Journal of Computer Security 22 (2014) 415–465
DOI 10.3233/JCS-140498
IOS Press

Automated detection of parameter tampering opportunities and vulnerabilities in web applications

Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky and V.N. Venkatakrishnan *
Department of Computer Science, University of Illinois at Chicago, Chicago, IL, USA

Parameter tampering attacks are dangerous to a web application whose server fails to replicate the validation of user-supplied data that is performed by the client in web forms. Malicious users who circumvent the client can capitalize on the missing server validation. In this paper, we provide a formal description of parameter tampering vulnerabilities and a high level approach for their detection. We specialize this high level approach to develop complementary detection solutions in two interesting settings: blackbox (only analyze client-side code in web forms) and whitebox (also analyze server-side code that processes submitted web forms). This paper presents interesting challenges encountered in realizing the high level approach for each setting and novel technical contributions that address these challenges. We also contrast utility, difficulties and effectiveness issues in both settings and provide a quantitative comparison of results. Our experiments with real world and open source applications demonstrate that parameter tampering vulnerabilities are prolific (total 47 in 9 applications), and their exploitation can have serious consequences including unauthorized transactions, account hijacking and financial losses. We conclude this paper with a discussion on countermeasures for parameter tampering attacks and present a detailed survey of existing defenses and their suitability.

Keywords: Parameter tampering attacks, symbolic evaluation, dynamic monitoring

1. Introduction

Interactive form processing is pervasive in today's web applications. It is crucial for electronic commerce and banking sites, which rely heavily on web forms for billing and account management. Originally, typical form processing took place only on the server-side of a web application. Recently, however, with the facilities offered by the use of JavaScript on web pages, form processing is also being performed on the client-side of a web application. Processing user-supplied inputs to a web form using client-side JavaScript eliminates the latency of communicating with the server, and therefore results in a more interactive and responsive experience for the end user. Furthermore, client-side form processing reduces network traffic and server loads.

*Corresponding author: V.N. Venkatakrishnan, Department of Computer Science, 851 S. Morgan (M/C 152), Room 1120 SEO, Chicago, IL 60607-7053, USA. Tel.: +1 312 996 4860; Fax: +1 312 413 0024; E-mail: venkat@uic.edu.

The form processing performed by the browser mostly involves checking user-provided inputs for errors. For instance, an electronic commerce application accepting credit card payment requires the credit card expiry date to be valid (e.g., be a date in future and be a valid month/day combination). Once the input data has been validated, it is sent to the server as part of an HTTP request, with inputs appearing as parameters to the request.

A server accepting such a request may be vulnerable to attack if it assumes that the supplied parameters are valid (e.g., the credit card has not yet expired). This assumption is indeed enforced by the browser-side JavaScript; however, malicious users can circumvent client-side validation by disabling JavaScript, changing the code itself, or simply crafting an HTTP request by hand with any parameter values of the user's choice.

While there has been extensive work to address specific server-side input validation problems such as SQL injection and Cross-site scripting, the parameter tampering problem itself has received little attention in the research literature despite its prevalence. SWIFT [14] and Ripley [44] focus on the broader issue of ensuring data integrity in web application development frameworks. The goal of these approaches is to construct *new* web applications that are effectively immune to parameter tampering attacks. In contrast, the focus of this paper is on *vulnerability analysis*, i.e., on detecting parameter tampering vulnerabilities in *existing* web applications (or legacy applications) that are already in deployment.

In this paper, we investigate the problem of finding parameter tampering vulnerabilities in legacy web applications in two complementary and interesting settings: (a) web application's source code is unavailable (blackbox setting) and (b) source code is available (whitebox setting). Intuitively, to detect parameter tampering vulnerabilities one needs to reason about checks performed by the client-side validation in a form and if the corresponding server-side validation misses any of these checks. Computing client-side checks requires analysis of client-side code (e.g., HTML, JavaScript) whereas server-side checks can be learned by means of server-side code analysis (e.g., application codebase in PHP, JSP, ASP, etc., and database schemas in MySQL, MSSQL, etc.).

The blackbox setting would solely rely on client-side code analysis and be agnostic to server-side technologies (e.g., PHP, JSP, etc.) i.e., it does not require access to the server-side codebase. This is most appealing in cases where codebase is either not available (say for the purpose of protecting trade secrets/intellectual property) or is hard to analyze (due to proprietary languages or complex architecture). Thus this setting boasts wider applicability appeal and often yields detection solutions that can be used by remote testers e.g., as Software-as-a-Service. Finally, due to code independence, blackbox setting is often deployment friendly i.e., blackbox defenses can be seamlessly integrated in peripheral environments of web applications such as firewalls. However, in the absence of server-side source code knowledge, blackbox detection has to rely on heuristics to detect whether the server-side code validation

is missing any checks. Such heuristics are bound to have some imprecision and minimizing that is a challenge in the blackbox setting. Another challenge for this setting is to develop systematic ways to learn validation performed by client-side code in a web form.

The whitebox setting can benefit from client-side validation analysis and replace blackbox heuristics by analyzing the server-side code and precisely determine missing checks i.e., parameter tampering vulnerabilities. The knowledge of server-side code would likely enable the whitebox setting to yield more precise results when compared to the blackbox setting. Given that the server-side logic can be embedded in several layers (e.g., application code, database schema), the biggest challenge for whitebox setting is in precisely extracting validation performed by the server-side code.

Our goal is to develop systematic solutions for detecting parameter tampering vulnerabilities in the above two settings such that they can be used by testing professionals, website administrators or web application developers. In both the settings, given a web site (i.e., a deployed web application) and/or its source code, we aim to produce a report of potential vulnerabilities and the associated HTTP parameters that triggered these vulnerabilities. We envision this report being used in a variety of ways: by professional testers to develop and demonstrate concrete exploits using the inputs generated by our tool ; by web application developers checking server code and developing patches as needed; and finally, web site administrators using the report to estimate the likelihood that their site is vulnerable and alerting the concerned developers.

Summary of contributions.

- We present the first systematic approach for reasoning of parameter tampering vulnerabilities in web applications and a high level approach for detecting them.
- We specialize the high level approach to a blackbox setting (server-side code not analyzed), and propose the first systematic approach to detect parameter tampering opportunities (need manual analysis in confirming vulnerabilities) in web applications, which we call NOTAMPER.
- We also specialize the high level approach to a whitebox setting (server-side code analyzed), and propose the first systematic approach to generate parameter tampering exploits by construction, which we call WAPTEC. To the best of our knowledge this is the first approach that analyzes client, server and database code to build an in-depth and precise understanding of validations performed at the server and client, and uses it in precisely generating confirmed exploits.
- In realizing the above approaches, we make several novel technical contributions:
  - Client-side JavaScript code analysis techniques specialized to form validation code.

  - Input-generation techniques that cope with the many challenges of blackbox vulnerability analysis and novel heuristics to generate and prioritize inputs that are likely to result in vulnerabilities.
  - In-depth server-side analysis that reasons about validation performed by different server-side modules (application code and database).

- For both, blackbox and whitebox settings, we narrate our experiences in implementing them. We empirically demonstrate effectiveness of these tools by reporting several parameter tampering opportunities as well as confirmed exploits from six open source applications and three online web sites. Starting from parameter tampering opportunities, we develop concrete exploits for a majority of these applications/web sites. We also manually confirmed exploits reported by our whitebox tool. Overall, we were able to find 47 exploits in these applications. These exploits demonstrate serious security problems: unauthorized monetary transactions at a bank, unauthorized discounts added to a shopping cart, and so on. We also contrast our experiences in using whitebox and blackbox tools for detecting parameter tampering, compare their results and provide quantitative comparison of their results.
- We provide an in-depth discussion of challenges posed in defending parameter tampering attacks. We also survey the existing defenses for web application security vulnerabilities and assess their suitability in the context of parameter tampering vulnerabilities.

This paper is organized as follows. Section 2 presents interesting aspects of parameter tampering vulnerabilities through attacks on a simple web form and then presents our formulation of the parameter tampering attack vectors. Section 2 presents a high level overview of our approach as well as challenges in detecting such vulnerabilities. Section 3 provides architectures of our blackbox approach (NOTAMPER) as well as whitebox approach (WAPTEC) and describes challenges met by these approaches. Section 4 describes key components of NOTAMPER and WAPTEC. Section 5 presents our evaluation over several real world examples and web sites. Section 6 presents the related work, and Section 7 describes possible ways to fix parameter tampering vulnerabilities in legacy applications. In Section 8 we conclude.

2. Overview

Figure 1(a) depicts a typical web application that solicits user inputs through a web form. All user supplied inputs undergo client-side validation. This validation rejects invalid inputs, and otherwise submits them to server for further processing. Ideally the server must first validate the inputs again (as the client-side is an untrusted environment), before using them in sensitive operations. If the server-side validation

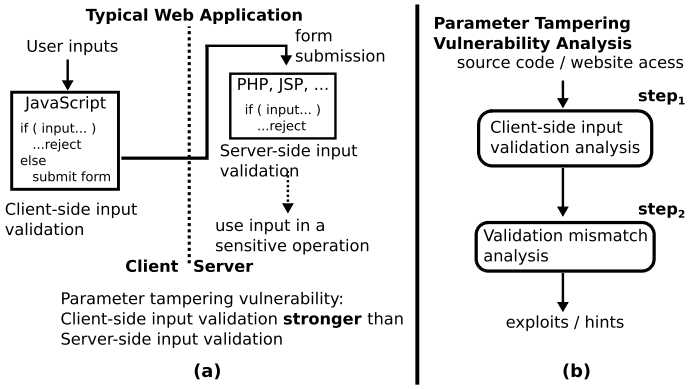
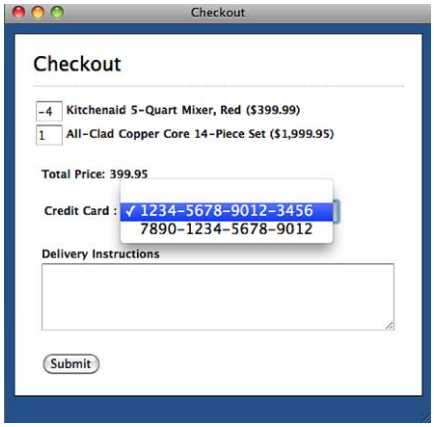


Fig. 1. High level overview: (a) parameter tampering vulnerabilities in web applications, and (b) proposed vulnerability analysis.



(a)

```
1
2  function validateForm(){
3
4    var q1 = document.getElementById("
         quantity1");
5    var q2 = document.getElementById("
         quantity2");
6
7    var n = document.getElementById("
         name");
8    var d = document.getElementById("
         directions");
9
10   if(q1 < 0 || q2 < 0 || n.length()
         > 10 || /[^a-zA-Z]/.match(d)
         ){
11     // show error, don't submit
12     return false;
13   } else {
14     // submit form
15     return true;
16   }
17 }
```

Listing 1. client.js

(b)

Fig. 2. Running example: (a) Web form of a shopping cart application, and (b) Client-side validation code in JavaScript. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-140498.)

fails to reject inputs that the client-side validation would have rejected, an attacker can bypass client-side validation and submit malicious inputs.

As a more concrete example, Fig. 2(a) illustrates the client-side of a small web application that serves as the running example throughout this paper. This example is based on real-world scenarios. Consider the checkout form of a shopping cart

application in which a user has already selected two products for purchase. The form asks the user for the quantity of each product, the credit-card to be charged (displayed in a drop-down list of previously-used cards), the name and address for shipping, any special delivery instructions, and a hidden field `op` set to "purchase". (These fields assume the usual meaning as in a typical shopping session.)

Before this data is submitted to the server, the client-side JavaScript code shown in Fig. 2(b) ensures that the quantity for each product is non-negative, that the delivery instructions include no special characters, and the name is less than 10 characters. The web browser will only submit the form to the server if these conditions are met. The server-side code shown in Listing 2 computes the cost of purchase and inserts this into the `orders` database.

Notice that the server fails to replicate some of the validations performed by the client, leading to a number of attacks, which we describe below.

Attack 1: *Negative quantities*. We discovered the following attack on the website of an online computer equipment retailer. By disabling JavaScript, a malicious user can bypass the validation check on the quantity of each product (parameters $\texttt{quantity}_1$ and $\texttt{quantity}_2$) 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 server involving debit and credit. 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 rebate on the total price. In the figure, the quantities chosen were -4 and 1 respectively, resulting in a 'discount' of \$1600.

Attack 2: *Charging another user's account*. We discovered a similar exploit at a financial institution and were able to transfer funds between arbitrary accounts. When the form is created, a drop-down list is populated with the user's credit card account numbers (parameter `card`). By submitting an account number not in this list, a malicious user can purchase products and charge someone else's account.

Attack 3: *Pattern validation bypass*. This attack enabled us to perform a Cross-site Scripting attack and escalate to admin privileges. The web form ensures that the delivery instructions (parameter `directions`) contain only uppercase and lowercase letters. In particular, special characters and punctuation are disallowed to prevent command injection attacks on the server. By circumventing these checks, a malicious user can launch attacks such as XSS or SQL injection.

Attack 4: *Submitting additional fields*. We discovered a zero-day attack on the open source application `dcpportal` that enables privilege escalation of an ordinary user to an administrator. While the web form does not even include a field called `discount`, the server code reduces the total price of the order by 15% when this field is present. This attack arises because the server code is used both to process checkout forms for regular users and checkout forms for employees.

```
1  $ca = $_POST['card'];
2  if($ca matches '1234-5678-9012-3456'|'7890-1234-5678-9012')
3      // generate HTML to show a
4      //selected card in the form
5
6  $n = $_POST['name'];
7  if( strlen($n) > 10 )
8      $n = substr ($n, 10);
9
10 if($_GET['op'] == "purchase"){
11
12  $cost = $_POST['quantity1'] * $price1 + $shipping;
13  $cost += $_POST['quantity2'] * $price2;
14
15  if(isset($_POST['discount']))
16    $cost = $cost - $_POST['discount'] * $cost / 100;
17
18  $q = "INSERT INTO orders (`name`, `address`, `card`, `cost`, `directions`)
        ";
19  $q .= " VALUES ($n, $_POST['address'], $ca, $cost, $_POST['directions'])
       ;";
20
21  mysql_query($q);
22  if(mysql_error())
23    $html .= " Please specify an address";
24 }
```

Listing 2. server.php.

2.1. Problem description

More formally, in a form submission, the client-side of a web application solicits n string inputs from the user and sends them to the server for processing. Formally, each string input is a finite sequence of characters from some alphabet Σ. We will denote an n-tuple of such inputs as I, and the set of all such I as $\mathcal{I}$.

$$\mathcal{I} = \Sigma^* \times \Sigma^* \times \cdots \times \Sigma^*.$$

Conceptually, both the client and the server perform two tasks: checking that user-supplied inputs satisfy certain constraints, and either communicating errors to the user or processing those inputs. For the problem at hand, we ignore the second task on both the client and server and focus entirely on the constraint-checking task. Formally, constraints can be formulated as a function $\mathcal{I} \to \{true, false\}$, where *false* indicates an error. We use p_{client} to denote the (explicit or implicit) constraint-checking function on the client and p_{server} to denote the constraint-checking function on the server.

Problem formulation. Our approach is based on the observation that for many typical form processing web applications there is a specific relationship between p_{server} and p_{client}: that p_{server} is more restrictive than p_{client}. Because the server often has

access to more information than the client, p_{server} sometimes rejects inputs accepted by p_{client}. For example, when registering a new user for a website, the server will guarantee that the user ID is unique, but the client will not. In contrast, if p_{server} accepts an input, then we expect p_{client} to accept it as well; otherwise, the client would be hiding server-side functionality from legitimate users. Thus, we expect that for all inputs I

$$p_{\text{server}}(I) = \mathit{true} \Rightarrow p_{\text{client}}(I) = \mathit{true}. \tag{1}$$

The server-side constraint checking is inadequate for those inputs I when the negation of this implication holds:

$$p_{\text{server}}(I) = \mathit{true} \wedge p_{\text{client}}(I) = \mathit{false}. \tag{2}$$

We call each input satisfying 2 a potential *parameter tampering attack vector*.

In practice, parameter tampering attack vectors sometimes arise because the developer simply fails to realize that the client checks should be replicated on the server. But even if the developer attempts to replicate the client checks on the server, the server and client are usually written in different languages, e.g., JavaScript for the client and PHP, ASP, or Java for the server. When there are two codebases, improvements made to one (such as additional new validation checks and maintenance updates) do not always translate to changes to the other, leading to a mismatch between the validations performed by the client and server.

2.2. Discussion

The above formulation is kept simple for the purpose of capturing the essence of parameter tampering. The basic formulation can be extended to handle a number of additional scenarios that arise in practice. For instance, web applications that engage a third-party Cashier service (such as PayPal) can be viewed as those where the (trusted) server component involves the traditional application server as well as the cashier service, both of which interact with an untrusted client. In this case, the constraint checking function p_{server} is split across the traditional web server as well as the cashier service [45].

The basic model above also does not require that the constraint checking is explicitly represented in client code. We envision p_{client} to be composed of both *explicit* and *implicit* client constraints. Explicit constraints include those that are explicitly specified in the code of the client, such as those enforced in JavaScript. The constraint checked in Line 10 of Fig. 2(b) is an example of an explicit constraint. Explicit constraints are usually application-specific. Implicit constraints are those that the web application expects to hold at the client, but not explicitly represented by the code of the application. Examples of such constraints include "read-only" constraints in certain header fields such as cookies, hidden-fields in forms, and ensuring

the absence of delimiters in URL query strings. Such implicit constraints are usually application-agnostic. Such implicit functions allow our basic approach to capture many tampering vulnerabilities such as cookie tampering and server-side HTTP parameter pollution attacks [13].

2.3. Conceptual approach

Our goal is to automatically construct parameter tampering exploits for web applications. We note that these vulnerabilities arise when the server-side input validation for a web form is weaker than the corresponding client-side validation. Constructing such exploits therefore boils down to finding such mismatches in validation. Conceptually our approach is comprised of two steps: (1) extract the validation performed by the client (Client Validation Analysis) and (2) analyze the server to find validations that ought to be performed but are not (Validation Mismatch Analysis).

Client-side validation analysis ($Step_1$). We first analyze the code of a given web form to extract the validation it performs. Specifically, we extract a logical representation of p_{client}, which we call f_{client}, by applying program analysis techniques. We then apply an SMT solver to f_{client} to effectively *generate* inputs that are accepted/rejected by the client.

Validation mismatch analysis ($Step_2$). This step is performed differently depending on whether in the blackbox or whitebox settings. The blackbox analysis must overcome the fact that the source code for the server is unavailable, whereas the whitebox analysis must carefully analyze the server's source code.

Blackbox validation mismatch analysis. Given f_{client} of a web form, the next step is to find out if the server fails to check any constraints that are checked by the client and to identify parameter tampering exploits – inputs that the client rejects and the server accepts. Finding inputs that the client rejects is straightforward: use an SMT solver to generate inputs that falsify f_{client}. However, detecting whether or not the server accepts or rejects such an input is more difficult because the only information about whether or not an input is accepted or rejected is the web pages the server sends in response to our inputs. Consequently, we generate two classes of inputs: benign and hostile. Benign inputs are those the client accepts (i.e., satisfy f_{client}), and hostile inputs are those the client rejects (i.e., falsify f_{client}). We compare the web pages the server sends in response to the benign inputs to the pages the server sends in response to the hostile inputs. The more similar a hostile page is to the benign pages, the more likely the server accepted the hostile input. This approach is inherently heuristic but as we show in the evaluation is effective in practice.

Whitebox validation mismatch analysis. The blackbox approach has an obvious drawback: our heuristic for determining whether or not a hostile input was accepted by the server may be wrong, leading to false positives and negatives. The whitebox approach is similar to the blackbox approach in that they both manipulate benign

and hostile inputs. But it differs in that it inspects the code the server executes in response to a given input to better determine whether or not the server accepted that input. Seen this way, the whitebox approach generates concrete exploits that are correct by construction.

Discussion. The crux of our approach is the assumption that all inputs rejected by the client ought to be rejected by the server; however, there are cases when this assumption fails to hold, e.g., when the server is a generic web service (such as Google maps), and the client is an application using a portion of that service (such as a map of Illinois). While this falls outside our intended scope, our basic approach can be used in such settings by replacing the automatic extraction of f_{client} from a web form with a manually constructed f_{client}. The construction of (potential) parameter tampering exploits can then continue just as described above. In other words, our approach treats f_{client}, however it is generated, as an approximate specification for the intended behavior of the server and then attempts to find inputs that fail to satisfy that specification. Our approach can therefore be viewed as combining formal verification for parameter tampering vulnerabilities with a program analysis front-end for automatically extracting a specification of intended behavior.

3. Approach

In this section, we discuss the architecture and algorithms of our approach.

Figure 3 illustrates how the two steps of our approach (Input Validation Analysis and Validation Mismatch Analysis) are realized in the blackbox and whitebox scenarios. In both scenarios, the starting point (on the left-hand side of the figure) is the Web Page Analyzer, which given a web page extracts a logical formula f_{client} representing the inputs that the page accepts. That formula is then fed into a Constraint Solver, which performs one of two tasks, depending on whether deployed in the blackbox or whitebox scenario.

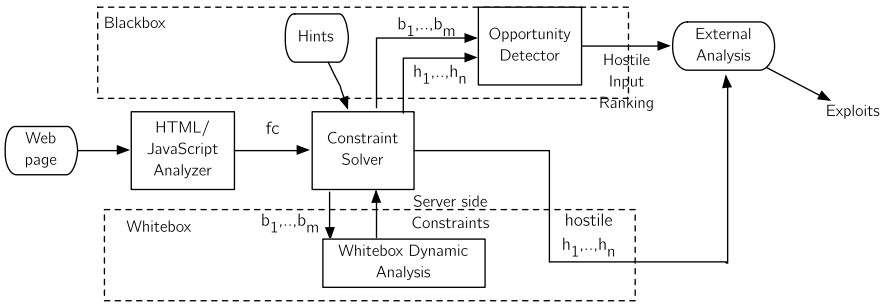


Fig. 3. Illustration of validation mismatch phase for blackbox and whitebox scenarios.

In the blackbox scenario (depicted in the top half of the figure) the Constraint Solver finds several distinct benign inputs (inputs satisfying f_{client}) and several distinct hostile inputs (inputs satisfying $\neg f_{\text{client}}$). (A user can influence which benign and hostile inputs are generated by providing hints to the solver.) In our running example, f_{client} is the following.

$$\begin{aligned}&\neg(\texttt{quantity}_1 < 0 \vee \texttt{quantity}_2 < 0)\\ &\quad \wedge (\mathit{len}(\texttt{name}) \leqslant 10)\\ &\quad \wedge (\texttt{directions} \in \texttt{[a-zA-Z]}*)\\ &\quad \wedge (\texttt{op} = \text{“}\mathit{purchase}\text{”})\\ &\quad \wedge \texttt{card} \in \texttt{(1234-5678-9012-3456 | 7890-1234-5678-9012).}\end{aligned}$$

The Constraint Solver might enumerate the benign and hostile inputs given in Table 1.

Those hostile and benign inputs are then given to the Opportunity Detector, which submits the inputs to the server and compares the resulting web pages to rank the hostile inputs by how likely it is the server accepted them, i.e., by how likely it is they are parameter tampering exploits. The Opportunity Detector then presents the results to an external tester for further analysis.

We note that because the Opportunity Detector is imperfect, the ranking it produces may be wrong; thus, we require an external tester to manually verify which hostile inputs are true exploits. That said, the blackbox analysis is still valuable to testers because it greatly prunes the space of inputs that a manual tester would need to consider, thereby making the testing process more efficient than a conventional testing approach.

The whitebox approach (depicted in the bottom half of the figure) improves on the blackbox notion of “server acceptance” by ensuring that every hostile input it claims to be accepted by the server (i) causes the server to execute a sensitive operation and (ii) takes a path through the code that some benign input also takes through the code. Such inputs are likely to be accepted by the server and therefore be parameter tampering exploits.

Table 1

Example benign and hostile inputs generated by the constraint solver

benign	$\texttt{quantity}_1$ =1, $\texttt{quantity}_2$ =1, name = “a”, directions =“”, op = “purchase” card =1234-5678-9012-3456
hostile	$\texttt{quantity}_1$ =-3, $\texttt{quantity}_2$ =1, name = “a”, directions =“”, op = “purchase” card =1234-5678-9012-3456
hostile	$\texttt{quantity}_1$ =2, $\texttt{quantity}_2$ =-2, name = “a”, directions =“”, op = “purchase” card =1234-5678-9012-3456
...	...
hostile	$\texttt{quantity}_1$ =2, $\texttt{quantity}_2$ =2, name = “a”, directions =“”, op = “purchase” card =9999-9999-9999-9999

Table 2
Constraints imposed by HTML form controls

Control	Example	Constraints
SELECT	⟨select name=**x**⟩ ⟨option value="**1**"⟩ ⟨option value="**2**" ⟩ ⟨option value="**3**" ⟩	**x** ∈ **(1|2|3)**
RADIO/ CHECKBOX	⟨input type=radio name=**x** value="**10**"⟩ ⟨input type=radio name=**x** value="**20**"⟩	**x** ∈ **(10|20)**
HIDDEN	⟨input name=**x** type=**hidden** value="20"⟩	**x = 20**
maxlength	⟨input name=**x maxlength=10** ⟩	**len(x)** ⩽ **10**
readonly	⟨input name=**x readonly** value="20"⟩	**x = 20**

To accomplish these goals, the whitebox approach first uses the Constraint Solver to find a benign input (or several benign inputs) and then uses the Dynamic Analyzer to ensure that input causes the server to execute a sensitive operation (to ensure the benign input is actually accepted by the server). In addition the Dynamic Analyzer extracts a logical formula, which we call f_{server}, representing the code path that the benign input took through the server. In our running example, f_{server} might be the formula below.

$$\big(\mathit{len}(\texttt{name}) \leqslant 10\big) \wedge (\texttt{op} = \text{“}\mathit{purchase}\text{”}) \wedge \big(\mathit{required}(\texttt{address})\big).$$

Then the Dynamic Analyzer sends f_{server} to the Constraint Solver, which computes a hostile input that takes the same path as the benign input through the server, i.e., it finds a solution to $f_{\text{server}} \wedge \neg f_{\text{client}}$. The resulting hostile input is then a correct-by-construction parameter tampering exploit.

Below, we describe this process and the components in Fig. 3 in more detail.

3.1. Phase 1: *Input validation analysis*

The first phase, which is common to both blackbox (NOTAMPER) and whitebox (WAPTEC) solutions, builds an understanding of client-side validation. Conceptually, the result of this phase is a logical formula f_{client} that represents the constraints imposed by the client on any data it allows to be submitted to the server. For web pages built using HTML and JavaScript (a common class of web pages), f_{client} is comprised of the constraints enforced by HTML and constraints enforced by JavaScript. As discussed below, extracting the constraints enforced by HTML requires a simple analysis; however, extracting the constraints enforced by JavaScript requires more sophisticated techniques.

Extracting HTML constraints. The constraints implied on form fields by HTML are easily extracted by inspecting the type of HTML input control used to display each form field. Table 2 summarizes the constraints imposed by each HTML input control through examples. In our running example, the hidden parameter `op` yields the constraint (`op` = "$purchase$"). Similarly, there is a drop-down list for the `card` control that includes two credit card values. The resulting constraint requires `card` to be assigned one of the values in that list:

$$\texttt{card} \in (\texttt{1234-5678-9012-3456} \mid \texttt{7890-1234-5678-9012}).$$

We note that HTML 5 provides richer input controls for many commonly occurring fields such as phone numbers and email addresses, that could also be a source of interesting client validation functions.

Extracting JavaScript constraints. Extracting the constraints enforced by JavaScript code requires overcoming several obstacles: (i) identifying the JavaScript code relevant to validation, (ii) addressing the fact that JavaScript code utilizes the browser's Document Object Model (DOM) to reference form field values, and (iii) actually analyzing the relevant JavaScript code.

Identifying JavaScript validation code. Identifying JavaScript validation code is difficult because many web pages utilize a plethora of JavaScript for validation, display, and even raw functionality. Validation code in particular is spread out across different event handlers, e.g., validation may occur when a form is submitted as well as when each of the form fields is changed. The fact that different code executes at different times means that we must treat the web page as a state machine that validates different code depending on the actions of the user. Each state in the machine represents the data the user has entered and flags indicating which data contains an error. As the user supplies or edits data, JavaScript code validates the data and updates the error flags accordingly, resulting in a state transition. The constraints imposed by the client on some particular data set could in theory be dependent on the path the user took through the state machine to enter that data, and hence the formula f_{client} could depend upon the structure of that state machine.

We address this challenge by analyzing the JavaScript event handlers as if they were all executed when the form was submitted. When user supplies inputs in a form, it triggers events that cause JavaScript event handlers (validation routines) to be executed. Thus we collect all the event handlers (and associated scripts) and generate a single function that invokes all those event handlers, returning `true` exactly when all the event handlers return true.

The benefit of doing so is mainly computational: it obviates the need to manually simulate events or consider the order in which events occur. But it also reflects a reasonable assumption users often make about data entry – that the order in which data was entered does not affect the validity of that data. For those cases where the order of data entry matters, our analysis may be overly restrictive, e.g., considering all event handlers may simulate the occurrence of mutually exclusive events.

DOM interactions. To complicate matters, JavaScript validation routines refer to the form fields using the browser's DOM (e.g., via `document.getElementById('quantity1')`). This is especially troublesome if the DOM is dynamically modified by JavaScript by adding/deleting additional input controls or disabling/enabling existing input controls.

We address this challenge by building a DOM simulator during analysis by recursively building a DOM from the HTML and running any initialization JavaScript code that dynamically modifies the DOM. The DOM simulator supports a small set of core methods for managing the DOM (e.g., `getElementById`). Currently, we do not support `document.write` or `document.innerHTML`, but limited support for these functions have been explored in a related project [38].

Analyzing JavaScript code. The key observation for extracting parameter validation constraints from a given JavaScript snippet is that form submission only occurs if that code returns `true`. In the simplest case, the code includes the statement `return true` or `return <boolexp>`, where `<boolexp>` is a Boolean expression. In theory, the code could return any value that JavaScript casts to `true`, but in our experience the first two cases are far more common. This observation leads to the key insight for extracting constraints: *determine all the program conditions that lead to `true` return values from all event handler functions.* To find all such program conditions, we employ a symbolic analyzer to concolically execute the JavaScript snippet computed in the previous step.

The symbolic analyzer begins by executing the validation code concretely. When a Boolean expression with symbolic variables is encountered, the execution forks: one assuming the Boolean expression is `true` and the other assuming it is `false`. Both executions replicate the existing variable values (program state) except for those affected by assuming the Boolean expression is `true` or `false`. Concrete execution then resumes. Supported DOM modification APIs act on the DOM specific to a fork.

For a given program location, the `program condition` is the set of conditions that must be satisfied for control to reach that point. If a fork returns `false`, it is stopped and discarded. If a fork returns `true`, it is stopped and the program conditions to reach that point are noted. Further, the DOM representation at this point reflects state of the HTML input controls while submitting the form including any modifications done by the JavaScript as well. The constraints checked on this fork are then computed by combining constraints of enabled controls in the DOM representation and program conditions using a conjunction ($\wedge$).

Once all forks have been stopped, f_{client} is computed by combining formulas for each path that returned `true` with disjunction ($\vee$).

For the running example one control path succeeds in returning `true`, resulting in the following formula.

$$\begin{aligned}&\neg(\texttt{quantity}_1 < 0 \vee \texttt{quantity}_2 < 0)\\ &\quad \wedge \mathit{len}(\texttt{name}) \leqslant 10\\ &\quad \wedge \texttt{directions} \in \texttt{[a-zA-Z]}*\end{aligned}$$

The above is then combined with constraints on parameters `card` and `op` mentioned before to generate f_{client}.

3.2. *Phase* 2: *Validation Mismatch Analysis – Blackbox setting* (NOTAMPER)

The Input Validation Analysis phase produces a logical formula f_{client} representing the validation being done by the client code. Given this formula, Validation Mismatch Analysis aims to find out if the server-side code fails to replicate any of the constraints enforced by the client. In the blackbox setting, the main challenge is in finding missing constraints without having any access to the server-side code.

Our tool for blackbox analysis, NOTAMPER, performs validation mismatch analysis in two steps: (a) Constraint Solver: generate benign and hostile inputs by solving f_{client} (representing inputs the client will accept) and $\neg f_{\text{client}}$ (representing inputs the client will reject), respectively, and (b) Opportunity Detector: send these inputs to the server, and identify those hostile inputs that have likely been accepted by the server-side code, thus representing parameter tampering opportunities.

We discuss the challenges encountered in realizing the above two steps and our algorithms in more detail below.

3.2.1. *Constraint solver*: *Generating benign and hostile inputs*

The Constraint Solver is conceptually simple: analyze f_{client} or $\neg f_{\text{client}}$ to find inputs that satisfy all the constraints. Typically this is done by an SMT solver. However, the Constraint Solver is not just an SMT solver because it must generate several, interestingly distinct solutions to f_{client} or $\neg f_{\text{client}}$.

Ideally, the solutions the Constraint Solver finds are orthogonal to one another. Intuitively, two inputs are orthogonal if they force the server to exercise different code paths. For hostile inputs, orthogonality is especially important because it ensures that each hostile input probes the server for a different weakness. Since if two hostile inputs take the same server code path, they are either both rejected or both accepted and hence testing the second provides no more information than the first.

The way the Constraint Solver attempts to generate orthogonal solutions to f_{client} is by asking the SMT solver to find solutions to different subsets of the constraints of f_{client}. In particular, the Constraint Solver converts f_{client} to DNF[1], and finds one solution per disjunct. Intuitively, DNF is an equivalent representation of f_{client} that identifies the semantically distinct subsets of constraints within f_{client}.

In the running example, suppose f_{client} is the formula

$$(\texttt{quantity}_1 > 0 \vee \texttt{quantity}_1 = 0) \wedge (\texttt{directions} \in \texttt{[a-zA-Z]*}).$$

Constraint Solver finds one solution for $\texttt{quantity}_1 > 0 \wedge \texttt{directions} \in \texttt{[a-zA-Z]*}$ and another for $\texttt{quantity}_1 = 0 \wedge \texttt{directions} \in \texttt{[a-zA-Z]*}$.

[1] In our experience DNF conversion was inexpensive (despite its worst-case exponential character) because of f_{client}'s structural simplicity.

If the type of $quantity_1$ is `[0-9]+` and the type of `directions` is `[a-zA-Z0-9]*`, Constraint Solver includes the constraints $\texttt{quantity}_1 \in \texttt{[0-9]+}$ and $\texttt{directions} \in \texttt{[a-zA-Z0-9]*}$.

3.2.2. Opportunity Detector: *Identifying parameter tampering opportunities*

The Constraint Solver produces a set of hostile inputs $h_1, \ldots, h_n$ and a set of benign inputs $b_1, \ldots, b_m$. The goal of the opportunity detector is to determine which hostile inputs are actually parameter tampering opportunities. The main challenge is that the Opportunity Detector must ascertain whether or not a given hostile input is accepted by the server while treating the server as a blackbox.

The Opportunity Detector addresses this challenge by ordering hostile inputs by how structurally similar their server responses are to the server responses of benign inputs. The more similar a hostile response is to the benign responses, the more likely the hostile input is a parameter tampering opportunity. In our running example, consider a hostile input where the parameter $\texttt{quantity}_1$ is assigned a negative number. If the server fails to verify that $\texttt{quantity}_1$ is greater than or equal to zero, both the hostile and benign responses will present a confirmation screen, the only difference being the number of copies and total price. On the other hand, if the server checks for a negative number of $\texttt{quantity}_1$, the hostile response will be an error page, which likely differs significantly from the confirmation screen.

As the server's responses are in HTML, we employ HTML similarity detection. There are many similarity detection algorithms for HTML responses in the literature, the most notable being algorithms for computing tree edit distance (Ref. [6]). These are especially useful in case of documents derived from a variety of sources that may contain similar content (e.g., news articles from various newspapers). In our case, since the HTML documents are produced by a single web application, it is very likely that these responses are structurally more aligned than documents from different sources, and therefore we use a home-brewed document comparison strategy based on the Ratcliff and Obsershelp algorithm [36] on approximate string matching.

Approximate matching. An important issue to be addressed in response comparison is that the contents of a HTML response will frequently include a number of variable elements that are not dependent on the server inputs, e.g., time stamps, user names, number of people logged in. A large number of such elements introduce differences in benign responses, even when the inputs are identical; therefore, we resort to an approximate matching strategy that filters out such noise from benign responses before comparing to hostile responses.

Suppose we have just two benign responses B_1 and B_2. Analyzing these responses and extracting their differences will often isolate the noisy elements in the page. These noisy elements can then be removed. For this purpose, we developed a utility that analyzes these two responses and returns the following: (1) the common sequences in B_1 and B_2, (2) content in B_1 that is not in B_2, and (3) content in B_2 that is not in B_1. Elements (2) and (3) comprise the noise, and once eliminated from B_1 and B_2 respectively, we arrive at the same HTML document C_1.

To analyze hostile response h_i, we repeat the noise elimination procedure, only this time with files B_1 and H_i. The resulting HTML, C_2, produces two possibilities, depending on whether the input h_i was accepted or not. If the input was accepted, based on our observation above, the server response H_i is likely to be similar (modulo noise) to B_1, and therefore the result C_2 is likely to be structurally the same as C_1. In case the input was rejected, the server returns a response that is likely to be structurally dissimilar, and therefore C_2 will be less similar to C_1.

The final step is the comparison between C_1 and C_2. Again, a naive comparison will not work because of the possibility that not all noise causing elements were removed during the earlier step. For example, page generation times are often embedded in the page itself, if the times were the same for B_1 and B_2, but different for H_1, then C_1 and C_2 will not be strictly structurally the same. Instead, we again use our approximate matching strategy on C_1 and C_2 as inputs. Only this time, we compute the edit distance between the two structures, resulting in a numeric value (that we call *difference rank*) for each hostile input. The higher the rank for a given hostile input, the less likely it is that the input points to a potential vulnerability.

3.3. *Phase* 2: *Validation Mismatch Analysis – Whitebox setting* (WAPTEC)

The Validation Mismatch Analysis phase in the whitebox setting eliminates the need for the Opportunity Detector and identifies parameter tampering *exploits* (as opposed to opportunities in the blackbox setting). This is done in a two-step process: (i) find benign inputs that cause the server to execute a sensitive operation (such as the `INSERT` query in line 18 of our running example) and (ii) find hostile inputs that take the same control paths as the accepted benign inputs. The hostile inputs that we find represent correct-by-construction parameter tampering exploits because (a) the server cannot distinguish them from benign inputs, (b) they successfully execute sensitive operations, and (c) they are rejected by the client.

WAPTEC, our tool embodying this approach, performs Step (i) using a form of constraint-guided search that submits benign inputs to the server and analyzes the code the server executes to find a control path that leads to a sensitive sink. When successful, WAPTEC constructs a logical formula f_{server} that represents the control path the server executed. Then in Step (ii), WAPTEC solves the logical formula $\neg f_{\text{client}} \wedge f_{\text{server}}$ to enumerate parameter tampering exploits. Conceptually, every such solution amounts to a parameter tampering exploit, but to ensure the input is in fact an exploit, WAPTEC submits it to the server and ensures it reaches a sensitive operation.

As shown in Fig. 3, the two key components for this analysis are: (a) the Constraint Solver, which finds solutions to logical formulas, and (b) the Whitebox dynamic analysis, which captures and analyzes the sequence of statements executed by the server. Since we discussed the challenges of the Constraint Solver in Section 3.2, in the rest of this section we focus on the Whitebox dynamic analysis.

We begin by describing the process of computing the formula f_{server} that represents the control path the server took for a given input. When go on to explain how WAPTEC finds benign inputs that reach a sensitive operation using f_{server}. Finally we discuss how WAPTEC utilizes the f_{server} corresponding to a benign input that reaches a sensitive sink to construct parameter tampering exploits, focusing on the soundness of the approach.

3.3.1. Computing f_{server}

For any given input, WAPTEC can observe the sequence of instructions the server executes (the *trace*) in response to that input. Representing that control path as the logical formula f_{server} is conceptually simple: f_{server} is the conjunction of all the conditions in `IF` statements in the trace.[2] In practice, however, the construction of f_{server} is more involved, a process we describe in three stages: (a) processing `IF` statements, (b) addressing the sink expression, and (c) coping with database operations.

Processing `IF` *statements*. The sequence of `IF` statements in a trace is a crucial characterization of that trace, since if two inputs satisfy all the same `IF` conditions they will take the same path through the server code. But simply conjoining all the conditions in a trace is inadequate because the conditions are often written in terms of program variables, instead of form fields. Hence, as WAPTEC analyzes the trace, it replaces program variables in the conditions of `IF` statements with the values of those variables in terms of the form field's inputs. More precisely, WAPTEC walks backwards in the trace and recursively replaces program variables with the values assigned to them until the condition is expressed in terms of inputs, concrete values and built-in functions.

For example, the `IF` statement on Line 2 (Listing 3) checks if `$main_ca` matches (`1234-5678-9012-3456 | 7890-1234-5678-9012`). We expand `$main_ca` with `$_POST['card']` because of the assignment statement on Line 1.

A challenge in precisely representing a given control path with `IF` statements is that some `IF` statements are irrelevant because they have no impact on the code path the server takes after they finish executing. A naive approach that considers all `IF` conditions as relevant would report imprecise results. For example, consider the first `IF` statement in the trace (Listing 3). This `IF` statement checks the value of parameter `card` and sets the HTML form to show the selected entry. Although the trace contains a check on `card`, it is executed regardless of the inputs and hence does not influence the code path the server takes afterwards. In particular, regardless of whether the `IF` is true or false, the query at line 20 is still vulnerable to an attacker tampering with `card`. Similarly, a form may contain several parameters but a server-side sink may only use some of them. Therefore, our analysis must factor whether a tampered parameter is actually going to be used at a sensitive operation.

Thus when computing f_{server}, WAPTEC identifies those conditionals relevant to a given sink by employing data- and control-dependency analysis: the data dependency analysis identifies conditionals that actually contributed data to a sink, and

[2] We discuss this assertion in more detail in the section on soundness.

```
1  $main_ca = $_POST['card'];                //
2  if($main_ca matches 'card-1|card-2'){  //
3
4  }
5
6  $main_n = $_POST['name'];
7  if(! strlen($main_n) > 10 ) {
8  }
9
10 if($_GET['op'] == "purchase"){
11
12     $main_cost = $_POST['quantity'] * 100 + 10;   // where $price is 100
13
14     if(!isset($_POST['discount']){
15  }
16
17     $main_q = "INSERT INTO order (`name`,`address`, `card`, `cost`)";
18     $main_q = "INSERT INTO order (`name`, `address`, `card`, `cost`)" . "
            VALUES('" .  $main_n . ", '" .  $_POST['address'] . "'" .  $main_ca
             . "," . $main_cost .  ");";
19
20     mysql_query ($main_q);
21     $_wb_status = "SUCCESS";        // query execution denoted by SUCCESS
            status
22
23 }
```

Listing 3. Trace generated for running example.

the control dependency analysis identifies conditionals that actually dictated control flow to a sink. For the running example, the query executed at line 20 is neither data nor control dependent on conditional statement at line 2 and hence this conditional is ignored while analyzing sink at line 20.

For the trace in Listing 3 the above process contributes the following constraints to the f_{server} formula:

$$\mathit{len}(\texttt{name}) \leqslant 10 \wedge \texttt{op} = \text{“}\mathit{purchase}\text{”} \wedge \neg \mathit{isset}(\texttt{discount}).$$

Sink expressions. While the sequence of `IF` statements controls which sink is executed, the sink itself rarely operates directly on the user-provided inputs. Rather, the sink operates on variations of the user's input. For example, at line 8 (Listing 2) the server-side variable $n, which is used in the sink, contains only the first 10 characters of the user-provided `name` parameter. In terms of generating hostile inputs that reach this sink, it is important to represent in f_{server} the fact that only the first 10 characters of `name` will have any effect on the sink. That is, this control path implicitly enforces the constraint $\mathit{len}(\texttt{name}) \leqslant 10$.

WAPTEC adds these implicit constraints by analyzing the sink expression (e.g., SQL query), extracting the constraints, and adding them to f_{server}. To extract the constraints, WAPTEC first rewrites the sink expression purely in terms of user inputs and concrete values (following a process similar to expansion of `IF` conditions). The

resulting SQL sink expressions are then parsed with a SQL parser thus identifying data arguments to SQL queries that contain user inputs (or a function of user inputs).

Database operations. Database operations are another example of implicit constraints on inputs. Each time the database is updated, it automatically checks if its integrity constraints are satisfied after the update, and if not rolls back that update. Thus, each time a database update is executed, there are additional constraints that are checked that do not appear in the web application's code. In our running example, without considering the database constraint (NOT NULL) on the `address` field, it is not possible to generate acceptable benign inputs. Note that this also forbids discovery of legitimately exploitable parameters for such sinks, thus resulting in false negatives e.g., the $\texttt{quantity}_1$ exploit cannot be constructed without providing a non-null `address` value. WAPTEC therefore analyzes the database's schema and integrity constraints to extract those additional constraints and then adds them to f_{server}, a process that is not as simple as it seems.

We first note that the database schema is a sequence of SQL queries that creates different tables and views and expresses certain restrictions on data that can be inserted into each column of a table. Suppose we know that a user input u is being inserted into a column c of a table, then all constraints implied on c by the database schema, must be satisfied (if validation) or will be enforced when data is added to the database (if sanitization). However, finding the mapping between u (typically server-side variables) and c (column name in a database table) is challenging as it requires bridging the namespace differences between application code and database schema i.e., application code and database tables may refer to same data with different names. WAPTEC analyzes database schema and queries issued in traces to build a mapping between server-side variables and database columns which enables it to then express constraints imposed by database in terms of user inputs.

In the first step, this analysis parses the schema of an application's database. For each table creation statement we analyze the column definitions that typically specify constraints on values that can be stored e.g., "`NOT NULL`" clause enforces non-null values whereas `enum` specifies domain of accepted values. We handle `MySQL` formatted schemas and extract such conditions in the solver language.

In the second step, we generate a symbolic query for SQL sinks found in traces and parse them. This parsing enables us to map table column names to program variables. For example, on parsing a symbolic SQL query "`insert into T (uid, ...values('$_GET[u]',...`", we can associate column `uid` of table `T` to program variable `$_GET[u]`. Once this mapping is available, we generate constraints by replacing column names with program variables in constraints generated by the first step e.g., if `uid` column had a `NOT NULL` constraint, this analysis will yield a constraint `(NOT NULL u)`.

Discussion. The above discussion highlights the relationships between server variable names, client form field names and database field names as intended by typical web applications. For the applications we analyzed, the relationships between server

variables, form fields, and database fields were fairly simple (i.e. one-to-one mappings). However, for applications where the relationships are more complex, reasoning across the three namespaces of the application will be more difficult to automate. For example, a web form might ask for an address as a single field, but the street, city, state, and zip code are all stored in different fields in the database. For applications where the namespace mappings cannot be automatically inferred, we might imagine accepting a *specification* for the mappings and in so doing broaden the applicability of our analysis.

3.3.2. Finding accepted benign inputs

Now that we have a mechanism for computing the formula f_{server} representing a control path, we discuss how WAPTEC uses such formulas to construct a benign input that is accepted by the server. While it may seem that every benign input ought to be accepted by the server, in reality, the server can reject benign inputs because the server enforces more constraints than the client (perhaps because it has more information). In our running example, the client does not require the `address` field to have a value but the server does.

Since not every solution to f_{client} will be accepted by the server, WAPTEC first finds one that is. To do this, it starts with any solution to f_{client} and checks if it causes the server to execute a sensitive operation. If so, WAPTEC is done; otherwise, it augments f_{client} with additional constraints, the intention being that any input satisfying the augmented f_{client} will take a different control path on the server, which will hopefully include a sensitive sink. In our running example, the augmentation of f_{client} will require `address` to have a non-empty value.

To compute this augmentation, the Trace Analyzer examines the execution trace of the code the server executed on the failed input, and computes a logical formula (f_{server}) representing that code trace. The intuition is that f_{server} represents (the conjunction of) the conditions on the server's inputs that if true will always lead to the same control path. Since that control path fails to lead to a sensitive sink, every input leading to a sensitive sink must falsify one of the conditions on the path, i.e., it must satisfy the negation of f_{server}. Thus, the augmentation of f_{client} when no success sink is found is $f_{\text{client}} \wedge \neg f_{\text{server}}$.

This process then repeats, starting with the augmented f_{client}, finding an input that satisfies it, and iterating until WAPTEC finds an input satisfying the augmented f_{client} that also reaches a sensitive sink. At a high level, this process generates a series of benign inputs, where each subsequent input has a better chance of being accepted than all of the previous.

Once WAPTEC finds an accepted benign input, it performs a depth-limited version of the procedure above to find additional, accepted benign inputs that take nearby control paths. To do that, the Trace Analyzer analyzes the trace of the first accepted benign input to extract f_{server}, which is a conjunction $C_1 \wedge \cdots \wedge C_n$. For each C_i, WAPTEC adds $\neg C_i$ to (the augmented) f_{client}, finds a satisfying input, and checks if that input leads to a sensitive operation. We call this process *perturbation*, since

WAPTEC attempts to perturb the constraints leading to one sensitive sink to find additional paths leading to sinks. Since each C_i can potentially produce a distinct control path leading to a sensitive sink, after this depth-limited search WAPTEC has between 1 and $n + 1$ control paths leading to sensitive operations. The perturbation process is motivated by the intuition that small changes to successful inputs may still drive execution successfully to sensitive sinks, which are often clustered together, and hence after finding a single sink, there is a high likelihood of finding additional sinks nearby.

3.3.3. Soundness

Once a benign input that is accepted by the server is found, WAPTEC computes f_{server} for the resulting control path, finds solutions to $\neg f_{\text{client}} \wedge f_{\text{server}}$, and checks that those solutions do in fact reach sensitive sinks. The pseudo-code for our approach can be found in Algorithms 1 and 2.

Algorithm 1. WAPTEC (url)

```
1:  f_client := clientAnalyzer(url)
2:  Q := {true}
3:  loop
4:      α := pop(Q)
5:      ν := solve(f_client ∧ α)
6:      (success, f_server) := server(url, ν)
7:      if success then
8:          genHostiles(url, f_client, f_server)
9:          for all C_i | f_server = C_1 ∧ ··· ∧ C_m do
10:             ν := solve(f_client ∧ α ∧ ¬C_i)
11:             (success, f_server) := server(url, ν)
12:             if success then genHostiles(url, f_client, f_server)
13:     else
14:         Q := Q ∪ {α ∧ ¬C_i | ¬f_server = ¬C_1 ∨ ··· ∨ ¬C_m}
15:         Q := simplify(Q)
16:         if empty(Q) then return
```

Algorithm 2. GENHOSTILES(url, f_{client}, f_{server})

```
1: for all δ ∈ DNF(¬f_client) do
2:     ν := solve(δ ∧ f_server)
3:     success := server(url, ν)
4:     if success then print Exploit found: ν
```

It is important to describe at a high level the mechanisms that we use for generating the client formula f_{client} and the server formula f_{server}, and their implications for the correctness of our approach.

The client formula f_{client} is generated by the Web page Analyzer using symbolic evaluation [33]. Since the formula is statically computed from the source, it is in fact an approximation. Specifically, due to the nature of the approximations made in [7], f_{client} is an *under-approximation* of the constraints the client enforces, which means that every time an input is generated that satisfies f_{client}, it is indeed the case that this input will lead to a successful form submission from the client. Similarly, $\neg f_{\text{client}}$, represents an *over-approximation* of input instances that are rejected by the client. Inputs satisfying $\neg f_{\text{client}}$ are therefore not necessarily rejected, but we can always execute those inputs in the actual client code to ensure they are rejected by the client.

In our approach, the server-side behavior is obtained by dynamic analysis of server-side code. This means that the server-side formula f_{server} will be specifically tied to each run, and is generated from the program trace induced by the run. By its very nature, dynamic analysis only considers the operations done by code that is executed; hence, f_{server} precisely captures the server behavior for the run without any approximations.

Since f_{server} is precise, and WAPTEC can verify that any solution to $\neg f_{\text{client}} \wedge f_{\text{server}}$ is actually rejected by the client, all the exploits WAPTEC reports are by our definitions concrete parameter tampering exploits.

4. Implementation

Of the many components within NOTAMPER and WAPTEC, two warrant additional discussion: the Constraint Solver and the Dynamic Analyzer. Below we describe the logical language used by the Constraint Solver and how it compares to the logical language needed to express constructs in traditional programming languages. We also describe how the Dynamic Analyzer generates traces of server code so as to observe how the server responds to a given input.

4.1. Constraint language for NOTAMPER *and* WAPTEC

The language for expressing the constraints extracted by both NOTAMPER and WAPTEC includes the standard Boolean connectives AND, OR, NOT together with the atomic constraints described in Table 3. The equality and numeric constraints are standard. The regular expression constraints require that a value assigned a variable belong to a given regular expression. A number of language-specific constraints are also necessary to encode some of the functions built into the languages we analyzed. (User-defined functions can be inlined as long as they are not recursive.) For example, the function *trim* is built into PHP and removes whitespace from the ends of a string.

Table 3
Constraint language for f_{client} and f_{server}

Class	Examples	Instances
Equality *	$=, \neq$	$x \neq y$
Numeric *	$+, *, -, /, <, >$	$x < 7$
Modal	*required*	*required*(x)
Regex *	$\in, \notin$	$x \in$ `[abc]*`
PHP/JavaScript	*trim*, *len*, *concat*	*len*(x) < *len*(*concat*(y, z))

Instead of building a custom solver to enumerate solutions to a given formula in our language, we used the state-of-the-art SMT solver Kaluza and translated our constraint language into the language Kaluza supports. Roughly, Kaluza supports the categories of constraints marked with an asterisk, plus functions for computing the length of a string and concatenating two strings. Kaluza only handles conjunction (AND), but because we always first convert formulas to DNF before solving (see Section 3), Kaluza's support for Boolean operators is sufficient. Thus, translating our constraint language to Kaluza's language mainly requires translating the modal and language-specific constraints.

The paper in which Kaluza was presented [38] describes how a number of language-specific constraints can be translated, and we refer the reader interested in additional details. Unfortunately, not all of the constraints generated by PHP/JavaScript analysis can be faithfully translated into Kaluza's constraint language. For example, PHP employs a number of built-in data structures not handled by Kaluza, and PHP functions often accept and return such data structures. For example, MyBloggie application employs the *preg_replace* function, which is a regular-expression version of a string replacement operation. *preg_replace* can both accept and return arrays as arguments. Arrays are difficult to translate to Kaluza because they correspond to an unknown number of variables, and Kaluza expects a fixed number of variables in the constraints. Another example of a function we did not translate is found in DCPPortal application: the *md*5 function computes the MD5 hash of its argument.

For constraints that cannot be translated to Kaluza's language, we simply drop those constraints, producing a constraint set that is weaker than it ought to be, potentially leading to unsoundness and incompleteness in the search for parameter tampering exploits. However, because we always checks if the variable assignment produced by the solver satisfies the original constraints, unsound results are never reported.

A lower-level but more fundamental difference between our constraint language and that of Kaluza is that Kaluza requires every variable to have a single type and does not provide functions to cast from one type to another.[3] This is problematic because the languages we investigated (PHP and JavaScript) allow variables to take on

[3]Type casting functions, while included in the documentation, were unavailable at the time of evaluation.

arbitrary values. This mismatch makes the translation difficult because a constraint such as $x \neq 0 \wedge x \neq$ "0" causes a type error in Kaluza but appears frequently in the semantics of PHP, e.g., when defining whether a variable evaluates to true or false.

Our approach approximates the semantics of PHP and JavaScript functions with a combination of type inference to detect type mismatches, type resolution to choose one type for mismatched arguments, static casting to convert problematic arguments to the chosen types, and type-based simplification to eliminate constraints that do not actually affect the satisfiability of the constraints but cause Kaluza to throw type errors.

4.2. Trace generation transformation

The key goal of trace generation transformation is to enable generation of traces that faithfully capture processing of user supplied data in a web application. To do so, the trace generation transformer performs a source-to-source transformation of applications written in PHP language. A trace is a well formed straight-line PHP program that is comprised only of assignments, calls to inbuilt functions and IF-THEN statements.

For the running example (Listing 2), Listing 3 shows the generated trace for inputs `card='card-1'`, `name='alice'`, `address='wonderland'`, `op='purchase'` and `quantity=1`. Each line in the generated trace (Listing 3) corresponds to the line in the running example (Listing 2) that generated it.

The other choice for capturing such traces is to instrument a PHP interpreter itself. Although, this approach requires less effort on a per application basis, it may require extensive changes to the PHP interpreter. Also, there are considerable analysis needs that led us to adopt a program rewriting route. First, we needed taint tracking to identify the flow of untrusted inputs. Second, we needed data and control flow analysis required to identify conditions only relevant to the sink. Third, to handle PHP5 object-oriented features, we need to unambiguously identify each object in order to avoid name collisions. While these can be done by hacking various internal parts a PHP interpreter, such changes would generally not be portable across revisions to the interpreter. Our implementation does so in a much cleaner fashion while retaining portability across various PHP interpreters and is not broken by revisions to the interpreter.

Assignments. Table 4 shows the key rules used in transformation. All text shown in bold font is recorded in traces and τ represents application of one or more transformation rules. Rule $\mathcal{R}_1$ shows the most common step in this transformation that adds two statements after each assignment statement of the original program: one to propagate taint (function `taint`) and another to generate corresponding trace (function `trace`). Certain features of PHP language such as `exit, return, break, continue` mandate computation of taint and trace information before the actual statement.

Table 4
Trace generation transformation, condition c is of the form v_1 op v_2

Original	Transformed	Rule
u = v;	u = v; u_t = taint(u,array(v_t)); trace("u = v");;	$\mathcal{R}_1$

if(c){
  $S1$;
} else {
  $S2$;
}
	if(c){
	  trace("if(c){"); $\tau(S1)$;trace("}");
	} else {
	  trace("if(!c){"); $\tau(S2)$;trace("}");
	}
		$\mathcal{R}_2$

while(c){
  S;
}
	while(c){
	  trace("if(c){"); $\tau(S)$;trace("}");
	} trace("if(!c){...force-analyze;}");
		$\mathcal{R}_3$

u = f(v);	u = f(v,v_t); u_t = f_ret_t; trace("u = f_ret_v;");	$\mathcal{R}_4$

class c {
}
	class c {
	  var id;
	  function c(){ this→id = uniq(); ...}
	}
		$\mathcal{R}_5$

Algorithm 3. TRACE(v_1 op v_2)

1: **if** v_1_t and v_2_t are false **then**
2: return;
3: v_1_t == true ? print ("*PREF*_v_1") : print (v_1)
4: print ("op");
5: v_2_t == true ? print ("*PREF*_v_2") : print (v_2)

For most parts, the function `taint` implemented standard information flow techniques to propagate taint for user inputs. However, special care was needed to initialize and propagate taint as PHP recursively defines some of the inbuilt arrays e.g., super global array GLOBALS contains itself as a member.

Algorithm 3 shows the key processing done by the function `trace`. It takes two variables v_1 and v_2 as arguments, which are connected by a PHP operator *op*. This function checks taint values of variables v_1 and v_2. If both variables are untainted, it simply returns as this operation does not manipulate user inputs (i.e., tainted values); otherwise it records variable names for tainted variables (represented by print("PREF_v_1")) and concrete values for untainted variables (represented by

`print(`v_1`)`). All PHP operators and keywords are reported verbatim in traces.

A challenge in reporting variable names in traces is caused by the possibility of *name collisions*. As traces are straight-line programs, all functions (except `PHP` in-built) executed by the web application needs to be in-lined. As this in-lining merges variables from several lexical scopes it could result in *name collisions* and could generate traces that misrepresent run of the web application e.g., name-collisions could result in traces that incorrectly capture use/reachability of an important variable. To avoid name collisions, a unique prefix is attached to each variable being reported in the trace (shown by `PREF_` in $\mathcal{R}_1$ and Algorithm 3). To compute these prefixes, we use function/method signatures and for variables appearing in classes, a per object unique identifier is used additionally (discussed later). We do not show prefix in rules other than $\mathcal{R}_1$ for ease of presentation.

Conditional statements. Rule $\mathcal{R}_2$ shows transformation of an `if-else` statement and captures evaluated condition with explicit `if` statements. In specific, for the `else` branch, negation of the `then` branch condition is captured. Similarly, for an `If` statement without the corresponding `else` clause, if the `then` branch is not taken, negation of the `if` condition is reported in the trace.

To faithfully capture conditional execution of statements in traces, the transformer enables printing of scope delimiters (opening and closing curly braces) with `if` statements. It retains a counter to track number of opened curly braces and uses this information to print right number of closing curly braces in traces. Certain `PHP` language features may terminate a variable number of lexical scopes (e.g., `return` would close all open lexical scopes in a function), and the counter is updated accordingly to ensure correct accounting of remaining scope delimiters to be reproduced in traces.

Loops. All looping constructs are first translated to `while` loops and rule $\mathcal{R}_3$ shows its transformation. The generated trace captures execution of each loop iteration with the help of an `if` statement. On termination of the loop, guard condition is always false and an empty post-loop `if` captures that. To ensure that such empty `if` statements are not ignored during analysis of traces, meta information (`force-analyze`) is added. Note that tainted loop conditions are typically of the form `c op u` where `c` is a loop counter and `u` is a user input. As the loop counter increments, for each loop iteration the corresponding trace will contain concrete value of the loop counter thus yielding a unique condition (refer to Algorithm 3).

Function calls. All user defined functions are transformed according to rule $\mathcal{R}_4$ which modifies the function signature to pass taint of each argument. Further, two global variables are introduced for each function to retain its return value and the corresponding taint (`f_ret_v` and `f_ret_t` respectively). To inline user defined functions, corresponding trace assigns return value of the function to the left hand side variable. Notice that the generated trace would contain executed function statements immediately before this assignment.

Calls to inbuilt functions are retained in traces. Invocation of sink functions (sql query execution and file I/O) require additional processing to report their status (success/failure) in traces. This requires sink specific computation e.g., for `MySQL` sinks this amounts to checking the return value of `mysql_errno()` as well as analyzing return value of a query `mysql_query("show warning")`.

Classes. Object-oriented features are often used in `PHP` programs (2 of the 6 applications we evaluated were object-oriented and used inheritance). As multiple instantiations of a class yield objects with same methods, method signatures are same for all such objects. Thus appending signatures to variable names may still lead to name collisions in object-oriented programs. Further, a member variable can be accessed using multiple namespaces e.g., by using the *this* operator (inside methods) or by using names assigned to objects. Although, all such instances are accessing the same memory region, a naive renaming scheme may lose precision by failing to identify these accesses with a single variable name.

The main changes required to classes are for computing unique prefixes for variables and is shown in rule $\mathcal{R}_5$. Here transformer adds a member variable *id* to hold the unique identifier for each instance of the class. The constructor methods are augmented to initialize the *id* variable to a unique value. Further, inheritance is inherently handled in this scheme as the *id* member of inheriting class shadows the *id* member of base class. With the help of *id* variable, accesses to a member variable through an object ($\$o \rightarrow member_1$) or `this` operator ($\$this \rightarrow member_1$) are uniformly transformed as $v\_\$id\_member_1$. This enables subsequent analysis to correctly identify accesses to a single memory location from disparate namespaces.

5. Evaluation

We implemented both the blackbox and whitebox approaches as prototype tools (NOTAMPER and WAPTEC, respectively) to enable testing on real world applications. We selected six medium to moderately large opensource PHP applications (that were used as benchmarks in other papers [43,46]) and three live websites for our test suite. Table 5 provides background information on these applications (lines of code, number of files, and functionality). The test suite was deployed on a Mac Mini (1.83 GHz Intel, 2.0 GB RAM) running the MAMP application suite, and the prototype was deployed on an Ubuntu workstation (2.45 GHz Quad Intel, 2.0 GB RAM).

5.1. Summary

Experiments. We first evaluated both approaches by running the prototype in blackbox mode (NOTAMPER) and then again in whitebox mode (WAPTEC). For the open source applications, we were able to run both approaches, but we could run only NOTAMPER on the websites due to their source code not being openly available. Then we used the results of both evaluations to compare and contrast whitebox and blackbox analysis in the context of parameter tampering attacks.

Table 5
Summary of results

Application	Size (KLOC)	Files	Use	Exploits
SnipeGallery	9.1k	54	Image Mgmt	2
SPHPBlog	26.5k	113	Blog	1
DcpPortal	144.7k	484	Content Mgmt	32
PHPNews	6.4k	21	News Mgmt	1
Landshop	15.4k	158	Real Estate	3
MyBloggie	9.4k	59	Blog	6
www.wiley.com	Closed source		Library	0
www.selfreliance.com	Closed source		Banking	1
www.codemicro.com	Closed source		Shopping	1

Results summary. The outcome of our experiments is summarized in Table 5. We evaluated one form in each application. Our prototypes found a total of 47 exploits that were also manually verified. For each application shown in column 1, the last column shows reported exploits. As shown in this table, WAPTEC successfully generated one or more exploits for each open source application in the test suite, and NOTAMPER generated exploits for 2 of the 3 closed source applications. We highlight several interesting exploits below.

5.2. Exploit details

Unauthorized money transfers. The online banking website www.selfreliance.com allows customers to transfer money between their accounts online. A customer logs onto the web site, specifies the amount of money to transfer, uses a drop-down list to choose the source account for the transfer, and uses another drop-down list to choose the destination account. Both drop-down lists include all of the user's account numbers. It turns out that the server for this application did not validate that the account numbers provided were selected from the drop-down lists. Thus, sending the server a request to transfer money between two arbitrary accounts succeeded, even if the user logged into the system was an owner of neither account. When NOTAMPER analyzed this form, it generated a hostile input where one of the account numbers was a single zero. The server response was virtually the same as the response to the benign inputs (where the account numbers were selected from the drop-down lists). Therefore, this input was ranked highly by NOTAMPER as a potential vulnerability. When we attempted to confirm the vulnerability, we were able to transfer $1 between two accounts of unrelated individuals. (Note that if the server had checked for valid account numbers but failed to ensure the user owned the chosen accounts, NOTAMPER would not have discovered the problem; however, if the human tester provided valid account numbers as hints, NOTAMPER would have identified the problem.) We note that this vulnerability could have significant impact given that the bank in question

has over 30,000 customers. Further, a successful exploit requires only the knowledge of victim account numbers, which are shared routinely when writing checks. The bank was contacted about this vulnerability and fixed it in less than 24 hours, during which time the functionality for transferring money was disabled completely. Furthermore, Selfreliance had licensed the software that contained the vulnerability from ESP Solutions (`www.espsolution.net`), who applied a global patch for all their clients that utilized this functionality and additionally fixed similar problems in their other key product FORZA that provides online banking features.

Unlimited shopping rebates. The online shopping website www.codemicro.com sells computer equipment, e.g., hard drives, printers, network switches. The form in question shows the contents of the shopping cart and allows a user to modify the quantities of the selected products. The quantity fields employ JavaScript to restrict shoppers to enter only positive numeric values. When NOTAMPER analyzed this form, it circumvented JavaScript and supplied a negative number for one of the quantity fields. The resulting HTML page was identical to the pages produced by benign inputs, except for the quantities and total price; thus NOTAMPER ranked it highly as a potential parameter tampering exploit. We were able to further develop this into another serious exploit. After adding two items to the cart and disabling JavaScript in the browser, we set the quantity of one item to a negative number. Then when we re-enabled the JavaScript, the total purchase price was computed by multiplying the quantity of each product by its price. Thus, the negative quantities enabled unlimited rebates for any purchase. Furthermore, these negative quantities were successfully accepted by the server, thus permitting the user to purchase at the reduced price. The potential of exploiting this vulnerability could have been significant as the website contains a very large inventory of computer equipment. The site administrators confirmed the vulnerability and fixed it within 24 h.

Privilege escalation. The DcpPortal application allows guests to register for an account. The registration form solicits standard information, such as name, e-mail, username, password, etc. Upon normal registration, a user is provided with an account having basic privileges. When the form is submitted, the server-side form processing code validates the provided information and checks if a cookie make_install_prn is set. When this cookie is set to 1, the user is registered with administrative privileges. By setting this cookie, it is possible for an attacker to register an account with escalated privileges. Discovery of the above vulnerability required WAPTEC to construct a negative parameter tampering exploit, i.e., the client-side formula f_{client} for this form did not contain any restriction on the parameter make_install_prn; however, the server-side formula f_{server} checked its value. After analyzing the server-side code, WAPTEC discovered this additional parameter and set it to true, which resulted in the escalation of privileges of the user being registered to an administrator. After confirming the exploit, we analyzed the application to understand the root cause of this flaw. We found that the application used the cookie

make_install_prn during initial installation to allow creation of an administrator account. To patch this vulnerability, the application can use additional server-side state (e.g., sessions) to avoid depending on the cookie value alone or have a separate form for this purpose.

Duplicate users. The DcpPortal application requires unique usernames comprising of at most 32 alphanumeric characters for new account registrations. The client-side allows only 32 alphanumeric characters, while the server enforces uniqueness by checking that the database does not contain a matching username before creating an account. Further, during insertion of new user details, the database enforces the length by truncating usernames to 32 characters. During vulnerability analysis, WAPTEC recognized that the server fails to enforce the length constraint before checking for existing usernames. For this vulnerability, WAPTEC generated hostile inputs that exceeded 32 characters, which because all existing user names in the database are 32 characters caused the username existence check to always return false. In addition, the server also fails to replicate the alphanumeric constraint on username and WAPTEC generated a hostile input that contained invalid characters. When confirming these exploits, we were able to refine them. Although true account duplication works only for long usernames, it is possible to create imposter accounts by appending URL encoded whitespace to existing usernames.

Blog category hijacking. MyBloggie, a blogging application, allows registered users to submit posts to the blog. When submitting a post, users are asked to choose a category from a drop-down list of existing categories. By submitting a value not in that list, an attacker can submit posts that will appear in a category that will be created in the future. This may negatively impact effectiveness/quality of the future category; thus, this attack can hijack a future blog category. WAPTEC discovered the missing validation exploited it by supplying an out of range value.

Additional exploits. Below we briefly describe one exploit from each of the other four applications we evaluated.

- *PHPnews*, a news management application, allows administrators to modify certain files through a form that includes name of the file as a hidden field. The server-side code fails to validate that the file name is not tampered and as a result attackers can update existing files, create arbitrary files and/or corrupt files of other applications deployed on the same web server.
- *SnipeGallery*, a photo album application, allows users to arrange albums hierarchically by selecting a parent category for each new album from a drop down list. By selecting a value not in that list, the new album becomes invisible; furthermore, additional analysis shows that a carefully constructed parent album value leads to a SQL injection attack.
- *Landshop*, a real estate application, includes a form with a hidden field not pertinent to that form. When the value of this field is set to the ID of an existing listing (which are displayed prominently on the site), that listing is deleted from the application whether the user is the owner or not.

- *SPHPBlog*, a blogging application, allows users to choose a language for the blog from a drop down menu. By selecting a language not in the list, an attacker can make the application unusable and thus conduct a denial-of-service attack.

The severity of the generated exploits underscore a widespread lack of sufficient replication of the client-side validation in the corresponding server-side code.

5.3. Comparison of blackbox and whitebox results

We evaluated the blackbox and whitebox approaches individually to compare and contrast them. Specifically, we were interested in comparing the number of exploits, false positives and false negatives. The results of the comparison are summarized in Table 6. For each application, this table reports the number of confirmed exploits found by NOTAMPER (column 2) and WAPTEC (column 3). The next two columns report false positives and false negatives for NOTAMPER when compared to WAPTEC. In total, the blackbox approach resulted in 23 false positives, and 24 fewer confirmed exploits when compared to the whitebox approach. Further, for the DcpPortal and Mybloggie applications, WAPTEC found several exploitable sinks. For example, for DcpPortal column 3 shows 16 (32): each hostile input generated by negating 16 f_{client} disjuncts was used in 2 distinct sinks and hence were exploitable (total 32 exploits). We wish to note that all these disjuncts would have contributed to one hostile each, at best, in NOTAMPER. In the rest of this section we describe some of the qualitative benefits of using WAPTEC when compared to NOTAMPER, using the exploits described in Section 5.2 as examples.

Multiple sink analysis. A single form input can be used by the server at multiple sensitive operations and can potentially cause problems at each such operation. WAPTEC has the ability to detect multiple vulnerabilities along a single path reached by an input, whereas NOTAMPER is incapable of reasoning about multiple sinks due to its blackbox nature. The duplicate user exploit in DcpPortal demonstrates a case where a single hostile input exploited multiple sinks. When WAPTEC negated

Table 6
Comparing whitebox and blackbox analysis results

Application	Confirmed exploits		False positives	False negatives
	Blackbox	Whitebox	Blackbox	Blackbox
SnipeGallery	2	2	1	0
SPHPBlog	1	1	0	0
DcpPortal	13	16 (32)	9	19
PHPNews	1	1	0	0
Landshop	3	3	1	0
Mybloggie	1	5 (6)	12	5
Total	21	45	23	24

the 32 alphanumeric character length constraint, it produced an invalid string that was used at two sinks. The string was first used in a sink that checked if a duplicate username exists in the database, and later it was inserted into the database at a second sink. WAPTEC detected that the malformed username was used at both sinks and reported an exploit for each. On the contrary, NOTAMPER reported a single vulnerability for a similar hostile input. This is because NOTAMPER is incapable of reasoning about multiple sinks and, therefore, suffers from false negatives.

Negative tampering. Negative tampering attacks are those where the user supplies data for form fields not present in the actual form. The server is vulnerable to such attacks when it checks the value of such fields, a reasonable behavior if the server code is used to process multiple forms. WAPTEC uncovered a negative tampering vulnerability on the DcpPortal registration form. It found a conditional that depends on the value of a parameter make_install_prn, which is not found in the client-side formula. To explore this branch, it satisfied the conditional by setting make_install_prn to 1. By analyzing data and control dependencies, it then determined that this branch modifies parameter values used in the sink, and therefore, reported the exploit. The result of this exploit allows a malicious user to obtain a new account with escalated privileges. NOTAMPER is inadequate to discover such exploits because that requires analysis of server-side form processing logic to uncover hidden functionality, which is out of scope for a blackbox tool.

Sanitization. Once a web server has been given inputs, it will sometimes sanitize those inputs to make potentially unsafe inputs safe to operate on. Since NOTAMPER treats the server as a blackbox, it cannot determine whether or not the server sanitizes inputs and so even when the web page returned by the server for a hostile input indicates error-free processing, the hostile input may not be an actual exploit. The server could have transformed the input into safe equivalent before using it. This sanitization can occur either in the web application code or in the database itself. Because WAPTEC can analyze the source code, it can either identify sanitization or avoid it. As mentioned in Section 3.3, WAPTEC avoids sanitization in the web application code by only finding hostile inputs that take the same code path as some benign input. For database sanitization, WAPTEC analyzes the database schema and checks for warnings on database operations that signal sanitization. Not accounting for sanitization led NOTAMPER to yield false positives in two different applications. In DcpPortal, NOTAMPER produced a hostile input by falsifying a range constraint on a field for the user's birthdate. The server quietly modified any value not in the required range to 0000-00-00. Thus, while NOTAMPER saw only an error-free web page and reported a potential exploit, WAPTEC observed a database warning when saving the date, and thus correctly avoided a false positive. Similarly, in SnipeGallery, the database automatically enforced a length constraint on a form field by truncating extra values; hence, NOTAMPER reported a false positive having submitted an overly long value and having received no errors, but WAPTEC discovered the database's truncation of that value. In general, it is possible to deal with sanitization functions correctly in

the whitebox approach, if the code uses well-known sanitization functions. (More on this in Section 5.4.)

Incomplete client specification. Another source of problems for the blackbox case is attributed to insufficient client-side information for generating a benign input that is actually accepted by the server. In the blackbox case, it is critical that NOTAMPER's benign inputs are actually accepted by the server, since otherwise it is likely that many of the hostile input server responses will look similar to the benign input responses even if they were actually rejected by the server. A simple example is when the client enforces no constraints, but the server requires that some of the fields have values. Unfortunately, there is little opportunity in the blackbox case to enhance the information from the client about how the server ought to perform, which is why analyzing the server's code in the whitebox case is so valuable. We can take information we learn about the server and add it to the information we know about the client. For example, NOTAMPER failed to catch the category hijacking exploit in the MyBloggie application because it did not know what all of the 'required' variables were (i.e., the set of variables that are required for server-side processing). In this example, the server-side code required the client to set the value of either the submit or the preview field. NOTAMPER happened to set neither field, and hence regardless the values for other variables, the server returned an error, making the responses for benign and hostile inputs all look similar. To alleviate these problems, testers can improve the automatic benign input generation step by supplying hints to NOTAMPER and still preserve the automatic hostile input generation and analysis.

Summary. WAPTEC demonstrated that a whitebox approach produces improved results over the blackbox approach used by NOTAMPER. WAPTEC uncovered a greater number of exploits and eliminated false positives and false negatives by precisely reasoning about form inputs across the entire application (client and server). In contrast, NOTAMPER is limited to using constraints implied by the client-side code and employs heuristics to determine if the server-side code accepted/rejected inputs and thus inherently suffers from false positives and false negatives.

Although WAPTEC results are consistently better than NOTAMPER, both of these approaches have their own utility. As NOTAMPER does not rely on analyzing server-side code, it can be employed to analyze a wider range of applications and websites. However if the source code is available, a whitebox analysis like WAPTEC can be employed to perform deeper code analysis to pinpoint more security problems more accurately. This can greatly reduce the human effort required to confirm exploits.

5.4. Soundness

Even though the whitebox approach consistently yields better results compared to the blackbox approach, the whitebox approach is imperfect. WAPTEC may yield false positives if its computation of f_{server} is imprecise due to loop approximations

Table 7
Impact of loops and sanitization on results

Application	# of loops	Loop impact	# of sanitizers	Sanitization impact
SnipeGallery	0	none	4	none
SPHPBlog	17	none	22	none
DcpPortal	18	none	86	none
PHPNews	2	none	3	none
Landshop	15	none	34	none
Mybloggie	10	none	13	1 potential fp

and unknown sanitizing functions. Recall that WAPTEC computes f_{server} by analyzing the trace of instructions executed for a fixed set of inputs and approximates the semantics of loops. Further, if inputs are processed enroute to sensitive operations, WAPTEC does not always know whether inputs are sanitized before they reach a sensitive operation (a fundamental limitation of many whitebox approaches as they fail to distinguish sanitization functions from those functions that may simply process inputs for non-security reasons). To understand the pervasiveness of these potential sources of unsoundness, we examined all instances of loops and sanitization for each of our application's (relevant) traces. The results of our findings are summarized in Table 7. For each application, column 2 shows the number of loops encountered in traces and column 3 shows the impact loop approximations had on results. Column 4 shows the number of sanitization functions appearing throughout each trace, and the last column shows the impact that those sanitizers had on results.

Loops. As discussed in Section 3.3, WAPTEC extracts server-side constraints from straight line traces which include unrolled loop bodies. As a consequence, the constraints that are extracted from traces might be approximate representations of the true f_{server}. This is true when the following conditions are met for any loop: (i) the number of loop iterations is influenced by form inputs, and (ii) the loop body contains constraints on form inputs. When these conditions are met, there is a potential for false positives. For example, a loop that iterates the length of an input to filter characters, will result in a constraint that checks only as many characters as contained in the benign input. In that case, a subsequent hostile input containing illegal characters past that length would be falsely reported as an exploit by WAPTEC. One way to mitigate the occurrence of false positives due to loop approximations is to conduct a preliminary test to detect loops that satisfy the 2 conditions, and then either discard them during formula extraction or ask testers to supply loop invariants for them.

In our analysis of server traces, we found no cases where loop approximations impacted soundness of our results. From the 62 unique loops we encountered, their breakdown is as follows. There were 15 that iterated over the result set of a SQL query not involving inputs, 1 that iterated over the result set of a SQL query involving inputs, 7 that iterated over data from the server's environment, 3 that iterated over

a static range, 30 that iterated over a program variable unrelated to any inputs, and 6 that iterated directly over inputs. None of the loops whose iterations were influenced by inputs imposed constraints on inputs, thus we found no false positives due to loops. Although there is a potential hazard for false positives due to loop approximations, the applications from our test suite suggest that the occurrence of loops that are both input dependent and input constraining is uncommon. It is straightforward to alert the user of our tool to alert the user in these special circumstances, so that the user can carry out further analysis on any impact that the presence of loops may have on the results of the analysis.

Sanitization. Another potential source of false positives is server-side sanitization functions. Note that in our analysis, f_{server} includes the actions of sanitization functions on the paths traversed by the benign inputs (cf. Algorithm 1). However, when WAPTEC explores the other parts of the server codebase, and when it handles a hostile input by correcting it with the help of a filter function, the f_{server} may not properly reflect the validation that is performed by the server. Consequently, a false positive could arise when a hostile input gets sanitized in the basic block that contains a sink expression. A necessary condition for a false positive is for the client to constrain an input and for the server to sanitize that same input, such that a hostile input is altered into a benign value. Since sanitization functions are troublesome only in these special circumstances, we set out to determine whether any false positives caused by sanitization were present in our results.

Among all the tested applications, there were 162 unique functions that resembled sanitization, e.g., preg_replace, trim, stripslashes. Out of this total, there were 0 instances where sanitization had an adverse impact on our results. In the majority of cases (127 occurrences), sanitization was performed on program variables that were irrelevant to form inputs, e.g., during HTML code generation by template engines. In all but one of the remaining 34 cases, f_{client} did not contain a constraint corresponding to the input being sanitized on the sever-side, therefore, a hostile input was not generated and no false positives were encountered.

5.5. Additional details

For each evaluated application, Table 8 captures the complexity of generated formulas (column 2 – client-side constraints, column 3 – server-side constraints, column

Table 8
Additional results

Application	Formula complexity			Avg. trace size (KB)	Time (s)
SnipeGallery	11	5	11	5	41
SPHPBlog	37	1	1	1	4
DcpPortal	187	2	48	135	10,042
PHPNews	1	1	1	1	12
Landshop	20	2	8	20	60
MyBloggie	37	5	4	738	2082

4 – database constraints), average size of generated traces (column 5 – kilobytes) and average time taken to run the tool (column 6 – seconds).

Outliers. The most notable application we tested, DcpPortal, included the largest formula complexities, the largest number of exploits, and the longest running time. The larger the formula complexity, the larger and more complex the form; hence, a longer running time is to be expected. The large number of exploits is partially attributed to large formula complexity because the potential number of exploit generation attempts is larger; however, the presence of a large number of confirmed exploits points to poor server-side validation of inputs.

Manual intervention. In a preliminary analysis of the chosen applications, we selected forms that contained interesting client-side specifications and collected login credentials necessary to access them (in 5 applications). We also extracted form action parameters in cases where applications reused processing code between multiple forms (total of 4). These hints were necessary to facilitate automatic analysis and to restrict exploration of server-side code pertaining to other forms. Overall, it required typically less than 5 minutes to collect this data for each form.

6. Related work

There has been much work on the topic of detecting cheating in a client-server paradigm. These works can be broadly classified into two categories: (1) those that *detect* such vulnerabilities (i.e., an offline system vulnerability analysis) and (2) those that *prevent* (or eliminate) cheating by monitoring the system (or fixing the root causes of vulnerabilities). The latter will be discussed in Section 7, and we focus on the former category in the rest of this section.

Historical context. Parameter tampering attacks have existed since the early days of the Web an e-commerce. We are aware of at least one SANS security bulletin [37] from 2000 talked about the prevalence of "price modification" vulnerabilities in online security vendors. Fu [20] highlighted the issue of tampering in the context of online authentication. Since then, multiple factors have led to the rise of these vulnerabilities: growth of JavaScript and AJAX technologies, increased application complexity, and the increase in attack surface. This has motivated a detailed investigation of these problems that was presented in this paper.

Multi-tier web application analysis. Web applications, those following the LAMP model (Linux, Apache, MySQL, PHP) in specific, are inherently multi-tiered and feature different components that are written in different programming languages. Specifically, a LAMP application features client-side code written in HTML/ JavaScript, server-side code written in PHP and database schema expressed in MySQL. To precisely construct parameter tampering exploits, WAPTEC reasons across these tiers and expresses them uniformly in the language of the solver. To the best of our knowledge, WAPTEC is the first work that offers a systematic multi-tiered analysis for legacy web applications. Most existing works on web application

analysis do not reason across all tiers. Balzarotti et al. [3] offer a system that tries to reason across modules of a web application to find data and work flow attacks on web applications and in doing so offer limited support for finding URLs embedded in JavaScript and HTML code. Programming languages such as Links [15,16] and frameworks such as [14,24] offer principled construction of multi-tiered applications but do not assist analysis of legacy web applications. In contrast, WAPTEC offers a more powerful analysis framework that combines concolic analysis of the HTML/JavaScript with static analysis of runtime traces for legacy web applications.

Specification inference. AutoISES [42] is an approach for C program bug detection that mines for common security-related patterns and identifies deviations from these as vulnerabilities. Engler [18] detects security bugs in C programs by mining temporal safety patterns and checking for inconsistencies. Srivastava [40] et al. exploit the difference between multiple implementations of the same application programming interface to detect security violations. Felmetsger et al. [19] develop a system that monitors normal execution of a web application to infer a set of behavioral specification. This specification is then used to find paths in program that will likely violate these specifications and hence may indicate missing checks. In contrast to these approaches, in our problem context, we are analyzing the two distinctive code bases of a single web application and have developed techniques to check consistencies between these two code bases.

Test input generation. A rich literature exists on automating the task of test input generation [17,22,23,26,32,38,39]. Saxena et al. [38] combines the use of random test generation and symbolic execution for testing JavaScript applications with a goal to find code injection vulnerabilities in the client-side code that result from untrusted data provided as arguments to sensitive operations. Halfond et al. [26] employ symbolic execution and constraint solving to infer web application interfaces for improved testing and analysis of web applications. Kieżun et al. [32] use symbolic execution and a library of attack strings to find code injection attacks in web applications. Sen et al. [39] propose a technique that combines concrete and symbolic execution to avoid redundant test cases as well as false warnings. Authors of [22,23] propose techniques to record an actual run of the program under test on either a well-formed input [23] or random inputs [22], symbolically evaluate the recorded trace, and gather constraints on inputs capturing how the program uses these. The collected constraints are then negated one by one and solved with a constraint solver, producing new inputs that exercise different control paths in the program. Although NOTAMPER and WAPTEC aim to find hostile inputs and in that sense are similar to these approaches, our formulation of the parameter tampering problem as one checking the consistency of the server and the client code bases and development of web application-specific methods such as perturbation that are specialized to this problem make them distinctive.

Emmi et al. [17] concolically execute server-side code and analyze executed SQL queries to find missing database records to improve branch coverage in testing. WAPTEC tests legacy applications that typically contain relevant records in databases

and extracts database constraints to improve precision of results. A key technical difference is that Emmi et al. decode WHERE clauses to reason about "missing records" in the current database and do not elaborate satisfying "database metadata" (typically database table schema) to generate such inputs. WAPTEC's database handling criteria is based on such schema analysis. In particular, it relies on the insight that database schema encodes constraints that must be satisfied by acceptable hostile and benign inputs.

Input validation. The lack of sufficient input validation is a major source of security vulnerabilities in web applications, including the type of vulnerabilities reported in this paper. As a result, there is a fairly well developed body of literature in server-side techniques that attempt to curb the impact of untrusted data. Attacks such as SQL injection and Cross-site Scripting are well studied examples (e.g., [41] and many others) in which untrusted data can result in unauthorized actions in a web application. Both WAPTEC and NOTAMPER are similar to such studies in the sense that they can find vulnerabilities that could be exploited by SQL injection or Cross-site Scripting attacks. However, our approach uses client-side code as a specification of the expected server-side behavior and hence is able to also find logic vulnerabilities that do not necessarily require code injection. Recent work has focused on the similar issue of automatically discovering server-side parameter pollution [1] as well as finding vulnerabilities in cashier-as-a-service stores [45]. As discussed earlier, these types of vulnerabilities are captured by our basic model of parameter tampering, although additional improvements to our implementation are required in order to identify these vulnerabilities.

Sanitization. Sanitization of inputs is an effective layer of defense for attacks that ride user inputs. Typically sanitization aims to re-write hostile inputs to render them benign. Unfortunately, there is no standard technique to sanitize user inputs, which often results in vulnerable applications. Saner [2] attempts to identify and validate adequacy of sanitization routines in web applications. It models sanitization performed by the web application as an automata and detects inadequacy by finding nonempty intersections, which characterize successful attacks. Recently, BEK [29] proposes a language for writing sanitizers that enables systematic reasoning about their correctness. To select a server-side control path to analyze, WAPTEC generates inputs that satisfy the client-side validation. In general, this leads to selection of paths in the server-side code that do not sanitize user inputs. For cases where sanitization is performed on all control paths, WAPTEC offers a limited reasoning of sanitization. In summary, all of the above research works provide the much needed starting points for sound reasoning about sanitization in web applications, an important area that needs further research.

7. Countermeasures

As shown in our evaluation, legacy (existing) code in both open source and commercial world is vulnerable to parameter attacks. Fixing these vulnerabilities in

legacy code is both important and challenging. (We discuss development methodologies for new applications that reduce this attack's surface in the related work Section 6.) Given the large number of legacy applications, techniques that involve manual programmer effort to find/fix these problems is expensive. Hence an automated means to patch applications is desirable.

The goal of this section is to present an analysis of the problem space of automated countermeasures, in order to identify the requirements for a good solution in the whitebox and blackbox settings. We do so by outlining the challenges (Section 7.1) to any automated approach that aims to prevent exploitation of parameter tampering vulnerabilities. We then discuss the requirements for any whitebox solution that analyze server-side source code (Section 7.2) as well as by blackbox solutions that do not require access to server source code (Section 7.3).

7.1. Challenges for automated patch generation

In our running example the client-side JavaScript code requires the end user to specify a positive value for the parameter `quantity`, thus encoding intent of the application with respect to this parameter (Listing 1). However, the corresponding server-side code does not check value of the parameter `quantity` and fails to enforce that intent (Listing 2). This lapse can then be exploited by supplying a negative value for the parameter `quantity`.

The goal for automated patch generation is to locate and fix such missing enforcement of these checks. In our running example, the tampering vulnerability involved the use of parameter `quantity`. In order to patch the application of this vulnerability, such inputs need to be forbidden. To generate such patches automatically, several questions must to be answered:

- *Identifying patch constraints.* (Section 7.1.1) What constitutes a patch? Specifically, what are the constraints on inputs that need to be forbidden? For the running example, the single check `quantity` $<$ `0` constitutes the constraint that when checked on the server prevents the parameter tampering vulnerability on negative quantities.
- *Patch placement*. (Section 7.1.2) Where do we place the patch on the server? In the case of whitebox changes to the server, does it suffice to check the patch constraints at the program entry point? Or, do we check the patch constraints before each sensitive operation?
- *Patch side-effects*. (Section 7.1.3) What are the remedial actions needed when an input does not match the patch constraints? How can one ensure that the placement of the patch does not introduce unwanted side-effects?

7.1.1. Challenges identifying patch constraints

The main challenge for identifying patch constraints 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 the 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 $1111-2222-3333-4444$, the original constraint implied by the drop down menu

$$\boxed{\texttt{card} \in \{1234\text{-}5678\text{-}9012\text{-}3456, 7890\text{-}1234\text{-}5678\text{-}9012\}}$$

changes to the following constraint.

$$\boxed{\texttt{card} \in \{1234\text{-}5678\text{-}9012\text{-}3456, 7890\text{-}1234\text{-}5678\text{-}9012, 1111\text{-}2222\text{-}3333\text{-}4444\}}.$$

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 add and use new cards during the lifetime of a session). Consequently, the challenge is in computing patches that account for possible changes in server-side state.

7.1.2. Challenges in patch placement

A first-cut approach to enforcing client constraints on the server may just involve checking client constraints at the entry point of server-side code. This may appear to be a simple way to solve the patch placement problem. However, this approach may raise false alarms because a parameter tampering vulnerability may be specific to a control path in the server-side code. From the discussion in Section 7.1.1, we know that computing the right constraints for each specific path may be hard. Some paths may not use the parameter (and hence are not vulnerable) or merely use it in non-sensitive operations. Even if used in sensitive operations along a different path, the expected constraints on any input on that path may be different. To illustrate this, let us consider an example. Quite often, server-side code is written to handle multiple distinct forms. For example, it is common to have a common registration module on the server-side (say `register.php`) that handles registrations for admin as well as non-admin users. The registration form for admin users may use a different client form (and a different set of associated constraints) compared to non-admin users. In server-side modules such as `register.php`, typically there are distinctive control paths for admin and non-admin users.

In our running example, the check on the $\texttt{quantity}_1$ must be performed before it is used in a purchase operation:

$$\boxed{\text{exit if (} \texttt{quantity}_1 < 0 \text{ AND } \texttt{op} == \text{“purchase”)}}$$

i.e., forbid execution if the value of the parameter $\texttt{quantity}_1$ is negative and the requested operation is “purchase”. The check on the `op` field ensures that the client constraints are only checked along the path that leads to the (sensitive) purchase operation.

7.1.3. Challenges in side-effect free patching

While introducing patches to the server, It is important to consider the operations that have side effects. Examples of these operations may include database or file updates on the server. It is important to ensure that the introduction of patches does not introduce any operations that have side effects. Although form validation code in web applications are typically performed before most sensitive operations in server code, the possibility of operations with side effects cannot be ruled out. In this case, remediation procedures need to be incorporated in the server code if the inputs do not match the constraints to be enforced. Automatically generating such remediation measures is a challenging, especially because it requires methods to learn and re-employ remediation measures present in the server's code base.

7.2. Whitebox defenses for preventing tampering attacks

As we discussed in Section 5, our whitebox detection was more precise than black-box. So a whitebox solution to fix tampering vulnerabilities is a natural choice. Given the source code of a web application, the two ways in which such patches can be generated and enforced are: (a) insert a reference monitor and forbid malicious behavior or (b) fix the root cause of vulnerability without monitoring. We note that both the above techniques modify source code but as discussed below, vastly differ in their methods.

7.2.1. Prevention solutions: *Monitor and forbid malicious behavior*

Monitoring based prevention solutions have been extensively used in literature e.g., [4,10,12,27,28,34,35,41,47]. These solutions typically embed a reference monitor in the source code to monitor a security relevant property and disallow executions that may violate this property. For example, [4,12,27,28,34,35,41,47] offer to prevent exploitation of SQL injection vulnerabilities by employing monitors that track taint like properties for computed queries to identify injected SQL payloads.

For preventing tampering vulnerabilities using this approach, a runtime monitor could compute client-side constraints when a web form is generated. When this form is submitted for processing, the monitor could then deduce missing validation and enforce it before tampered parameters are used.

The first challenge such an approach faces is in knowing the precise f_{client} to enforce when a form is submitted. Specifically, a user may have opened several web forms (thus generating multiple f_{client} in monitor's records) and the monitor will need a reliable way of using the f_{client} corresponding to the submitted form. In other words, the above discussion elicits a critical need *for associating each generated web form with its corresponding submission*. To do so, a monitor needs to chain together a generated web form (and hence constraints encoded by it) with its submission (hence the knowledge of which constraints to check). One preliminary idea to achieve this is by associating a unique id with each computed f_{client}, embedding the unique id in the form (e.g., as a hidden field), and then checking f_{client} based on unique id submitted with the form.

The second challenge is in determining missing checks in the server-side for a given f_{client}. An inefficient but simple patch would be to check the complete f_{client} again before the form is processed (essentially augmenting `server.php`). Alternatively, by analyzing conditions checked by the server, the monitor could eliminate parts of f_{client} that the server checks correctly. Specifically, to eliminate constraints correctly checked by server, the monitor would need to perform logical deductions about conditions and whether they embody equivalent or stronger constraints than f_{client}. Whether or not this is feasible depends on the nature of the checks and the limitations of state-of-the-art solvers.

Summary. Each submitted form can be associated with f_{client} computed at the form creation site, by using the unique id. Thus, the monitor can essentially lookup and enforce f_{client} associated with the unique id in submitted form data. This unique id serves to associate the right set of constraints to be checked with each form submission addressing the challenges discussed in Section 7.1.2. Further, as the monitor freshly computes f_{client} each time a form is generated, it can correctly handle evolving constraints encoded by forms due to changes in server-side state (Section 7.1.1). Some engineering challenges in this approach, as typical of monitor based solutions, are to ensure that the monitoring code does not change semantics of the original application (except when attacks are detected) and that the performance overheads are acceptable.

7.2.2. Elimination solutions: *Fix the root cause of vulnerabilities*

Unlike monitoring solutions that only forbid malicious behavior, elimination solutions aim to achieve an ambitious goal of fixing the source code. Such solutions are desirable as they aim to repair source code while introducing low performance overheads when compared to monitoring solutions. Privtrans [11] and TAPS [9] are two such examples. Privtrans [11] uses static analysis to split the application for privilege separation whereas TAPS re-writes SQL query generation to employ `PREPARE` statements thus eliminating SQL injection vulnerabilities.

In the context of parameter tampering vulnerabilities, an elimination solution would need to change the server-side code such that it is free of parameter tampering vulnerabilities. Specifically, this entails statically augmenting the form processing code such that it computes and employs the intuitive patch discussed earlier at runtime. However, such a program transformation is quite challenging to realize for the following reasons:

- *Computing input-verifier:* To identify missing checks the solution needs to statically reason about all possible forms an application may generate, constraints they embody and constraints that corresponding form processing files fail to check. The above knowledge seems sufficient to identify missing checks, however it must be generalized to deal with dynamic forms. For example, if a SQL query is used in populating a drop down menu, the form processing code must also issue the same query to retrieve data and then check if the user submitted

input is one of the values returned by the query. Intuitively, the form processing side must mirror processing done by form generation code to correctly validate form fields that rely on server-side state. Further care must be taken to ensure that such mirrored processing is side-effect free; otherwise the application semantics may change (e.g., if form generation code increments a global counter to embed it in the form, the form processing code must not mirror this processing as it would increment the global counter and may result in unexpected behavior).

- *Computing path-selector:* The input-verifier parts of the patches must be placed in the server-side code such that they forbid parameter tampering attempts. To do so, for each form the solution must statically identify server-side code fragments that process it and an ideal program location where the patch can be checked without affecting other control paths. Unfortunately, even when such program locations can be identified, some or all user inputs may have been modified (e.g., due to sanitization or application specific data processing) by statements preceding such program locations. These changes may invalidate input-verifier part of the patch that contains constraints user submitted inputs must satisfy before being sanitized or processed in any way.

Summary. Elimination solutions have great appeal as they can eliminate monitoring overheads while making minimal changes to a codebase. However, as discussed above, developing an elimination solution that addresses both the challenges above effectively seems hard.

7.3. Blackbox defenses for parameter tampering attacks

A blackbox solution for preventing parameter tampering would detect tampering attacks just by analyzing HTTP requests and responses, without analyzing the server code. Such a solution would be platform agnostic and deployment friendly as it can be "plugged-in" to the existing deployment environment without requiring any code changes. Since blackbox defenses do not instrument server-side code, they do not suffer from the problem of server-side-effects (Section 7.1.3). We closely look at the popular approaches in the following areas: (a) Web application firewalls (WAFs), (b) Monitoring using expected models of client behavior and (c) Taming web clients.

Web application firewalls. A typical web application firewall intercepts and analyzes all traffic (HTTP requests and responses) to and from a web application and forbids requests/responses that appear malicious. To differentiate between benign and malicious traffic, these solutions typically undergo a training phase that builds a profile of benign requests/responses. Typically, the duration of the training period is a function of the complexity of the web application and the size of incoming traffic during the training phase.[4] In the commercial world, there are several such solutions

[4]Rarely such training phases lasts more than a month, but they require the active involvement of deployment personnel to add exceptions to reduce false positives (so that benign traffic that appears to be malicious is allowed).

available e.g., Citrix NetScaler,[5] IBM Security Network IPS,[6] F5Networks – Application Security Manager.[7] In the context of parameter tampering vulnerabilities, a WAF solution would need to learn the type/range of benign values that each form field accepts as well as the relationships between form fields. Once this knowledge is available, a WAF solution can identify and reject tampered parameters.

The major challenge for such a solution would be to determine the actual set of constraints that embody a patch. Typically WAFs are good at detecting violations of expected types, e.g., a violation that arises because a string value submitted where a numeric value was expected. This may be effective in detecting a large class of code injection attacks. However, parameter tampering attacks can also be based on data injection, i.e., the tampered parameter value is of expected type but has malicious value. For example, suppose a form field can only have numeric values between 1 and 4. On the one hand, if the WAF accepts all numerical values for this field then attacks with both positive and negative numbers are possible. On the other hand, if WAF only allows numbers seen during the training, it risks rejecting valid inputs due to insufficient training.

The above challenge is further complicated by form fields that change with server-side state, as the training phase for such fields will never be complete. In this regard, a WAF solution must find middle ground such that the training phase is completed in reasonable time and the resulting checks are reasonably effective, i.e., they do not reject any valid values but may fail to reject a high number of malicious values (err on the side of false negatives).

The second challenge is in deciding when to check restrictions learned during the training phase. One idea is to associate the learned restrictions with each form field (alternatively name-value pairs in the HTTP responses). However, this may cause false positives if two forms contain the same form field but one of them requires stricter checks than the other. In this case, the WAF must either distinguish these fields (e.g., if the two forms have at least one uncommon form field) or compromise by selecting the lesser restrictive check (otherwise it would cause false positives).

Summary. WAFs provide a practical compromise by avoiding complicated source code analysis to reduce false positives through a training phase, sometimes assisted by operators. For defending parameter tampering attacks, WAFs must find ways to infer constraints on form fields that go beyond type checking, and ways to enforce sufficiently restrictive checks on user inputs. Whether WAF solutions can be effective in preventing all or a good majority of parameter tampering attacks in a web applications is dependent on the quality of training phase as well as the application (specifically, composition of forms).

[5] http://www.citrix.com/English/ps2/products/product.asp?contentID=2312027.

[6] http://www-01.ibm.com/software/tivoli/products/security-network-intrusion-prevention.

[7] http://www.f5.com/products/big-ip/big-ip-application-security-manager.

Enforcement of models of client behavior. A conceptually closely related line of work [5,21] aims to curb malicious clients in online games by constructing a model of proper client behavior against which behavior of actual clients are compared. Giffin et al. [21] compute a control flow model of expected system calls for the code executing on the client, and Bethea et al. [5] employ symbolic analysis of the client-side code to model an untampered client. Guha et al. [25] compute a model of the expected flow of requests from the client portion (HTML and JavaScript) of Ajax web applications.

The above works are all placed in problem settings where the clients are fixed (i.e. their code bases does not change over executions). For instance, [5] leverages the fact that online games usually have two distinct codebases: one for the server and one for the client. Guha et al. [25] use a model where the web application's clients are fixed (served by the static HTML pages on the server, with all dynamism factored into AJAX calls of the client). In these settings, the client can therefore be analyzed offline, and could also benefit from an expensive analysis. Hence these techniques cannot be directly applied to detect parameter tampering vulnerabilities in applications considered in this paper, because the clients are dynamically generated.

Client replication. Ripley [44] executes two copies of the client-side code: one in a trusted environment and another in a user controlled environment. It then identifies differences in outputs of these two clients as malicious behavior. To do so, it augments the user client code to relay all client-side events (e.g., user interactions) to the trusted copy of the client, which then computes the expected output.

In the context of parameter tampering vulnerabilities, if successfully relayed, validation events will cause the trusted client copy to reject tampered parameters whereas the malicious client will proceed with form submission and hence be rejected. Unfortunately, apart from limitations mentioned in [44], solutions such as Ripley require substantial and careful engineering. Moreover, Ripley requires all "relevant" client events (an application specific concept) to be transmitted to the server. When the number and frequency of relevant events becomes too high, solutions such as Ripley would become prohibitively expensive (e.g., an online game where all mouse-movement events are relevant).

There is another reason that client replication is insufficient for preventing all parameter tampering attacks: some clients assume that data will be entered in a particular order, and when that order is violated the validation they perform is inadequate. Consider a simple web form with two fields `beginDate` and `endDate`. When the user edits `endDate`, the form validates that `endDate` is later than `beginDate`; however, the same validation is not performed when the user edits `beginDate`. A malicious user can therefore violate the intended constraint that `beginDate` comes before `endDate` by first entering data into `endDate` and only then entering data into `beginDate`. Because the form performs different validation depending on the order in which data is entered, an attacker can launch parameter tampering attacks without compromising the client at all. Client replication will therefore not

prevent this kind of attack. Instead, we need methods for extracting and enforcing the validations the client *intended* to perform on its inputs, since the server sees only the data – not the order in which the data was entered.

Associating requests with responses. An interesting class of blackbox solutions [30,31] aims to prevent Cross-site Request Forgery (XSRF) attacks. The key goal of these solutions is to forbid submission of unsolicited forms (i.e., forms not generated by a web application). Typically these solutions embed an XSRF token in each web application-generated web form and reject submitted forms that do not present expected tokens.

The problem of preventing parameter tampering attacks is related to XSRF prevention as the latter also concerns submissions of web forms; however, the analysis of forms performed by XSRF solutions is limited to token management (add an XSRF token to a form and delete the token record after the form is submitted). In contrast, a countermeasure for parameter tampering must understand semantics of forms, i.e., which constraints a form implies and whether the submitted inputs satisfy those constraints.

An interesting property of XSRF tokens is that they can effectively associate each generated form with its submission. An XSRF token is attached to each web application generated URL (GET as well as POST). When a benign user follows these URLs, the resulting HTTP requests contain valid XSRF tokens. However, a malicious entity that attempts to initiate requests on behalf of the victim will fail to present valid XSRF tokens and be rejected. If XSRF tokens are nonces (e.g., [30]) instead of per-session tokens (e.g., [31]) then they are effectively links between the generation of a web form and its submission.

Linking the HTTP requests for form generation and form submission is good starting point to developing a blackbox countermeasure to parameter tampering attacks. If we begin with a nonce-based XSRF defense, we can then consider the possibility of analyzing the client code each time it is generated and using that analysis to check all of that client's submissions. Because the client is analyzed each time it is generated, we can account for any dynamic state changes on the server; furthermore, that analysis can attempt to extract the intent of the client instead of simply replicating its validation behavior. The main drawbacks to this approach are performance-related. Each time the server generates a web form, it must perform program analysis (an expensive operation). Each time the server receives a form submission, it must perform the proper validation. Further research is necessary to understand whether the performance overheads of this approach are within reason.

Summary. XSRF prevention solutions aim to reject unsolicited HTTP requests. For preventing parameter tampering attacks the semantics of web forms must be understood. Nonce-based XSRF tokens form a reasonable basis for building a blackbox parameter tampering defense if web forms can be efficiently analyzed en-route to clients.

8. Conclusion

We presented a formal description of parameter tampering vulnerabilities. Then starting with an intuitive two step high level approach for their detection, we described two tools instantiating that approach: NOTAMPER (suitable for a blackbox setting) and WAPTEC (suitable for a whitebox setting). To the best of our knowledge, NOTAMPER is the first systematic approach for discovering parameter tampering vulnerabilities and WAPTEC is the first approach to constructing parameter tampering exploits by construction. The design and evaluation of these tools clearly indicated the superiority of WAPTEC in precisely discovering parameter tampering vulnerabilities over NOTAMPER. However, NOTAMPER provides an attractive alternate as it neither requires access nor the knowledge of the source code of the application. Our evaluation with NOTAMPER and WAPTEC, clearly establishes the severity of parameter tampering vulnerabilities in open source and commercial web applications. Using these tools, we developed exploits for online banking, online shopping and open source web applications that can result in financial losses, account hijacking, etc. Our survey of existing defenses illustrated that none of them are appropriate to fix parameter tampering vulnerabilities and pointed to the possibility of blackbox solutions being effective for preventing these vulnerabilities. Given that these vulnerabilities are prolific and have severe consequences, further research is warranted to develop defenses for parameter tampering attacks.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research was supported by NSF grants CNS-0716584, CNS-0551660, CNS-0845894, CNS-0917229, CNS-1065537, IIP-1248717 and DGE-1069311. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the government or the National Science Foundation.

References

[1] M. Balduzzi, C.T. Gimenez, D. Balzarotti and E. Kirda, Automated discovery of parameter pollution vulnerabilities in web applications, in: *18th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2011.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda and G. Vigna, Saner: Composing static and dynamic analysis to validate sanitization in web applications, in: *SP'08: Proceedings of the 29th IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2008.

[3] D. Balzarotti, M. Cova, V.V. Felmetsger and G. Vigna, Multi-module vulnerability analysis of web-based applications, in: *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2007.

[4] S. Bandhakavi, P. Bisht, P. Madhusudan and V.N. Venkatakrishnan, CANDID: Preventing SQL injection attacks using dynamic candidate evaluations, in: *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2007.

[5] D. Bethea, R. Cochran and M. Reiter, 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.

[6] P. Bille, A survey on tree edit distance and related problems, *Theoretical Computer Science* **337**(1–3) (2005), 217–239.

[7] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz and V.N. Venkatakrishnan, NoTamper: Automatic blackbox detection of parameter tampering opportunities in web applications, in: *17th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2010.

[8] P. Bisht, T. Hinrichs, N. Skrupsky and V.N. Venkatakrishnan, 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.

[9] P. Bisht, A.P. Sistla and V.N. Venkatakrishnan, Automatically preparing safe SQL queries, in: *FC'10: Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, Tenerife, Canary Islands, Spain, 2010.

[10] P. Bisht and V.N. Venkatakrishnan, XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks, in: *DIMVA'08: Proceedings of the 5th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, Paris, France, 2008.

[11] D. Brumley and D. Song, Privtrans: Automatically partitioning programs for privilege separation, in: *SS'04: Proceedings of the 13th Conference on USENIX Security Symposium*, San Diego, CA, USA, 2004.

[12] G. Buehrer, B.W. Weide and P.A.G. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: *IWSM'05: Proceedings of the 5th International Workshop on Software Engineering and Middleware*, Lisbon, Portugal, 2005.

[13] L. Carettoni and S. di Paola, HTTP parameter pollution, in: *OWASP AppSec Europe 2009*, Poland, 2009.

[14] S. Chong, J. Liu, A.C. Myers, X. Qi, K. Vikram, L. Zheng and X. Zheng, Secure web application via automatic partitioning, *SIGOPS Oper. Syst. Rev.* **41**(6) (2007), 31–44.

[15] E. Cooper, S. Lindley, P. Wadler and J. Yallop, Links: Web programming without tiers, in: *FMCO*, 2006.

[16] B.J. Corcoran, N. Swamy and M. Hicks, Cross-tier, label-based security enforcement for web applications, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2009, pp. 269–282.

[17] M. Emmi, R. Majumdar and K. Sen, Dynamic test input generation for database applications, in: *ISSTA'07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, UK, 2007.

[18] D. Engler, D.Y. Chen, S. Hallem, A. Chou and B. Chelf, Bugs as deviant behavior: A general approach to inferring errors in systems code, in: *18th ACM Symposium on Operating Systems Principles*, Banff, AB, Canada, 2001.

[19] V. Felmetsger, L. Cavedon, C. Kruegel and G. Vigna, Toward automated detection of logic vulnerabilities in web applications, in: *19th USENIX Security Symposium*, Washington, DC, USA, 2010.

[20] K. Fu, E. Sit, K. Smith and N. Feamster, Dos and don'ts of client authentication on the web, in: *Proceedings of the 10th Conference on USENIX Security Symposium, SSYM'01*, Vol. 10, USENIX Association, Berkeley, CA, USA, 2001, p. 19.

[21] J.T. Giffin, S. Jha and B.P. Miller, Detecting manipulated remote call streams, in: *Security'02: Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, USA, 2002.

[22] P. Godefroid, N. Klarlund and K. Sen, DART: Directed automated random testing, *SIGPLAN Not.* **40**(6) (2005), 213–223.

[23] P. Godefroid, M.Y. Levin and D.A. Molnar, Automated whitebox fuzz testing, in: *NDSS'08: Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2008.

[24] Google Web Toolkit, http://www.google.com/webtoolkit/.

[25] A. Guha, S. Krishnamurthi and T. Jim, Using static analysis for AJAX intrusion detection, in: *WWW'09: Proceedings of the 18th International Conference on World Wide Web*, Madrid, Spain, 2009.

[26] W. Halfond, S. Anand and A. Orso, Precise interface identification to improve testing and analysis of web applications, in: *ISSTA'09: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, Chicago, IL, USA, 2009.

[27] W.G.J. Halfond, A. Orso and P. Manolios, Using positive tainting and syntax-aware evaluation to counter SQL injection attacks, in: *FSE'06: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, OR, USA, 2006.

[28] W.G.J. Halfond, A. Orso and A. Orso, AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks, in: *ASE*, 2005.

[29] P. Hooimeijer, B. Livhsits, D. Molnar, P. Saxena and M. Veanes, Fast and precise sanitizer analysis with BEK, in: *20th USENIX Security Symposium*, San Francisco, CA, USA, 2011.

[30] M. Johns and J. Winter, RequestRodeo: Client side protection against session riding, in: *OWASP'06: Proceedings of the OWASP Europe 2006 Conference*, 2006.

[31] N. Jovanovic, E. Kirda and C. Kruegel, Preventing cross site request forgery attacks, in: *SecureComm'06: Proceedings of the Second IEEE Conference on Security and Privacy in Communications Networks*, 2006.

[32] A. Kieżun, P.J. Guo, K. Jayaraman and M.D. Ernst, Automatic creation of sql injection and cross-site scripting attacks, in: *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009.

[33] J.C. King, Symbolic execution and program testing, *Commun. ACM* **19**(7) (1976), 385–394.

[34] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley and D. Evans, Automatically hardening web applications using precise tainting, in: *ICIS'05: Proceedings of the IFIP TC11 20th International Conference on Information Security*, Turin, Italy, 2005.

[35] T. Pietraszek and C.V. Berghe, Defending against injection attacks through context-sensitive string evaluation, in: *RAID'05: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, USA, 2005.

[36] J.W. Ratcliff and D. Metzener, Pattern matching: The gestalt approach, *Dr. Dobbs Journal* **July** (1988), 46.

[37] Sans Windows Security Digest, http://archives.neohapsis.com/archives/sans/2000/0102.html, 2000.

[38] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song, A symbolic execution framework for JavaScript, in: *31st IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2010.

[39] K. Sen, D. Marinov and G. Agha, CUTE: A Concolic Unit Testing Engine for C, in: *10th European Software Engineering Conference*, 2005.

[40] V. Srivastava, M.D. Bond, K.S. McKinley and V. Shmatikov, A security policy oracle: Detecting security holes using multiple API implementations, in: *ACM Conference on Programming Language Design and Implementation*, San Jose, CA, USA, 2011.

[41] Z. Su and G. Wassermann, The essence of command injection attacks in web applications, in: *POPL'06: Proceedings of the 33rd Symposium on Principles of Programming Languages*, Charleston, SC, USA, 2006.

[42] L. Tan, X. Zhang, X. Ma, W. Xiong and Y. Zhou, AutoISES: Automatically inferring security specifications and detecting violations, in: *17th USENIX Security Symposium*, San Jose, CA, USA, 2008.

[43] F. Valeur, G. Vigna, C. Kruegel and E. Kirda, An anomaly-driven reverse proxy for web applications, in: *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC'06*, 2006, pp. 361–368.

[44] K. Vikram, A. Prateek and B. Livshits, 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.

[45] R. Wang, S. Chen, X.F. Wang and S. Qadeer, How to shop for free online – security analysis of cashier-as-a-service based web stores, in: *Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP'11*, 2011, pp. 465–480.

[46] Y. Xie and A. Aiken, Static detection of security vulnerabilities in scripting languages, in: *15th USENIX Security Symposium*, Vancouver, BC, Canada, 2006.

[47] W. Xu, S. Bhatkar and R. Sekar, Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks, in: *SS'06: Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, 2006.