WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction

Prithvi Bisht
University of Illinois
Chicago, USA
pbisht@cs.uic.edu

Timothy Hinrichs
University of Chicago
Chicago, USA
tlh@uchicago.edu

Nazari Skrupsky
University of Illinois
Chicago, USA
nskroups@cs.uic.edu

V.N. Venkatakrishnan
University of Illinois
Chicago, USA
venkat@cs.uic.edu

Abstract

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. Malicious users who circumvent the client can capitalize on the missing server validation. In this paper, we describe WAPTEC, a tool that is designed to automatically identify parameter tampering vulnerabilities and generate exploits by construction to demonstrate those vulnerabilities. WAPTEC involves a new approach to whitebox analysis of the server's code. We tested WAPTEC on six open source applications and found previously unknown vulnerabilities in every single one of them.

Categories and Subject Descriptors

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

General Terms

Languages, Security, Verification

Keywords

Parameter Tampering, Exploit Construction, Program Analysis, Constraint Solving

1. INTRODUCTION

Interactive processing and validation of user input is increasingly becoming the de-facto standard for applications programmed for the Web. With the advent of client-side scripting, there has been a rapid transition in the last few years to process and validate user input in the browser itself, before it is actually submitted to the server. Some of the advantages of client-side processing is the elimination of delays associated with purely server-side data validation, and reduction of server-side loads.

Consider the example of a shopping cart application, where inputs such as the items in the shopping cart, submitted by a user are supplied as parameters to the server side. The server often makes certain assumptions about those parameters, e.g., the credit card expiration date is valid (not a past date). Most of those assumptions are being enforced by JavaScript on the client side, thereby avoiding extra round trips to the server caused by incorrect data entry. However, malicious clients often circumvent the client-side validation (e.g., craft HTTP requests by hand), and supply invalid data to the server. The correct way to program these applications is to ensure that the server performs the same (or stricter) validation checks that are performed at the client. If this is not the case with a server, then it is vulnerable to parameter tampering attacks.

Prior work [7] identifying such vulnerabilities in web applications used a blackbox approach that involved generating opportunities for potential tampering vulnerabilities. This blackbox approach, while being most suitable for testing web sites whose server side code isn't available, involves human labor in converting opportunities to actual exploits.

This paper improves the state-of-art by seeking a fully automated approach to identify the presence of such vulnerabilities in a web application, thus eliminating the need for a human in the loop. Since there is no human in the loop, our approach must identify such vulnerabilities without resulting in false alarms. Therefore, our approach must include mechanisms to *confirm* the existence of each potential vulnerability it identifies.

The basic problem of detecting parameter tampering vulnerabilities is to identify validation checks that are "missing" in a server. This can be done if we have a formal specification of the set of checks that must be performed by the server. Developing such specifications is often done through a manual process, and is a difficult task for legacy applications.

The key idea explored in this paper stems from the observation that in a web application, a client code already constitutes a description of the server's intentions regarding parameter validation checks. We can therefore, extract a specification directly from the client code. This specification can then be used to check the server side code for vulnerabilities.

Using the above observation, we develop a new formulation of this problem of automatically detecting parameter tampering vulnerabilities. In our formulation, a web application is said to be vulnerable when the server-side parameter validation is weaker than client-side validation. In other words, the server performs fewer checks than the client as to the well-formedness of the client supplied input. Such weaknesses point to security vulnerabilities on the server that can be exploited by malicious users. Whenever we find such a weakness, our approach automatically generates a concrete instance of the vulnerability in the form of an *exploit*.

Our approach tool that we call WAPTEC (Whitebox Analysis for Parameter Tampering Exploit Construction), performs web application vulnerability analysis by combining techniques from for-

CCS'11, October 17–31, 2011, Chicago, Illinois, USA.

Listing 1: client.js

```
1  function validateForm(){
2
3      var q = document.getElementById("quantity");
4      var n = document.getElementById("name");
5
6      if(q < 0 || n.length() > 10){
7        return false; // show error, don't submit
8      } else {
9        return true;  // submit form
10     }
11 }
```

Listing 2: server.php

```
1  $ca = $_POST['card'];
2  if($ca matches 'card-1'|'card-2')
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['quantity'] * $price + $shipping;
13
14  if(isset($_POST['discount']))
15    $cost = $cost - $_POST['discount'] * $cost / 100;
16
17  $q = "INSERT INTO orders (`name`, `address`, `card`,
          `cost`) ";
18  $q .= " VALUES ('$n', '$_POST[address]', $ca, $cost)
        ;";
19
20  mysql_query($q);
21  if(mysql_error())
22    $html .= " Please specify an address";
23 }
```

mal logic and constraint solving, symbolic evaluation and dynamic program analysis. Our approach implementation is targeted towards applications written using the LAMP (Linux, Apache, MYSQL, PHP) stack, one of the most widely used development and deployment platforms for web applications.

Due to the inherent multi-tiered nature of a LAMP application, the analysis that we need has to reason about the client side code that validates user supplied inputs, the server side transaction processing logic and (often) the database used for persistent storage. These tiers are implemented as different modules in different languages (HTML / JavaScript, PHP and SQL), and our core analysis needs to abstract the validation logic in each of these tiers and reason about them. While the Links [9, 10] programming language and several other frameworks [1, 8, 2] facilitate principled construction of multiple tiered applications, they are not applicable to reason across the three tiers of existing (legacy) LAMP applications.

To the best of our knowledge, this paper presents the first analysis that presents a uniform framework to reason about the three different tiers of an interactive LAMP application. Since our analysis spans the client, server and database, it is comprehensive and precise about its understanding of the validation performed on web application inputs, and identifies vulnerabilities "by construction". We discuss the design and implementation of this framework in this paper.

We evaluated six open source web applications using WAPTEC and were able to find 45 previously unknown vulnerabilities spanning every single one of these applications. These vulnerabilities have serious real world consequences including privilege escalation to an administrator account, overwriting files on the web server and denial of service. Furthermore, we show how our approach eliminates false positives and false negatives that are inherent in a black-box approach.

This paper is organized as follows: Section 2 presents a running example used in the rest of this paper. Section 3 provides a high-level overview of the basic ideas behind our approach. Section 4 describes the architecture of WAPTEC and its different components. Section 5 presents the implementation of WAPTEC. Section 6 presents an evaluation of our approach over several open source web applications. Section 7 presents related work. In Section 8 we conclude.

2. RUNNING EXAMPLE

Our main thesis is that it is possible to use the client of a web application as a specification of the server's intended behavior. The basis for this thesis stems from the following observations:

- Validation checks that are implemented at a client convey the "intention" of the server side of a web application.
- Server code on occasion does not replicate these intended checks often leading to security flaws.

The second point is worth further elaborating. The reason for the omission of security checks is multi-fold: first, not all web developers are aware of the security concerns about data received from a client cannot be trusted to respect these intended checks and therefore need to be replicated. Secondly, the client and the server often originate from two different codebases, the typical example is that a client is written in JavaScript and the server in one of the many platforms such as PHP, ASP or Java. 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 security violations. In this work, our aim is to detect such mismatches through automated code analysis.

We illustrate the general ideas in this work with the help of a running example. Consider a web application that provides a shopping checkout form with textfields `name`, `address`, item `quantity`, a dropdown menu displaying previously used credit cards to pick the `card` for the current purchase and a hidden field `op` that is set to "purchase". (These fields assume the usual meaning as in a typical shopping session). Listing 1 and 2 list the client side and server side code of this application, respectively.

The client side code in Listing 1 performs its validation checks at lines 6 through 7. The code checks if the `quantity` field is a positive integer, and if the supplied `name` is less than 10 characters, and submits input 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. To illustrate the basic parameter tampering attack, notice that the validation check for `quantity` is not replicated in the server. It is therefore possible that a malicious client can perform this attack by submitting a negative quantity field, reducing the cost computed to a low value.

In order to uncover this attack, the client JavaScript code in Listing 1 must be analyzed, leading to the inference that the constraint on the `quantity` field restricts it to a non-negative number. Similarly, the server PHP code in Listing 2 must be analyzed to infer

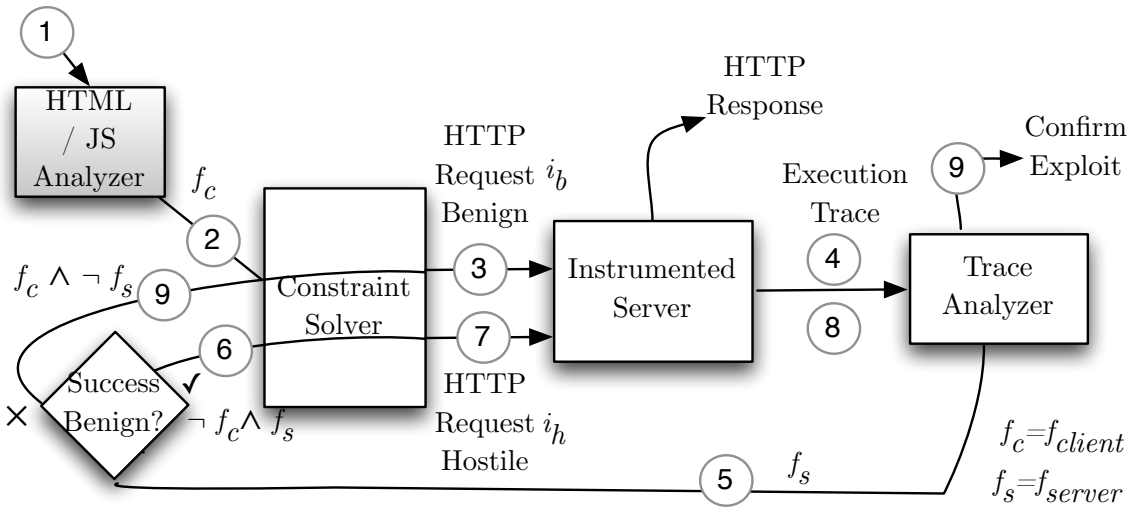

Figure 1: System Workflow

that it does not impose any constraints on this field. In addition, the following challenges need to be addressed as well.

Restrictive servers. While servers occasionally fail to replicate client checks, they are often designed to be more restrictive than clients in processing user input. In our example, note that the client restricts the length of the `name` field to 10 characters or less. On an input that does not meet this constraint (has 11 or more characters), the server chooses to "sanitize" this field by considering only the first 10 characters of the submitted value. A naive approach that doesn't satisfy the client restrictions and fails to consider the effect of sanitization in reaching a sensitive operation on the server will generate a false alarm. Our analysis is designed to factor such changes to input and avoids generating false alarms (§4.2).

Handling database operations. Any server side analysis should not only consider the effect of server side code, but also the effect of its database operations. For instance, database operations may further constrain data submitted by a client through integrity constraints. Failing to consider these constraints will also generate false alarms. For example, say the `address` field in database has an integrity constraint that ensures that it is not `null`. Failing to consider such constraints will generate false alarms. Our approach is designed to correctly handle the effect of such database constraints (§4.3).

Negative Parameter Tampering. Sometimes a server side file, such as `server.php` is written to handle multiple forms. In the above example, the server-side code additionally checks for parameter `discount`. While this code was intended for processing a totally different form that contains discounts for the user, it is not uncommon for LAMP applications to reuse the code that has some shared processing of content. An exploit that introduces this field `discount` can result in providing unlimited discounts to the total price. We call this *negative* tampering, as it is performed by an input field that is not present in the original form. By whitebox analysis of server side code, we are able to identify such vulnerabilities. We found a zero-day negative tampering attack on the open source application `dcpportal` that enables privilege escalation of an ordinary user to an administrator (§6).

3. APPROACH OVERVIEW

WAPTEC's basic approach to identifying parameter tampering exploits (inputs the client rejects but the server accepts) on a web application is a two-step process: (i) find server control paths that if taken result in the input being accepted, i.e., paths that lead to sensitive operations (such as the `INSERT` query in line 17 of our running example), and (ii) find inputs leading to each such control path that the client rejects (such as submitting a negative quantity to the server). In WAPTEC, step (i) is accomplished using a form of constraint-guided search that probes the server with inputs that the server ought to accept and then analyzes the code the server executed to determine if that control path led to a sensitive sink. We call any input the server ought to accept that results in execution of a sensitive operation a *benign input*. Step (ii) is also accomplished by probing the server with inputs and checking for a sensitive sink on the resulting control path, though this time the inputs are those the server ought to reject. Any input the server ought to reject that results in execution of a sensitive operation is a *hostile input*. Hostile inputs are *correct by construction* parameter tampering exploits.

Unlike many bug-finding program analysis efforts, WAPTEC leverages the existence of client-side code (a web form) for both steps. When searching for a benign input in step (i), WAPTEC only generates inputs that the web form accepts and would submit to the server; moreover, because the client code is relatively simple to analyze, WAPTEC extracts a logical representation of all such inputs (f_{client}) and utilizes constraint-solving technology to directly construct an input the client accepts (i.e., without fuzzing). While the server does not accept every input the client accepts, therefore requiring constraint-guided search, the client side code is a good enough approximation that WAPTEC often finds a benign input on the first try.

When searching for attacks on a given control path on the server in step (ii), WAPTEC again uses f_{client} to generate inputs, but in this case the inputs are designed to be hostile. The main thesis of WAPTEC's approach is that if the client code rejects an input, the server ought to reject it as well; thus, every input satisfying the negation of f_{client} is a potential hostile input (parameter tampering exploit), which constraint solvers can again construct directly. Fur-

thermore, WAPTEC uses the logical representation of f_{client} to group all the potential exploits by the vulnerabilities they illustrate and generates one (or any number) of exploits per distinct vulnerability.

Below we describe WAPTEC's two step approach in more detail and refer to the steps shown in Figure 1.

3.1 Finding benign inputs

The purpose of a web form that validates user input is to reject inputs that the server will (or in practice *should*) reject. The converse is also often true: if the web form accepts an input the server will also accept it. We can therefore reasonably treat the constraints the web form checks as an approximate specification for the server's intended behavior. WAPTEC extracts the constraints enforced by the web form (which we call f_{client}) using program analysis, which is accomplished by the HTML / JavaScript Analyzer in step 2 of Figure 1. For our running example, the client formula is $quantity \geq 0 \wedge len(name) \leq 10 \wedge card \in \{card\text{-}1|card\text{-}2\} \wedge op = \text{“}purchase''$ where the first two constraints are contributed by JavaScript and the rest are derived from HTML.

To find a benign input, WAPTEC starts by using its Constraint Solver component to find any input that satisfies f_{client} and then submits that input to the server (step 3). To check whether or not the input reaches a sensitive sink (i.e., is benign), WAPTEC analyzes the code executed by the server using its Trace Analyzer component (step 4). If the server reaches a sensitive sink, the input is benign. However, sometimes the input fails to reach a sensitive sink because the server enforces more constraints than the client. These extra constraints can arise, for example, because the server has more information than the client (e.g., the list of existing usernames). In our running example, the input satisfying f_{client} might be $quantity = 3, name = \text{“}JohnDoe\text{”}, card = card\text{-}1, op = \text{“}purchase''$. The server rejects this input because it requires $address$ to have a non-null value (i.e., $address$ is a required value).

When an input that satisfies f_{client} fails to reach a sensitive sink, WAPTEC attempts to augment f_{client} with additional constraints, the intention being that any input satisfying the augmented f_{client} will lead to a sensitive sink. To compute this augmentation, WAPTEC examines the execution trace of the code the server executed on the failed input, and computes a logical formula representing that code trace (called f_{server}, computed in step 5, by the Trace Analyzer). 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 $f_{client} \wedge \neg f_{server}$ (step 9). In our example, the augmented f_{client} would be $quantity \geq 0 \wedge len(name) \leq 10 \wedge card \in \{card\text{-}1|card\text{-}2\} \wedge op = \text{“}purchase'' \wedge required(address)$, where $required(x)$ means variable x is required to have a value.

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

Once WAPTEC finds a benign input, it performs a depth-limited version of the procedure above to find additional, nearby control paths that lead to sensitive operations. To do that, WAPTEC analyzes the trace 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 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. It is noteworthy that WAPTEC does not perturb a path that has no sensitive sinks because all the paths that it would reach by perturbation are already reachable by the augmentation of f_{client} by $\neg f_{server}$.

3.2 Finding hostile inputs

For each control path WAPTEC finds that leads to a sensitive sink, it attempts to generate inputs that the server ought not accept but that lead to that same sink. Generating inputs the server ought not accept is straightforward: find solutions to the negation of f_{client}, for if the client rejects a given input, we can be assured the server will reject it as well (or else the client fails to expose the server's full functionality to users). Generating inputs that cause the server to follow the same control path and therefore arrive at the same sensitive sink is likewise straightforward: find solutions to f_{server}. Thus, generating inputs that follow the same control path and therefore are accepted by the server but that the server should not accept amounts to finding a solution to $\neg f_{client} \wedge f_{server}$ (step 6). Conceptually, every such solution amounts to a parameter tampering exploit, but to ensure the input is in fact an exploit, we submit it to the server (step 7) and ensure it reaches a success sink (steps 8 and 9).

Furthermore, instead of generating one input for $\neg f_{client} \wedge f_{server}$, WAPTEC generates one input for each disjunct δ in the disjunctive normal form of $\neg f_{client}$ by finding a solution to $\delta \wedge f_{server}$. Each of those inputs satisfies a logically distinct set of constraints and hence is likely to represent a logically distinct vulnerability. Each $\delta \wedge f_{server}$ can be construed as a distinct server-side vulnerability witnessed by one of the exploits WAPTEC finds.

In our running example, the negation of f_{client} is $quantity < 0 \vee len(name) > 10 \vee op! = \text{“}purchase'' \vee card \notin \{card\text{-}1|card\text{-}2\}$. There is a control path through the server where f_{server} includes $required(address) \wedge \neg len(name) > 10$. Thus, to construct an exploit, WAPTEC uses the Constraint Solver to find one solution to the formula $quantity < 0 \wedge required(address) \wedge \neg(len(name) > 10)$ and another solution to the formula $len(name) > 10 \wedge required(address) \wedge \neg(len(name) > 10)$. In the first case, the server executes an INSERT operation, and is deemed an exploit (hostile). This exploit illustrates the vulnerability where $quantity$ is given a negative value. The second formula is not satisfiable and therefore there is no exploit reported.

The pseudo-code for steps (i) and (ii) of our approach can be found in Algorithms 1 and 2, respectively.

3.3 Soundness

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 HTML / JavaScript Analyzer (shown in Figure 1), and is based on our prior work [7]. The analyzer uses symbolic evaluation [20] to compute the client formula f_{client}. Since the formula is statically computed from the

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
```

source, the generated formula 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, $\neg f_{client}$, represents an *over-approximation* of input instances that are rejected by the client (e.g., line 7 of client code listing 1 in our running example). Inputs satisfying $\neg f_{client}$ 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 $\neg f_{client} \wedge f_{server}$ is actually rejected by the client, all the exploits WAPTEC reports are concrete parameter tampering exploits. Our implementation seeks to find such exploits.

3.4 Discussion

Section 2 described several challenges that WAPTEC addresses. Here we explain how those challenges are met by the algorithms just discussed.

Multi-tier analysis. The algorithms above are written as though WAPTEC is faced with analyzing only a single program, but in reality there are three programs written in different languages that it must analyze: the web form, the server code, and the database. To reason about the combination of these three programs, WAPTEC analyzes each program individually and extracts the relevant semantics into logical formulas (more specifically the logic of strings). Once the important portions of the three programs are expressed in a common language, reasoning about the combination is much simpler and can be carried out as described in this section. Details on translating web forms into logic can be found in Section 4.1; details on translating server code (one trace at a time) into logic can be found in Section 4.2; details on translating database code into logic can be found in Section 4.3.

Negative parameter tampering. Discovering attacks that utilize variables not appearing in the client-side web form (i.e., negative parameter tampering attacks) is a natural side-effect of our basic algorithm. Such variables appear in the server-side code, and when the server processes any given input, f_{server} will therefore include those variables. In our running example, line 14 checks if the variable $discount$ has a value. Therefore, every f_{server} generated from an input that fails to set $discount$ will always include the constraint $\neg required(discount)$. When the input fails to reach a sensitive sink, f_{client} is augmented with $required(discount)$, and when the input succeeds in reaching a sensitive sink, the perturbation process includes $required(discount)$ as one perturbation. In both cases, subsequent attempts to find satisfying inputs require $discount$ to be assigned a value.

Sanitization. Sometimes before validating user input, the server sanitizes those inputs. Sanitization violates the premise that if the client rejects an input so should the server. For example, instead of rejecting a $name$ value that is longer than 10 characters, the server truncates $name$ to 10 characters. WAPTEC can avoid triggering false positives for some sanitization cases because of the way it constructs f_{server} from a trace of the server's code (§4.2).

4. WAPTEC ARCHITECTURE

The previous section outlined high level challenges in designing a whitebox analysis tool to detect parameter tampering attacks. Specifically, we note that different components of a web application are written in different programming languages: client side code is written in HTML / JavaScript, server side code is written in server side programming languages such as PHP, JSP, etc., and finally, database schema is written in languages such as SQL. To compute formulas that represent restrictions imposed on inputs, we need to bridge the gap between different programming languages and express constraints imposed by them uniformly in terms of first-order logical formulas. Expressing constraints uniformly would then enable generation of benign and hostile inputs by solving formulas involving f_{client} and f_{server}.

This section discusses technical challenges faced in assimilating constraints from various components of a LAMP web application and algorithms that address them. f_{client} is computed from the client-side code and involves analysis of HTML / JavaScript code relevant to a web form (Section 4.1). f_{server} is computed from the server-side code and involves extracting constraints from PHP server-side code (Section 4.2) and SQL databases (Section 4.3).

4.1 Extracting constraints from client-side code

The client-side web form is typically expressed in HTML and JavaScript both of which encode restrictions on user inputs. We analyze HTML code of the web form to extract constraints implied by various form fields e.g., a drop down menu implies a range constraint on value of the user input. JavaScript validation code associated with the form is symbolically executed to extract conditions that, if satisfied, indicate successful input validation at the client. All restrictions imposed by HTML and JavaScript together then provide the client-side formula f_{client}. Generation of f_{client} is based on our prior work NOTAMPER[7] which provides a detailed treatment.

Listing 3: Trace generated for running example

```
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 }
```

4.2 Extracting constraints from server-side code

The formula f_{server} represents server side validation and sanitization of user inputs. To generate f_{server}, we first capture a trace comprising of statements that the server executed to process user inputs. 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.

To generate f_{server}, we need to identify statements in a trace that correspond to validation / sanitization done by the server side code. The server-side code may perform user input validation and sanitization in the following three ways: a) explicit validation of desired properties of user inputs in conditional statements and b) implicit validation / sanitization of user inputs through inbuilt functions in server-side code and c) implicit validation / sanitization of user inputs by database. In the running example (Listing 2), validation of the `card` parameter at line 2 illustrates explicit validation, truncation of the `name` parameter at line 8 illustrates explicit sanitization (as execution of line 8 ensures that value of the `name` parameter will contain 10 or less characters) and rejection of null value for the parameter `address` exemplifies database sanitization / validation. f_{server} is essentially computed by identifying and analyzing all the three types of validation / sanitization constructs present in a trace. We focus on the first two types of validation / sanitization constructs here and the database validation / sanitization is discussed in the next section (Section 4.3).

Extracting constraints due to explicit validation. Explicit validation of user inputs is captured by `IF` statements appearing in a trace e.g., four `IF` statements shown in the trace in Listing 3, capturing validation of parameters `card`, `name`, `op` and `discount`, respectively. To learn the constraint being checked by an `IF` statement, we analyze its condition argument. Each such condition argument is then repeatedly expanded until it only contains user inputs, concrete values and operators. For example, the `IF` statement on Line 2 (Listing 3) checks if $\texttt{\$main\_ca matches 'card} - \texttt{1|card} - \texttt{2'}$. We expand `$main_ca` with $\texttt{\$\_POST['card']}$ because of the assignment statement on Line 1. Intuitively, starting from the `IF` statement the above process walks backwards in the trace and replaces server-side variables appearing in conditions with values assigned to them until the condition is expressed in terms of inputs, concrete values and operators.

A challenge in precisely capturing explicit validation in `IF` statements stems from the presence of irrelevant statements. A naive approach that considers all `IF` conditions as relevant to a sink 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 check on `card`, it does not prevent the query computed at line 20 from using malicious values of `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.

WAPTEC identifies 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 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:

$$len(name) \leq 10 \wedge op = "purchase" \wedge \neg isset(discount).$$

Extracting implicit constraints due to sanitization. The server-side sanitization of user inputs may inherently enforce constraints on user inputs. For example, at line 8 (Listing 2) server-side variable `$n` which contains value of the parameter `name`, is sanitized. In specific, by truncating the `name` parameter with `substr` function, the server-side code ensures that after this sanitization the contents of `$n` variable will have 10 or less characters i.e., it implicitly enforces the constraint $len\ (name)\ \leq\ 10$.

WAPTEC avoids analyzing paths that would result in generating false alarms due to such sanitization. To see, we revisit the basic process by which WAPTEC identifies paths to a success sink. Notice that we demand that this path is satisfied by an input that satisfies f_{client}. In the event the server chooses to apply sanitization of input to satisfy f_{client}, such a path will not be considered by WAPTEC for trace analysis, because a benign input will never traverse that path. For example, in Listing 2, the statement in Line 8 will never be executed by WAPTEC.

Nevertheless, an application may have incomplete or partial sanitization. To handle these cases, we capture such implicit constraints by analyzing the sink expression (e.g., SQL query), and demanding that f_{client} be held true by the sink expression. We express the sink expression purely in terms of user inputs and concrete values by 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 which contain user inputs (or a function of user inputs). Currently, the restrictions on the operators appearing in the sink expression are limited to the language (shown in Table 1) supported by the underlying solver (as described in §5.2).

4.3 Extracting constraints from database

Database query operations present interesting consequences for approaches that analyze server-side code. With respect to such operations, many security analysis approaches limit their reasoning to reachability, e.g., most tainting approaches aim to find if a tainted data item can reach a database query execution location. Without analyzing outcome of the query execution, such approaches will result in imprecision as database engine may either sanitize hostile inputs to comply with its schema or reject them. For blackbox approaches, database triggered sanitization may result in false alarms. Additionally, whitebox approaches that ignore these constraints may never generate a benign set of inputs that will be truly accepted at the sink. For our running example, without considering 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 `quantity` exploit cannot be constructed without providing a non-null address value.

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. These relations are important from the perspective of sanitization as well. We already discussed a precise way to handle the effect of sanitization that requires the client validation to hold at the sink expression, (and is therefore safe for such operation). However, such an approach needs to make an assumption that the database field corresponding to the sink expression represents a corresponding client form field (that is transformed to the sink expression with some form of sanitization). While the discussions in this section suggest that such an assumption is reasonable across a large class of web applications, and indeed holds in the applications that we analyzed, it is very easy to construe examples where it could break. For instance, consider a (contrived) web application which assigns a sink expression to a value that does not satisfy client validation, and the intention behind such an assignment may be beyond the inference of any automated mechanism. More generally, the above discussion raises the need for a *specification* that provides a mapping between client inputs and database fields. While such specifications were not needed for the applications we analyzed, the availability of such specifications will be able to broaden the applicability of our analysis.

5. IMPLEMENTATION

To generate f_{server}, we need a trace of statements executed by the server-side code. Section 5.1 provides the high-level details behind a program transformation that enables PHP applications to generate a trace and facilitate computation of f_{server}. Generating benign and hostile inputs entails solving logical formulas and Section 5.2 describes the implementation details of the solver.

5.1 Trace generation transformation

Computation of f_{server} entails reasoning about server-side processing of user inputs e.g., properties of user inputs checked by the server-side code. We capture the server-side processing of user inputs in traces which contain program statements executed by the server-side code to process user inputs. To generate such traces we perform source-to-source transformation of applications written in `PHP` language. The transformed applications are then deployed and generate traces apart from processing user inputs.

Alternate implementation. 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 of 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.

Avoiding name collisions. Traces are straight-line `PHP` programs comprising only of assignments, calls to inbuilt functions and `IF-THEN` statements. 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` inbuilt) executed by the web application need 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, program transformation attaches a unique prefix to each variable name being reported in the trace. To compute these prefixes, we use function / method signatures and for variables appearing in classes, a per object unique identifier is used additionally (as described below).

PHP object-oriented features. 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 prefixing signatures to

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

Table 1: WAPTEC **constraint language**

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. Here, the transformer adds an id member variable to the class definition 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 the `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.

As f_{server} mainly concerns processing of user inputs, the transformer ensures that the generated traces only contain statements manipulating user inputs. We use standard taint tracking techniques to track user inputs and only include statements manipulating tainted arguments in traces. 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.

5.2 String solver

The string solver component analyzes logical formulae to construct inputs that are fed to the server; some of those inputs the system was designed to accept, while other inputs are intended to expose server-side vulnerabilities. The string solver component of WAPTEC was built on top of Kaluza [21], a state-of-the-art solver that finds variable assignments satisfying string and numeric constraints. The main challenge in building the string solver component was translating the WAPTEC constraint language into the language supported by Kaluza.

Constraint language. WAPTEC allows all boolean combinations of the atomic constraints shown in Table 1. The equality and numeric constraints are standard; regular expression constraints require a variable to belong to a given regular expression; PHP constraints include functions from PHP and JavaScript such as $trim$ (found in e.g., the MyBloggie application) for removing whitespace from the ends of a string and $strpos$ for computing the index at which one string appears inside another string. Kaluza roughly supports those categories of constraints marked with an asterisk, plus functions for computing the length of a string and concatenating two strings. Thus, translating WAPTEC's constraint language to Kaluza's language requires handling modals and PHP functions.

Static versus dynamic typing. Besides the difference in atomic constraints, there is a more fundamental difference between the constraint languages of Kaluza and WAPTEC. Kaluza requires every variable to have a single type and does not provide functions to cast from one type to another[1], whereas PHP allows variables to take on 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 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.

Untranslatable constraints. Some of WAPTEC's constraints cannot faithfully be 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 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 DCP-Portal application: the $md5$ function computes the MD5 hash of its argument.

For constraints that cannot be translated to Kaluza's language, WAPTEC simply drops 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 WAPTEC always checks if the variable assignment produced by the solver satisfies the original constraints, unsound results are never reported.

Disjunction. As mentioned above, disjunction is employed heavily by WAPTEC, and while Kaluza handles disjunction natively, the search for parameter tampering exploits sometimes requires finding different solutions for different disjuncts in a set of constraints—functionality Kaluza does not support. Thus WAPTEC manages disjunctions itself, sometimes converting to disjunctive normal form (DNF)[2] explicitly.

6. EVALUATION

We evaluated the effectiveness of WAPTEC on a suite of 6 open source PHP applications that were chosen to reflect prevalent application domains in commonplace settings. Table 2 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 WAPTEC was deployed on an Ubuntu workstation (2.45Ghz Quad Intel, 2.0GB RAM).

Experiments. We evaluated our approach by conducting two sets of experiments. In the first set of experiments, we ran WAPTEC to automatically analyze the chosen web forms and identify parameter tampering exploits that are correct by construction. In the second set of experiments, we ran NOTAMPER, a blackbox version of WAPTEC developed in our previous work [7], on the same web forms. We compared the results of the two experiments to quantify

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

[2]In our experience, converting to DNF was usually inexpensive (despite its worst-case exponential behavior) because of the structural simplicity of the constraint sets.

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

Table 2: Summary of WAPTEC **results**

the benefits of using whitebox analysis over blackbox analysis in the context of parameter tampering attacks.

Results summary. The outcome of the first set of experiments is summarized in Table 2. We evaluated one form in each application. WAPTEC *found a total of 45 exploits. We manually verified all of these exploits.* 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 application in the test suite underscoring a widespread lack of sufficient replication of the client-side validation in the corresponding server-side code. A detailed report of exploits found by WAPTEC can be found at *http://sisl.rites.uic.edu/waptec*. We discuss a few interesting exploits below and use them to motivate discussion in Section 6.2 that discusses improvements made by WAPTEC (whitebox) over our prior work NOTAMPER (blackbox).

6.1 Exploits

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. The whitebox view of the server-side code enabled WAPTEC to set this additional parameter and escalate privileges of 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 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-side 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 in this case caused the username existence check to always return false. This is because usernames stored in the database are truncated to 32 characters and checking for usernames of length > 32 will always return false. In addition to this, 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 for the current post 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 computed formulas f_{server} and f_{client} for this form, revealed missing validation of submitted category value by the server-side code and was exploited 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 which contains 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 value not in the drop down menu, an attacker can make the application unusable and thus conduct a denial-of-service attack.

6.2 Comparison of whitebox and blackbox results

The results of the comparison are summarized in Table 3. 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 reported by NOTAMPER, which were eliminated in WAPTEC, and false negatives reported by WAPTEC that NOTAMPER failed to find. In total, the blackbox approach resulted in 23 false positives, and 24 fewer confirmed exploits when compared to the whitebox approach. Further, for `dcpportal` and `mybloggie` applications WAPTEC found several exploitable sinks for each negated disjunct of f_{client} e.g., 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

Application	Conf. exploits		False pos.	False neg.
	BlackB.	WhiteB.	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

Table 3: Comparing whitebox and blackbox analysis results

Application	Formula Complexity			Avg. trace size (KB)	Time (sec)
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	2,082

Table 4: Additional WAPTEC **results**

these disjuncts would have contributed to one hostile each, at best, in NOTAMPER.

In the rest of this section we will refer to exploits described in Section 6.1 to highlight features of WAPTEC (whitebox) that enable it to produce better results than NOTAMPER (blackbox).

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. The duplicate user exploit in `dcpportal` demonstrates a case where a single hostile input exploited multiple sinks. When WAPTEC negated 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. WAPTEC showcased that it can uncover negative tampering vulnerabilities by discovering the privilege escalation exploit in `dcpportal`. While exploring additional server-side form processing code, WAPTEC found a conditional that depended on 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 the cookie `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. 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. As mentioned in Section 4.2, WAPTEC fundamentally avoids paths that may sanitize inputs by computing benign inputs that satisfy f_{client} and hence are not needed to be sanitized. For cases where filter functions appear in conditional expressions, WAPTEC maps built-in functions to constraints implied by them. In contrast, NOTAMPER is unable to detect the presence of sanitization routines on the server-side beyond using simple heuristics to guess. To account for database constraints, WAPTEC adds them into f_{server} and checks for errors / warnings on database operations. Ignoring database constraints can lead to false positives e.g., during testing of the registration form for `dcpportal`, database constraints helped to avoid a false positive. In this example, the hostile input was produced by negating a range constraint on the `birthdate` parameter in f_{client}, and f_{server} did not contain the range constraint. The server's response returned a success page, so NOTAMPER recognized a vulnerability. However, the default action by the database converted the invalid date to '0000-00-00'. Another example was found while testing the `snipegallery` application. The hostile input was produced by negating a length constraint found in f_{client}, and f_{server} did not contain the replicated length constraint. However, database implicitly enforced the length check and this attack did not succeed. Without considering sanitization and database constraints, such false positives cannot be avoided.

Required variables. Another source of false positives for NOTAMPER is attributed to required variables that are enforced only at the server-side. In these cases, the client contains insufficient information to generate a truly benign input that satisfies the server's demand for certain variables. Any required variables in f_{server} can easily be identified in a whitebox approach through code analysis, but have to be heuristically determined in a blackbox approach. For example, NOTAMPER failed to catch the category hijacking exploit in the `mybloggie` application because of missed required variables. In this example, the server-side code required the client to set value of either `submit` or `preview` parameter. As NOTAMPER failed to set any of these values, the server generated a response page containing the same form for both benign and hostile inputs thus resulting in a false positive.

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 could be employed to analyze a wider range of applications and websites. However if the source code is available, a whitebox analysis based approach like WAPTEC could be employed to perform deeper code analysis to pinpoint more security problems. Further, by ensuring production of exploits by construction, the whitebox approach can reduce the human effort in confirming exploits that may be unavoidable in blackbox approaches.

6.3 Complexity and performance

For each evaluated application, Table 4 captures complexity of generated formulas (column 2 - client-side constraints, column 3 - server-side constraints, column 4 - database constraints), average size of generated traces (column 5 - kilo bytes) 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.

7. RELATED WORK

The related work is organized along the dimensions of various contributions of WAPTEC.

Multi-tier reasoning of web applications. Web applications, those following LAMP model in specific, are inherently multi-tiered: 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. [5] 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 [9, 10] and frameworks such as [1, 8] offer principled construction of multi-tiered applications, however do not assist analysis of legacy web applications. In contrast, WAPTEC offers a much powerful analysis framework that combines concolic analysis of the HTML / JavaScript with static analysis of runtime traces for legacy web applications.

Specification inference. AutoISES [25] is an approach for C program bug detection that mines for common security-related patterns and identifies deviations from these as vulnerabilities. Engler [12] detects security bugs in C programs by mining temporal safety patterns and checking for inconsistencies. Srivastava [23] et al. exploit the difference between multiple implementations of the same application programming interface to detect security violations. Felmetsger et al. [13] monitor normal execution of a web application to infer a set of behavioral specification 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 [21, 16, 19, 11, 14, 15, 22]. Saxena et al. Kudzu [21] 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. [16] employ symbolic execution and constraint solving to infer web application interfaces for improved testing and analysis of web applications. Kieżun et al. [19] use symbolic execution and a library of attack strings to find code injection attacks in web applications. Sen et al. [22] propose a technique that combines concrete and symbolic execution to avoid redundant test cases as well as false warnings. Authors of [15, 14] propose techniques to record an actual run of the program under test on either a well-formed input [15] or random inputs [14], 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 WAPTEC aims to find hostile inputs and in that sense is similar to these approaches, our formulation of the parameter tampering problem as one checking the consistency of the server and client codebases and development of web application specific methods such as perturbation that are specialized to this problem makes it distinctive.

Emmi et al. [11] 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 (e.g., [24] and many others) examples in which untrusted data can result in unauthorized actions in a web application. WAPTEC is similar to such studies in the sense that it can find vulnerabilities that could be exploited by SQL injection or Cross-site Scripting attacks. However, WAPTEC 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. Few recent works have focused on automatically discovering parameter pollution [3] and parameter tampering vulnerabilities [7]. Bethea et al. [6] discuss enforcement strategies for misbehaving clients in the context of online games. Jayaraman et al. [18] present an approach to enforce intended sequence of requests in web applications to prevent request integrity attacks.

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 that inadequately sanitize inputs. Saner [4] attempts to identify and validate adequacy of sanitization routines in web applications. It models sanitization performed by web application as an automata and detects inadequacy by finding nonempty intersections with automata characterizing successful attacks. Recently, BEK [17] 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.

8. CONCLUSION

In this paper, we presented WAPTEC, an approach and tool for automatically generating exploits for parameter tampering vulnerabilities. Our approach uses a combination of formal logic and constraint solving, symbolic evaluation and dynamic analysis. We presented an evaluation of six open source applications and our tool was able to find at least one exploit in every single application. Our paper illustrates that it is indeed possible to extract and use specifications of intended behavior from its own (client side) code. The numerous exploits found by our approach further illustrate that there does exist a gap between validation checks that must happen in a web application and those that actually happen.

Acknowledgements

This work was partially supported by National Science Foundation grants CNS-0845894, CNS-0917229 and CNS-1065537. Thanks are due to Kalpana Gondi for her helpful comments. Finally, we thank the anonymous referees for their feedback.

9. REFERENCES

[1] Google Web Toolkit. `http://www.google.com/webtoolkit/`.

[2] Ruby on Rails. `http://www.rubyonrails.org/`.

[3] BALDUZZI, M., GIMENEZ, C. T., BALZAROTTI, D., AND KIRDA, E. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications . In *18th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA, 2011).

[4] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KRUEGEL, C., KIRDA, E., AND VIGNA, G. 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).

[5] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-Module Vulnerability Analysis of Web-based Applications. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA, 2007).

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

[7] BISHT, P., HINRICHS, T., SKRUPSKY, N., BOBROWICZ, R., AND VENKATAKRISHNAN, V. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA, 2010).

[8] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure Web Application via Automatic Partitioning. *SIGOPS Oper. Syst. Rev. 41*, 6 (2007), 31–44.

[9] COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. Links: Web programming without tiers. In *FMCO* (2006).

[10] CORCORAN, B. J., SWAMY, N., AND HICKS, M. 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.

[11] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic Test Input Generation for Database Applications. In *ISSTA'07: Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, UK, 2007).

[12] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *18th ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, 2001).

[13] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *19th USENIX Security Symposium* (Washington, DC, USA, 2010).

[14] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. *SIGPLAN Not. 40*, 6 (2005), 213–223.

[15] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated Whitebox Fuzz Testing. In *NDSS'08: Proceedings of the 15th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA, 2008).

[16] HALFOND, W., ANAND, S., AND ORSO, A. 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).

[17] HOOIMEIJER, P., LIVHSITS, B., MOLNAR, D., SAXENA, P., AND VEANES, M. Fast and Precise Sanitizer Analysis with BEK. In *20th USENIX Security Symposium* (San Francisco, CA, USA, 2011).

[18] JAYARAMAN, K., LEWANDOWSKI, G., TALAGA, P. G., AND CHAPIN, S. J. Enforcing Request Integrity in Web Applications. In *DBSec'10: Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy* (Rome, Italy, 2010).

[19] KIEŻUN, A., J. GUO, P., JAYARAMAN, K., AND D. ERNST, M. 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).

[20] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976).

[21] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2010).

[22] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A Concolic Unit Testing Engine for C. In *10th European Software Engineering Conference*.

[23] SRIVASTAVA, V., BOND, M. D., MCKINLEY, K. S., AND SHMATIKOV, V. 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).

[24] SU, Z., AND WASSERMANN, G. The Essence of Command Injection Attacks in Web Applications. In *33rd symposium on Principles of programming languages* (Charleston, SC, USA, 2006).

[25] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *17th USENIX Security Symposium* (San Jose, CA, USA, 2008).