web pages. If the secrecy of patchIDs proves to be a practical limitation of TamperProof's effectiveness, then so too the insertion of XSRF tokens will prove ineffective against XSRF attacks—something that today's web development community deems unlikely.

4.2 Efficiency

The other practical concern for TamperProof is whether or not it is sufficiently efficient to be deployed on real-world web applications. Here there are two main metrics of interest: the time required for the server to generate a single page (latency) and the number of requests the server can support per second (throughput). Below we describe the ways in which TamperProof has been designed to minimize its impact on both these metrics. (Another relevant metric is that of scalability: how easy it is to improve latency and/or throughput by adding hardware to the system. We do not discuss scalability below because the only techniques needed to scale TamperProof are straightforward adaptations of traditional web optimization techniques for load balancing.)

Latency. Latency reflects how a single user perceives the performance of a web application: it is the amount of time required for the web server to generate the results of a single HTTP request. TamperProof incurs some overhead for all HTTP requests because it uses a proxy, but the main overheads are when the server generates a web form or processes a web form submission.

For web form generation, TamperProof must analyze the HTML produced by the server (to extract the patch) as well as add a patchID to each form. The key insight to avoiding high latencies is that there is a (sometimes significant) window of time from when the user requests a form to when the user submits the form for processing. The only thing that must be done before the form is sent to the user is that its patchID must be inserted. The rest of the analysis can take place on the server while the user is filling out the form. For each form request, TamperProof returns the form after inserting a patchID and spawns another thread to do the patch extraction. Thus its latency overhead is the cost of inserting patchIDs, which is no more than the overhead of proxy-based XSRF prevention solutions [13, 12].

Throughput. Of course, the server must still analyze the HTML page to extract the expected fields, the target URL, and the constraints for each form—analysis that can significantly reduce a server's throughput (requests handled per second). The dominating cost in this analysis is extracting the constraints from the JavaScript embedded in each page (e.g., the quantity of each product must be positive).

Fortunately, not all of the JavaScript appearing in a form needs to be analyzed. TamperProof begins by isolating the JavaScript that effects form submissions into what we

call the page's JavaScript signature (the code run when the user submits the form). The JavaScript signature implicitly represents the set of constraints that are enforced by the form. Once that signature is identified, TamperProof applies symbolic execution to extract the constraints (see e.g., [5, 15] for details), a process that can be expensive because it may result in analyzing all possible code paths.

The key insight to reducing the overheads of JavaScript analysis is that in many web applications, much of the JavaScript is the same across web pages. More to the point, it is common that the JavaScript code for constraint checking is the same across many different pages (even if those pages differ significantly in terms of their HTML). In our running example, a profile page might allow a logged-in user to change her personal information. The JavaScript validation code will be the same regardless of which user is logged in, but the HTML constraints for each page may differ substantially, e.g., each user has her own list of previously used shipping addresses. This means that caching the results of JavaScript analysis can greatly improve throughput.

To this end, TamperProof caches the results of JavaScript analysis. Each time it generates the JavaScript signature for a page, it canonicalizes that signature (e.g., alphabetizing the list of function definitions) and checks to see if that canonicalized signature is in the cache. If so, it uses the constraints from the cache; otherwise, it performs JavaScript analysis and adds a new entry to the cache. To avoid memory problems stemming from web applications with many distinct JavaScript signatures, TamperProof limits the size of the cache (to a value chosen by the web developer), keeps counters representing the relative frequency with which cache entries are used, and replaces the most infrequently used cache entry when the cache becomes full. By adjusting the cache size, the developer can balance the needs of high-throughput and low-memory consumption.

Algorithm 1 TamperProof-toUser(html)

```
1: html := add-patchids(html)
2: fork(analyzeClient,html)
3: return html
```

Algorithm 2 analyzeClient(html)

```
1: for all forms f in html do
2:    js := javascript-signature(html)
3:    id := find-patchID (html)
4:    ⟨url, fields, constraints⟩ := codeAnalysis(html,js)
5:    patches[id] = ⟨url, fields, constraints⟩
```

4.3 Implementation Details

Algorithms 1, 2, and 3 describe TamperProof in more detail. TamperProof-toUser (Algorithm 1) runs whenever the proxy forwards a web page generated by the server to the user. It embeds patchIDs into all the forms and links on that page, forks off a thread to analyze that page, and returns the modified page, which is then returned to the user. analyzeClient (Algorithm 2) is the code that is run between a form request and the corresponding form submission: it extracts the patch for that form.

TamperProof-fromUser (Algorithm 3) runs every time the user submits a request to the server. When the user requests one of the entry pages (landing pages for web applications), TamperProof simply forwards it to the server as such requests could be made directly and thus may not legitimately have patchIDs. For a non-entry page, TamperProof either identifies a parameter tampering attack or returns the original request, which is then forwarded to the server.

Algorithm 3 TamperProof-fromUser(request)

```
1: if request.url ∉ Entries then
2:    id := request.data['patchID']
3:    if id ∉ patches.keys() then return error
4:    wait until patches[id] is non-empty
5:    ⟨url, fields, constraints⟩ := patches[id]
6:    if request.data.keys() ⊈ fields then return error
7:    if !sat(request.data,constraints) then return error
8:    patches.delete(id)
9: return request
```

TamperProof maintains two global variables shared by all of these algorithms: *patches* and a cache for JavaScript constraints (not shown). *patches* is a hash table keyed on patchIDs that stores the patch for each active patchID. All three algorithms access *patches*. TamperProof-fromUser uses *patches* to check if submitted data satisfies the necessary constraints; analyzeClient stores a new patch in *patches*. TamperProof-toUser implicitly updates *patches* so that all of the patchIDs added to an outgoing webpage have (empty) entries in *patches* to ensure TamperProof knows those patchIDs are valid.

The cache for JavaScript constraints (not shown) is a hash table keyed on JavaScript signatures that stores the constraints corresponding to those signatures. analyzeClient is the only one to manipulate that cache. When analyzeClient is invoked, it extracts the JavaScript signature from the webpage and relies on another routine codeAnalysis to do the actual analysis. That routine first consults the JavaScript cache to see if the constraints for the JavaScript have already been extracted and if so simply adds those constraints to the result of the HTML constraint extraction; otherwise, it extracts the constraints from the JavaScript and updates the cache.

5. EVALUATION

Implementation. We implemented TamperProof by extending NoForge [13] (a server-side proxy for preventing XSRF attacks) with 600 lines of PHP and 200 lines of Perl to include the algorithms TamperProof-toUser and TamperProof-fromUser described in Section 4.3. Checking that an input's data satisfies a form's Value constraints is performed by a Perl script created to check exactly those constraints. The code to generate the Perl script for a given set of constraints is 1K lines of Lisp code. The implementation of analyzeClient is 5K lines of JavaScript code and 2K lines of Java code.

Applications. Our test suite is comprised of 10 medium to large PHP web applications that contained 49 parameter tampering vulnerabilities. These applications were previously evaluated [6]. Table 1 provides background information on these applications: lines of code, number of files, and functionality. The test suite was deployed on a virtual machine (2.4 GHz Intel dual core, 2.0 GB RAM) running

Application	Size (LOC)	Files	Use	Exploits (Patched /Total)
DcpPortal	144.7k	484	Contnt Mgmt	32/32
Landshop	15.4k	158	Real Estate	3/3
MyBloggie	9.4k	59	Blog	6/6
Newspro	5.0k	26	News Mgmt	1/1
OpenDB	100.2k	300	Media Mgmt	1/1
PHPNews	6.4k	21	News Mgmt	1/1
PHPNuke	249.6k	2217	Contnt Mgmt	1/1
SnipeGallery	9.1k	54	Img Mgmt	2/2
SPHPBlog	26.5k	113	Blog	1/1
OpenIT	146.1k	455	Support	1/1

Table 1: Applications & Attack Results

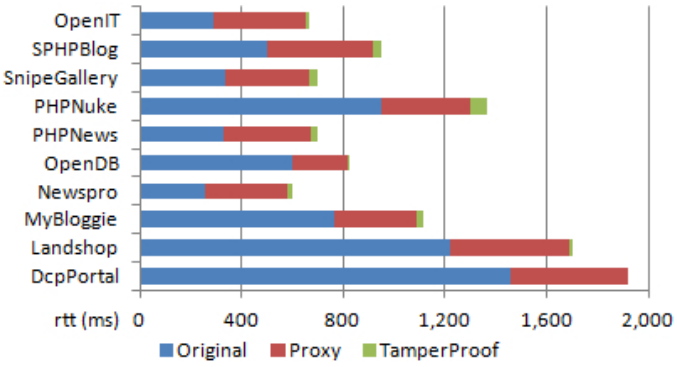


Figure 3: TamperProof: Incurred at most 4.8% overhead in round trip times.

Ubuntu 9.10 with the LAMP application stack and was connected via a local area network to the client (2.45Ghz Intel quad core, 4.0 GB RAM) running Windows XP.

5.1 Effectiveness

For evaluating the effectiveness of TamperProof in preventing parameter tampering exploits, we developed a `wget`-based shell script to generate HTTP requests to vulnerable web forms. The exact parameters needed to exploit each web form were manually provided to this automated script. This script was also manually given session cookies for forms that could only be accessed in authenticated sessions.

To test the correctness of our automated shell script, we tested each vulnerable web form without deploying TamperProof. Each form with tampered parameters that was successfully submitted and processed by the server confirmed the existence of a vulnerability as well as proper implementation of the shell script. We then deployed TamperProof and re-tested all of these applications. The result of this testing is summarized in the 5th column of Table 1, which shows the number of exploits that were prevented along with total number of known exploits for each form. As shown by this table, TamperProof was able to defend 100% of the known exploits. Below we discuss several exploits that represent the common types of vulnerabilities encountered in our test suite. The details of most of the tested exploits are available on the supplemental website [2].

Tampering HTML Controls. The OpenDB application is vulnerable to script injection through a tampered select input field. The root cause of this vulnerability is the server's failure to ensure that the submitted input belongs to one of the select box options available to the client. TamperProof detects inputs that are outside their allowed value range and also prevents attacks that involve tampering with other types of form fields including hidden fields, checkboxes, radio buttons, and text fields.

Tampering JavaScript Validation. The DcpPortal application fails to replicate the JavaScript validations on the server, allowing attackers to bypass a regular expression check and avoid mandatory fields during submission of a registration form. TamperProof captures JavaScript validation during constraint extraction and is therefore able to generate the appropriate patch to prevent such attacks.

Sequencing Attack. The PHPNuke application is susceptible to a sequencing attack that bypasses a CAPTCHA verification during the registration process. The application uses a hidden field in the form to control a user's registration progress. By tampering with this field, the CAPTCHA page can be skipped without affecting the registration process otherwise. TamperProof is able to prevent such sequencing attacks by rejecting tampered hidden field values.

Security of TamperProof. We also tested that the TamperProof solution itself could not be tampered with. To this end, we intercepted form submission in one of the applications from the test suite and conducted the following three attacks: (a) removed patchID, (b) replaced patchID in a form submission with a patchID from a prior submission, and (c) checked out two forms and swapped their patchIDs. The first two attacks were rejected as TamperProof either failed to find a patchID with the submitted form or the supplied patchID did not match any of the active patchIDs.

To test whether TamperProof protects again patchID swapping, we constructed a sample application that generated two forms, A and B, with the same fields: `name` and `age`. Form B forbids submission of empty values for both `name` and `age`, whereas form A forbids submissions where `age` is empty, i.e., validation for A is weaker than B.

In two separate browser windows we accessed the two forms. We then submitted form B with an empty `name` and the patchID for A, a submission that B would normally disallow. TamperProof forwarded this request to the server, despite the fact that we tampered with the patchID. Though this seems like a successful attack, the submission could just as easily have been created by filling out form A. Hence, had TamperProof rejected the submission, it would have also stopped a user from legitimately filling out form A.

The results from this experiment indicate that TamperProof allows requests that could be created without parameter tampering, but no more.

False Positives and False Negatives. TamperProof is guaranteed to not produce false negatives because the client code analysis engine used by TamperProof precisely models constraints implied by HTML code but conservatively approximates those implied by JavaScript. More specifically, this JavaScript engine conservatively assumes that all JavaScript event handlers relevant to validation were launched (which may not be the case in actual form submissions.) This enables TamperProof patches to be complete with respect to the HTML and JavaScript validation embedded in a form and subsequently enables it to prevent exploitation of all parameter tampering vulnerabilities.

False positives, on the other hand, may arise for the following reasons: (a) the JavaScript validation approximation computes stronger constraints than are actually enforced in the form or (b) the client JavaScript dynamically modifies the form, e.g., creates new input fields causing TamperProof to signal a negative parameter tampering attack. We tested TamperProof for false positives by submitting a wide variety of inputs, e.g., where optional fields were empty. TamperProof did not reject any valid submissions and in our evaluation was free of false positives.

One seemingly additional case for false positives occurs when the server sanitizes and accepts malicious inputs that are rejected by client. Since TamperProof rejects inputs the server can safely handle, it may seem to be a false positive; however, the only way to submit such inputs is via parameter tampering, and hence only impacts malicious users.

5.2 Performance

Latency. Since TamperProof performs additional processing both when a server sends a form to a user and when the user submits data to the server, we measured the overheads introduced by TamperProof for a combination of these events: the round-trip time (RTT). The RTT for a form is the sum of (a) the time from when a user requests a form to when she receives a response and (b) the time from when she submits form data to when she receives a reply. The time spent in filling out the form is *not* included.

For this experiment, we used an off-the-shelf, academic-grade proxy that includes XSRF protection. We deployed both the client and server on a LAN, simulating the worst-case scenario where network latencies are minimal and therefore RTT is almost entirely server performance. We then measured the RTT for forms from each of our applications.

Figure 3 shows the results. For each application, we measured the RTTs for the original application, the application with a server-side proxy with XSRF protection (but without TamperProof), and the application with the server-side proxy and TamperProof. The original application's RTTs are the blue portions of Figure 3; the overheads caused by the XSRF proxy without TamperProof are the red portions; the overheads caused by TamperProof are the green portions.

From the results, we observe that the bulk of the overhead is introduced by the XSRF proxy (32%-126%) and that the additional overhead for TamperProof is merely 0%-4.8%. This means that if an organization deploys an XSRF proxy defense, there is little added cost in deploying TamperProof. That said, the combined overhead of the XSRF defense and TamperProof is dominated by the cost of augmenting a form an XSRF tokens/a patchID. Table 2 Column 3 shows that this cost ranges from 170ms to 240 ms, which amounts to 15%-75% higher latency than the original application. The main conclusion we take away from these experiments is that a high-performance XSRF proxy (e.g., one built using an industrial-grade proxy with access to the byte stream) could be adapted to produce a high-performance version of TamperProof with little additional overhead.

Throughput. To understand the computational load added to a server by TamperProof (which influences throughput), we measured the processing times of TamperProof's internal components. For each application, Table 2 breaks down the processing times of the three core components:

Application	Patch Formu. Compl.	Processing Time (s)		
		form updt.	const extra. (w/ cache)	patch valid.
DcpPortal	187	0.22	14.68 (0.50)	0.01
Landshop	20	0.24	0.41 (0.41)	0.01
MyBloggie	37	0.22	5.66 (0.39)	0.01
Newspro	6	0.17	0.36 (0.36)	0.01
OpenDB	26	0.22	0.52 (0.52)	0.01
PHPNews	3	0.17	0.31 (0.31)	0.01
PHPNuke	11	0.18	1.15 (0.50)	0.01
SnipeGallery	11	0.16	1.47 (0.33)	0.01
SPHPBlog	37	0.18	2.41 (0.38)	0.01
OpenIT	17	0.22	0.64 (0.64)	0.01

Table 2: TamperProof: Other Evaluation Results

augmenting the form with patchID (Column 3), extracting constraints with and without a caching strategy (Column 4), and validating inputs (Column 5).

The form augmentation component averaged 197ms, though we think this component can be further optimized. The formula extractor component consumed the most time during processing. In our experience, HTML constraints were extracted at a constant time of about 300-600ms, whereas JavaScript constraints were extracted in time proportional to the formula size (Column 2) and ranged between 0.31s and 14.6s. To help save time on the costly JavaScript analysis, we implemented the caching strategy described in §4.2 to reuse the analysis of frequently appearing JavaScript code. We noticed that caching reduces the server load by as much as an order of magnitude. Column 4 shows caching improved times within parenthesis. This is an important savings because it improves the server's throughput (responses the server can generate per second). Input validation was the fastest component with an average time of 10ms.

Even with our caching strategy in place, constraint extraction dominates over all other components of TamperProof. But recall that this processing happens while the user is filling out the form (as explained in §4.3). To illustrate, Figure 4 depicts the timeline for when a client requests a form, receives the form, submits the form, and receives a reply. The timeline demonstrates that the time when the user is filling out the form is relatively long, which allows constraint extraction to occur without negatively impacting the user experience. Furthermore, when constraint extraction is expensive, it often means the form's constraints are complex, making it unlikely that users will quickly submit the form.

6. DISCUSSION: AJAX

TamperProof does not currently address applications written for Web 2.0 or Web 3.0 or those that dynamically alter the client code of a web form, e.g., by employing JavaScript. However, TamperProof can be used to safeguard many of the applications that have recently been shown vulnerable to parameter tampering attacks (e.g., [3, 18, 6, 5]). Here we discuss how to enhance TamperProof to properly handle a common feature of Web 2.0 web applications: Asynchronous JavaScript and XML (AJAX) requests.

Currently, a web page utilizing AJAX is problematic for TamperProof because every AJAX request the page sends

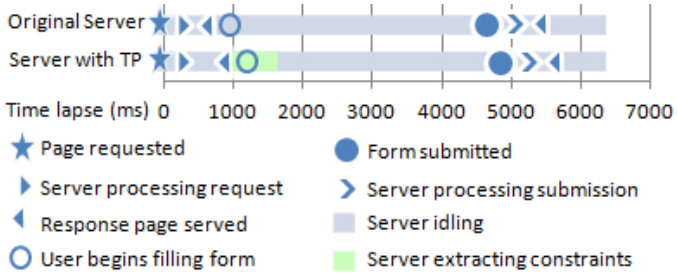


Figure 4: TamperProof: Timeline depicting actual delays experienced by end users (most expensive analysis coincides with user filling out the form).

to the server is rejected since it includes no patchID. One avenue for enhancing TamperProof to cope with AJAX would be expanding its instrumentation of outgoing HTML pages to ensure that all AJAX requests include the patchID. This fix alone does not suffice since TamperProof currently expires a patchID as soon as it receives an HTTP request that includes it. Without changing the patchID expiration policy, the first AJAX request would be properly validated and processed, but all future AJAX requests (as well as the user's ultimate submission) would be rejected as having an expired patchID.

Currently TamperProof expires patchIDs so that replay attacks are not possible. If an attacker gains access to an old patchID (e.g., in a caching proxy server), she would be able to fool TamperProof into running the same validations that were in place when the original user requested the web page, even if those validations are no longer the right ones. To properly address AJAX, TamperProof must allow multiple requests to use the same patchID. To do this while avoiding the problem of replay attacks, we could delay the expiration of a patchID to the first non-AJAX HTTP submission, thereby allowing AJAX submissions until the form itself has been submitted. This expiration policy enables AJAX clients to function properly; however, it also enlarges TamperProof's attack surface. In the non-AJAX expiration policy, the attacker can submit at most one attack for each patchID (assuming the legitimate user does not submit first), but the AJAX expiration policy allows the attacker to submit as many (AJAX) attacks as possible from the time the user requests the form to the time she submits it. Thus whereas the non-AJAX expiration policy only allows for attacks with a *single* HTTP request, the AJAX expiration policy allows for attacks comprised of *multiple* HTTP-requests.

While this enlarged attack surface is worrisome, it is important to keep in mind what a successful attack requires. First, the attacker must learn the patchID in the usually small window of time between when a legitimate user requests a form and when she submits it. Second, TamperProof still performs validation on incoming data, so the attacker must find a patchID whose validation is weaker than the validation on forms she has legitimate access to, or there is no point to the attack. Third, the attack does not fool the underlying application into acting as though the request came from the patchID's legitimate owner—it only fools TamperProof into checking a different set of constraints. Overall, allowing limited AJAX replay attacks may be a reasonable tradeoff for enabling TamperProof to properly address Web 2.0 applications.

7. RELATED WORK

Detecting and preventing malicious behavior from clients executing in untrusted environment is a topic of active research. Recently several works [5, 6, 18, 3] have reported severe parameter tampering vulnerabilities in open source applications and commercial websites. These works focus on learning the validations being performed at client and server and generate exploits by comparing them. While preventing attacks that exploit these vulnerabilities (the goal of TamperProof) is not the focus of the above work, the goals and techniques of that work is worth discussion.

The main difference between tools that detect parameter tampering vulnerabilities and tools that prevent parameter tampering attacks is that false positives (and to a lesser extent false negatives) for detection tools are less serious than for prevention tools. If a detection tool mistakenly signals that a particular input constitutes a parameter tampering attack, the developer/tester using the tool can simply ignore the false positive, but if a prevention tool mistakenly signals that a particular input is an attack and that input is rejected, a real user is prevented from legitimately using the application as it was designed.

The need for high precision results is especially problematic when analyzing web application code, which has significant technical challenges in extracting the constraints that ought to be added to the server's input validation defenses. A web application's code routinely generates code (HTML and to a lesser extent JavaScript), the results of which must be analyzed to extract the constraints imposed by the client. Moreover, the constraints imposed are sometimes dependent on the database, e.g., the client includes a drop-down list populated with credit cards the user has registered with the application (implicitly requiring the user to select one of the entries already present in the database). Finally, the web application code that runs when the user submits a web page may be completely separate from the code that generates that page, thereby requiring analysis that links the page-generation code to the page-submission code. While techniques for addressing these issues have been developed for detection tools, we were not convinced they were sufficiently robust to meet the high standards for precision required of a prevention tool.

TamperProof was designed as an online tool that analyzes each web page as it is generated. It avoids the problem of analyzing code-generating-code because it only analyzes the output. It avoids the problem of database-dependent constraints (embedded in HTML and non-AJAX JavaScript) because the database constraints have already been incorporated into the HTML and JavaScript that TamperProof analyzes. TamperProof addresses the problem of linking page-generation and page-submission code quite directly by using a patchID. Overall, this online approach is only susceptible to false positives to the extent that the HTML/-JavaScript analysis is inaccurate.

Taming Online Game Clients. A conceptually closely related line of work [10, 4] aims to curb client-side cheating in online games by constructing a model of proper client behavior against which actual clients are compared. The client codebases of online games are often separate from server and do not evolve over time as web applications do

when changes in server-side state often alters HTML and JavaScript. Analyzing web application client offline would require understanding of how the server generates the client and approaches for online game clients do not directly apply to web applications. Instead of this complicated offline analysis, TamperProof analyzes the client online (i.e., as soon as the server generates the client).

Taming Web Application Clients. Similar efforts have been made to curb malicious client behavior in web applications. Guha et al., [11] compute a model of the expected flow of requests from the client portion JavaScript) of Ajax web applications. Requests that violate the expected flow are rejected. This solution is applicable to web applications that serve static server-side content. It is vulnerable to a class of mimicry attacks that may arise due to imprecisions in static analysis. In contrast, TamperProof tackles the problem for dynamically generated server-side content. In addition, being a runtime method, it is more precise, at the cost of modest performance overheads.

Ripley. Ripley [17] executes a copy of client side code in a trusted environment to identify differences in outputs of the two clients as malicious behavior. Apart from limitations mentioned in the [17], Ripley requires substantial and careful engineering. Moreover, Ripley requires all "relevant" client events (application specific concept) be transmitted to the server. When the number and frequency of relevant events becomes too high, Ripley becomes impractical (e.g., an online game where all mouse-movement events are relevant.) Finally, Ripley makes no attempt to extract the intended validation encoded in the client which is the main goal of TamperProof. Consider a simple web form with two fields `beginDate` and `endDate` (associated JavaScript event handler `endDate` is later than `beginDate`). A malicious user can violate the constraint by entering the `endDate` before `beginDate`, as the event handler is only invoked when former is entered. While TamperProof correctly extracts and enforces the constraint, both actual client and Ripley will accept these malicious inputs.

Bypass-Shield. Bypass-Shield [14] is a direct competitor to TamperProof: a defense against parameter-tampering attacks deployed in a proxy. There are two main differences between Bypass-Shield and TamperProof: the class of applications they handle and the manner in which they check whether incoming data satisfies the appropriate constraints. Bypass-Shield employs two distinct phases of deployment: (i) an offline phase where a web crawler searches for web forms and records the constraints each one enforces and (ii) an online phase where the proxy validates incoming data against those constraints. Because all of the form constraints are extracted in an offline approach, Bypass-Shield does not properly deal with forms generated dynamically based on the backend database (e.g. in our running example, the form's drop-down list includes credit card numbers pulled from the database). In contrast, TamperProof does properly deal with dynamically generated forms because it extracts form constraints each time a form is generated.

Secondly, to validate incoming data against constraints, Bypass-Shield uses a fragment of the original JavaScript on the page; in contrast, TamperProof extracts a logical representation of the constraints the JavaScript enforces. Bypass-Shield's description in [14] leaves unanswered several basic questions. First, how is the JavaScript code for validating data identified, i.e., how do they separate validation code from non-validation code, and what happens if non-validation code is interleaved within validation code? Second, even the simplest validation code usually queries the browser's DOM to access the data it is validating; does Bypass-Shield simulate the DOM, or is the JavaScript code rewritten to take the data submitted in the HTTP request as inputs? Third, how does Bypass-Shield cope with the fact that the same data can be validated by JavaScript in multiple ways depending on the order in which the user entered that data (which can happen if the validation is performed in event handlers)? Overall, it is unclear whether this approach is closer to Ripley, attempting to perfectly replicate the data validation behavior of the client (discussed above), or to TamperProof, which attempts to extract the validation semantics of the JavaScript code and enforce exactly that.

Cross-site Request Forgery Prevention. Both XSRF prevention [13, 12] and TamperProof augment web forms and check certain properties when they are submitted. The XSRF prevention solutions typically intercept HTTP responses to augment web forms with tokens and intercept HTTP requests to validate tokens. While XSRF prevention solutions disambiguate origin of an HTTP request, TamperProof checks *semantic* aspects of an HTTP request.

Our instrumentation of forms with patchID is similar to token-based XSRF defenses. Such XSRF defenses obviously fail to prevent parameter tampering attacks, but they can protect against some sequencing attacks, depending on how the XSRF tokens are generated. If the XSRF token is the same for all forms and pages for a user's session, then it does not protect against sequencing attacks (same token can be used in submitting forms that bypass steps in desired sequence) and is therefore strictly weaker than TamperProof. But if the XSRF token is unique for each form, then it protects against those sequencing attacks that do not violate Value or Field constraints (e.g., tampering hidden fields indicating next step in the sequence). In this sense, TamperProof offers robust defense against parameter tampering and a certain class of sequencing attacks while also subsuming the protection offered by existing XSRF defenses.

Ensuring Security-by-Construction. Developing secure web applications in a ground up fashion is an exciting area of research and may eliminate vulnerabilities such as parameter tampering. The works in [8, 9, 7] aim to avoid mismatches in logic in various layers of a web application and enforce information flow confidentiality / integrity properties. Although such efforts can eliminate security vulnerabilities in new applications, they are not applicable to existing web applications. TamperProof is applicable to both new and existing web application that produce clients written in HTML and JavaScript and is independent of the language(s) and architecture used on the server.

Frameworks. To the best of our knowledge, none of the existing Web development frameworks effectively support prevention of parameter tampering in newly developed web applications. Ruby on Rails with the SimpleForm plugin [16] allows a developer to write the constraints that data should satisfy on the server, and enforces those constraints on the client. However, it only supports a handful of built-in validation routines and fails to factor in constraints encoded in HTML form fields that depend on server-side state.

Another related work is Open Web Application Security

Project (OWASP) AppSensor [1]. This project offers recommendations on avoiding parameter tampering vulnerabilities by incorporating detection points and actions in web applications. When compared to TamperProof, it does not offer automated prevention of parameter tampering attacks and like frameworks is suitable for freshly written applications.

8. CONCLUSION

We presented TamperProof, an approach to generating per-form patches for preventing parameter tampering exploits. TamperProof was evaluated on several medium to large applications demonstrating it to be both effective and efficient. As TamperProof does not require any changes or analysis of server-side source code, it can be deployed in existing proxies and defend any server-side technology or platform. By offering a robust protection that treats the server as a black box, TamperProof offers an attractive option to protect web applications from parameter tampering attacks.

9. REFERENCES

[1] Open Web Application Security Project (OWASP) AppSensor. `https://www.owasp.org/index.php/Category:OWASP_AppSensor_Project`.

[2] TamperProof demo site. `http://secwebapps.com`, 2012.

[3] Alkhalaf, M., Bultan, T., Choudhary, S. R., Fazzini, M., Orso, A., and Kruegel, C. ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies. In *ISSTA'12: Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA, 2012).

[4] Bethea, D., Cochran, R., and Reiter, M. Server-side Verification of Client Behavior in Online Games. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA, 2010).

[5] Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., and Venkatakrishnan, V. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *CCS'10: Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, IL, USA, 2010).

[6] Bisht, P., Hinrichs, T., Skrupsky, N., and Venkatakrishnan, V. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *CCS'11: Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, IL, USA, 2011).

[7] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. Secure Web Application via Automatic Partitioning. *SIGOPS Oper. Syst. Rev. 41*, 6 (2007), 31–44.

[8] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. Links: Web Programming Without Tiers. In *FMCO'06: Proceedings of the International Symposium on Formal Methods for Components and Objects* (Amsterdam, The Netherlands, 2006).

[9] Corcoran, B. J., Swamy, N., and Hicks, M. Cross-tier, Label-based Security Enforcement for Web Applications. In *SIGMOD'09: Proceedings of the ACM SIGMOD International Conference on Management of Data* (Providence, RI, USA, 2009).

[10] Giffin, J. T., Jha, S., and Miller, B. P. Detecting Manipulated Remote Call Streams. In *Security'02: Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002).

[11] Guha, A., Krishnamurthi, S., and Jim, T. Using Static Analysis for Ajax Intrusion Detection. In *WWW'09: Proceedings of the 18th International Conference on World Wide Web* (Madrid, Spain, 2009).

[12] Johns, M., and Winter, J. RequestRodeo: Client Side Protection against Session Riding. In *OWASP'06: Proceedings of the OWASP Europe 2006 Conference* (Leuven, Belgium, 2006).

[13] Jovanovic, N., Kirda, E., and Kruegel, C. Preventing Cross-site Request Forgery Attacks. In *SecureComm'06: Proceedings of the Second IEEE Conference on Security and Privacy in Communications Networks* (Baltimore, MD, USA, 2006).

[14] Mouelhi, T., Traon, Y. L., Abgrall, E., Baudry, B., and Gombault, S. Tailored shielding and bypass testing of web applications. In *ICST* (2011), IEEE Computer Society, pp. 210–219.

[15] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A Symbolic Execution Framework for JavaScript. In *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2010).

[16] Simpleform website. `http://blog.plataformatec.com.br/2010/06/simpleform-forms-made-easy/`, 2011.

[17] Vikram, K., Prateek, A., and Livshits, B. Ripley: Automatically Securing Distributed Web Applications Through Replicated Execution. In *CCS'09: Proceedings of the 16th Conference on Computer and Communications Security* (Chicago, IL, USA, 2009).

[18] Wang, R., Chen, S., Wang, X., and Qadeer, S. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *Oakland'11: Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011).