NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications

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

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

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

Radoslaw Bobrowicz
University of Illinois at Chicago
Chicago, Illinois, USA
rbobrowi@cs.uic.edu

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

ABSTRACT

Web applications rely heavily on client-side computation to examine and validate form inputs that are supplied by a user (e.g., "credit card expiration date must be valid"). This is typically done for two reasons: to reduce burden on the server and to avoid latencies in communicating with the server. However, when a server fails to replicate the validation performed on the client, it is potentially vulnerable to attack. In this paper, we present a novel approach for automatically detecting potential server-side vulnerabilities of this kind in existing (legacy) web applications through blackbox analysis. We discuss the design and implementation of NOTAMPER, a tool that realizes this approach. NOTAMPER has been employed to discover several previously unknown vulnerabilities in a number of open-source web applications and live web sites.

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, Constraint Solving, Blackbox Testing, Symbolic Evaluation

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.

The form processing performed by the browser mostly involves checking user-provided inputs for errors. For instance, an electronic commerce application accepting credit card payments 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. Servers with parameter tampering vulnerabilities are open to a variety of attacks (such as enabling unauthorized access, SQL injection, Cross-site scripting).

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 [8] and Ripley [24] focus on the broader issue of ensuring data integrity in web application development frameworks. The goal of these approaches is to realize *new* web applications that are effectively immune to parameter tampering attacks. In contrast, the focus of this paper is solely on detecting parameter tampering vulnerabilities in *existing* web applications (or legacy applications) that are already in deployment.

Our goal is to develop an approach and a tool that can be used by testing professionals, website administrators or web application developers to identify parameter tampering opportunities. Specifically we aim to determine in a blackbox fashion, if a given web site (i.e., a deployed web application) is vulnerable to parameter tampering attacks, and 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: professional testers using the inputs generated by our tool to develop and demonstrate concrete exploits; web application develop-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'10, October 4–8, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0244-9/10/10 ...$10.00.

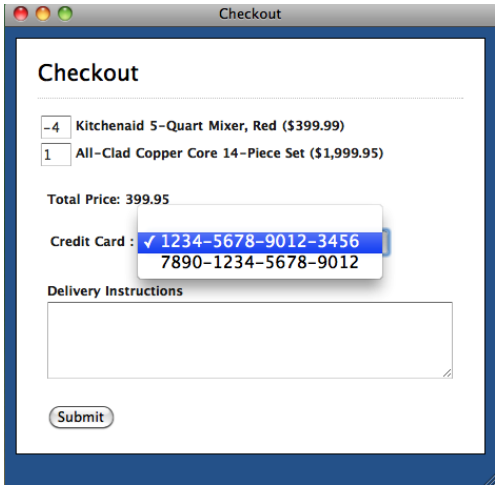

Figure 1: Running example of a shopping application

```
<script type="text/javascript">

function validateForm() {
 var copies, copies2;
 copies  = document.getElementById('copies');
 copies2 = document.getElementById('copies2');
 if(copies.value < 0 || copies2.value < 0){
   alert("Error: Need positive copies");
   return false;
 }
 return true;
}

function validateText() {
 var dir;
 dir = document.getElementById('directions');

 var textRE = /([a-zA-Z])*/;

 var bReturn = textRE.match(dir);
 if(!bReturn)
   alert("Error: No special characters.");
 return bReturn;
}
</script>
```

Figure 2: JavaScript validation for running example. `validateForm()` is called when the form is submitted, and `validateText()` is called when the delivery instructions change.

ers 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 develop the first systematic approach for detecting parameter tampering opportunities in web applications. We implement our approach in a tool that we call NOTAMPER. Our approach makes the following technical advances.
  - Client-side JavaScript code analysis techniques specialized to form validation code.
  - Input-generation techniques that cope with the many challenges of black-box vulnerability analysis.
  - Novel heuristics to generate and prioritize inputs that are likely to result in vulnerabilities.
- We empirically demonstrate NOTAMPER's use by reporting several parameter tampering opportunities from eight open source applications and five online web sites. Furthermore, starting from these opportunities, we develop concrete exploits for a majority of these applications / web sites. Our exploits demonstrate serious security problems: unauthorized monetary transactions at a bank, unauthorized discounts added to a shopping cart, and so on.

This paper is organized as follows. In Section 2, we provide motivation through a running example, formulate the problem precisely, and present a high-level overview of our approach. Section 3 describes the architecture of NOTAMPER and the main technical challenges addressed by our approach. Section 4 describes the algorithms used by NOTAMPER. Section 5 presents our evaluation over several real world examples and web sites. Section 6 presents the related work, and in Section 7 we conclude.

2. HIGH LEVEL OVERVIEW

Figure 1 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. It presents 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), and any special delivery instructions. Before this data is submitted to the server, the client-side JavaScript code (Figure 2) ensures that the quantity for each product is non-negative, and that the delivery instructions include no special characters. The `onsubmit` event handler performs this validation and submits the data to the server if it finds them valid, or asks the user to re-enter with an appropriate error message. The server, however, fails to replicate these validation checks, enabling a number of attacks.

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 `copies` and `copies2`) 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 `payment`). 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 (param-

eter `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.

2.1 Problem Description

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, constraint-checking code can be formulated as a function $\mathcal{I} \rightarrow \{true, false\}$, where $false$ indicates an error. We use p_{client} to denote the 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

$$p_{server}(I) = true \Rightarrow p_{client}(I) = true. \quad (1)$$

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

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

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, requiring the client and server checks to be implemented and maintained independently of one another. Over a period of time, the validation checks in these two code bases could become out of sync, opening the door for parameter tampering attacks.

2.2 Approach overview

Our goal is to automatically construct inputs that exercise parameter tampering vulnerabilities using a black-box analysis of the server. The benefit of black-box server analysis is that our approach is agnostic about the server's implementation (e.g., PHP, JSP, ASP) and is therefore broadly applicable, even including antiquated and proprietary server technology. The drawback of black-box server analysis is that we may not have sufficient information to eliminate false positives and false negatives. In particular, we may not be able to reasonably generate all of the inputs the server should be tested on, and even for those inputs that we do generate, there is no reliable way to know if the server accepts them. Our goal is therefore to identify *opportunities* for parameter tampering while requiring as little manual guidance as possible. In particular, we ask two things of human developers / testers: to provide hints about vital information not present on the client and to check whether or not the parameter tampering opportunities we identify are true vulnerabilities (perhaps by generating actual exploits).

Our high level approach is as follows: On the client, whose source is in HTML and JavaScript, we extract f_{client}: a logical representation of p_{client} using techniques from program analysis. Subsequently, using logical tools, we generate inputs $h_1,\ldots,h_n$ such that $f_{client}(h_i) = false$ for each i. We call each such input *hostile* because it is designed to illustrate a possible parameter tampering attack. In addition, we also generate inputs $b_1,\ldots,b_m$ such that $f_{client}(b_j) = true$ for each j. We call each such input *benign* because it is an input the server will process normally. In our approach, we take hints from developers to confirm that these generated inputs were indeed processed normally.

The benign inputs help assess which hostile inputs represent actual opportunities. We submit each hostile and benign input to the server, producing responses $H_1,\ldots, H_n$ and $B_1,\ldots, B_m$, respectively. We then compare each hostile response H_i to the benign responses $B_1,\ldots, B_m$ to produce a score that represents the likelihood that the server accepted h_i. Intuitively, each of the benign responses represent success messages from the server, and the more similar a hostile response is to the benign responses, the more likely the hostile input was successful and therefore a parameter tampering opportunity.

Finally, the hostile inputs and responses are presented to the human tester ranked by similarity to benign responses. The tester is then free to verify hostile inputs as bona fide parameter tampering vulnerabilities and explore the severity of each vulnerability by sending modified hostile inputs to the server.

Discussion. While we believe observation (1) holds for many interactive form processing applications, sometimes it does not, 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, NOTAMPER can be used in such settings by replacing the automatic extraction of f_{client} from HTML/JavaScript with a manually constructed f_{client}. The construction of benign/hostile inputs and their evaluation then proceeds as described above. In other words, NOTAMPER 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. NOTAMPER can therefore be viewed as a formal verification tool with a program analysis front-end for extracting a specification of intended behavior.

Finally, due to the inherent limitations of black-box analysis, our approach cannot offer guarantees of completeness; rather, we justify the utility of our approach by the severity of the real vulnerabilities we have discovered.

3. ARCHITECTURE & CHALLENGES

In this section, we discuss the architecture of NOTAMPER and the high level challenges addressed by each of its components. In Section 4, we discuss our implementation, focusing on our constraint language and algorithms.

Figure 3 shows the high-level architecture: the three components comprising NOTAMPER and how they interact. First, given a web page, the HTML / JavaScript Analyzer constructs logical formulas representing the constraint-checking function for each form on

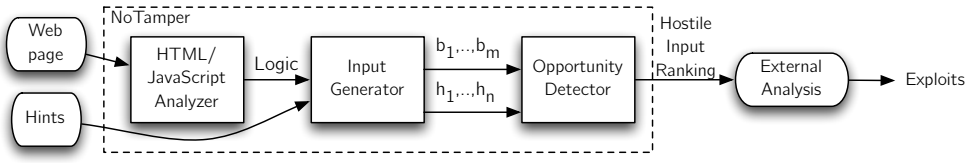

Figure 3: NOTAMPER **end-to-end architecture and application.**

that web page. For our running example, the HTML / JavaScript Analyzer constructs the following formula (f_{client}) that says the parameters `copies` and `copies2` must be greater than or equal to 0; the parameter `directions` must not contain special characters; and the parameter `payment` must be one of the values in the drop-down list.

$$\bigwedge \begin{array}{l} copies \geq 0 \wedge copies2 \geq 0 \\ directions \in \texttt{[a-zA-Z]*} \\ payment \in \\ \quad \texttt{(1234-5678-9012-3456 | 7890-1234-5678-9012)} \end{array}$$

The Input Generator takes the resulting formulas and any hints provided by the user and constructs two sets of inputs for the server: (i) those the server should accept (benign inputs $b_1, \ldots, b_m$) and (ii) those the server should reject (hostile inputs $h_1, \ldots, h_n$). In our example, the Input Generator constructs one benign input (variable assignment that satisfies the above formula):

$$\{copies \rightarrow 0, copies2 \rightarrow 0, directions \rightarrow \text{“”}, \\ payment \rightarrow \texttt{1234-5678-9012-3456}\}.$$

The Input Generator also constructs a number of hostile inputs (variable assignments that falsify the formula above). Below are two such inputs that are the same as above except in (1) `copies` is less than 0 and in (2) `directions` contains special characters.

$$1.\{copies \rightarrow -1, copies2 \rightarrow 0, directions \rightarrow \text{“”}, \\ payment \rightarrow \texttt{1234-5678-9012-3456}\}$$
$$2.\{copies \rightarrow 0, copies2 \rightarrow 0, directions \rightarrow \text{“;}*\&@\text{”}, \\ payment \rightarrow \texttt{1234-5678-9012-3456}\}$$

The third component, the Opportunity Detector takes the hostile and benign inputs, generates server responses for each one, ranks the hostile inputs by how likely they are parameter tampering opportunities, and presents the results to an external tester for further analysis.

Below we discuss the challenges each of the three components addresses in more detail.

3.1 HTML/JavaScript Analyzer

Web page initialization. The JavaScript analysis of NOTAMPER specifically focuses on features / properties that concern form validation and submission. In order to analyze the JavaScript code pertaining to form processing, NOTAMPER simulates an environment similar to a JavaScript interpreter in a browser, including the Document Object Model (DOM). In such an environment, user interactions cause JavaScript code to be executed, resulting in changes to the JavaScript environment and the DOM. (User interactions may trigger asynchronous server requests via AJAX, but our implementation currently does not support AJAX).

To analyze the JavaScript code that actually performs validation, it is often important to understand the global JavaScript state as it exists when the browser first loads the form. To compute this global state, NOTAMPER executes all the initialization code for the web form concretely. It downloads external JavaScript, executes inlined JavaScript snippets, and keeps track of changes to global variables.

Identifying JavaScript validation code. To construct f_{client}, the HTML/JavaScript Analyzer must identify the code snippets relevant to parameter validation and understand how those snippets interact. This can be difficult because validation routines can be run in two different ways: (1) when a form is submitted and (2) in event handlers each time the user enters or changes data on the form.

A state machine naturally models the event-driven execution of JavaScript. Each state 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.

NOTAMPER addresses this challenge by analyzing the JavaScript event handlers as if they were all executed when the form was submitted. The benefit of doing so is 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.

Analyzing JavaScript validation code. Once the validation routines contributing to f_{client} are identified, they must be analyzed. Such code may span several functions each of which may consist of multiple control paths. Each such control path may enforce a unique set of constraints on inputs, requiring an *all-path inter-procedural* analysis. Further, JavaScript may enforce constraints that are not dependent on user inputs e.g., disallow repeated submissions of a form through a global variable. The challenge is to extract only the constraints imposed on inputs by a given piece of JavaScript validation code.

NOTAMPER addresses this challenge by employing a mixed concrete-symbolic execution approach [9] to analyze JavaScript and identify the constraints enforced on user supplied data. Symbolic execution provides coverage of all control paths in the validation code and simulates validation of user supplied data. Concrete execution enables NOTAMPER to ignore code snippets not dependent on symbolic inputs and to provide a suitably initialized environment for symbolic execution.

Resolving document object model (DOM) references. JavaScript validation routines typically use the DOM to access the form input controls. In our simulation of the JavaScript environment, associating DOM references in JavaScript to HTML input con-

trols is non-trivial but necessary for constructing f_{client}. Further, the DOM may be dynamically modified by JavaScript by adding / deleting additional input controls or disabling / enabling existing input controls.

NOTAMPER addresses this challenge by constructing the pertinent portion of the DOM from the given HTML in such a way that it is available to the JavaScript concrete - symbolic evaluation engine during execution. Additionally, this DOM is maintained during the JavaScript evaluation by simulating DOM functions that are used to modify the DOM structure.

3.2 Input Generator

The logical formulas given to the Input Generator are written in the language of string constraints (described in Section 4). The Input Generator encompasses two independent tasks: (i) constructing new logical formulas whose solutions correspond to hostile and benign inputs and (ii) solving those formulas to build concrete inputs. Here we focus on the first task, leaving the second to Section 4.

Avoiding spurious rejections. Two superficial but common forms of server-side parameter validation hide server vulnerabilities from a naïve analysis: checking that all "required" variables have values and checking that all variables have values of the right type. Without accounting for such simple parameter validation, NOTAMPER would have discovered only a few parameter tampering opportunities.

To address this challenge, the Input Generator constructs hostile and benign inputs where all required variables have values and all values are of the right type. NOTAMPER employs heuristics (Section 4), which can be manually overridden, to compute the list of required variables and variable types.

Generating orthogonal hostile inputs. Each hostile input would ideally probe for a unique weakness on the server. Two hostile inputs rejected by the server for the same reason (by the same code path on the server) are redundant. In our running example, the client requires one variable (`copies`) to be greater than or equal to zero and another variable (`directions`) to be assigned a value that contains no punctuation. To avoid redundancy, NOTAMPER should generate one hostile input where `copies` violates the constraints (is less than zero) but `directions` satisfies the constraints (contains no punctuation), and another input where `copies` satisfies the constraints but `directions` does not.

To generate such orthogonal inputs, the Input Generator converts f_{client} to disjunctive normal form (DNF) [1] and constructs a hostile input for each disjunct. Generally, each disjunct represents inputs that violate f_{client} for a different reason than the other disjuncts.

Coping with incomplete information. Sometimes the formula f_{client} fails to contain sufficient information to generate a true benign input or a hostile input that exposes a real vulnerability, yet a human tester is willing to provide that information. For example, many web forms only accept inputs that include a valid login ID and password, but the client-side code does not itself provide a list of valid IDs and passwords; in this case, f_{client} does not contain sufficient information for generating inputs that will be accepted by the server.

To address this issue, the Input Generator accepts hints that guide the search for hostile and benign inputs. Those hints take the form of logical constraints (in the same language as f_{client}) and are denoted σ. For example, to force the login variable `user` to the value "alice" and the password variable `pass` to the value "alicepwd", the user would supply the logical statement $\texttt{user} = \text{“}alice\text{”} \wedge \texttt{pass} = \text{“}alicepwd\text{”}$.

Addressing state changes. Web applications often store information at the server, and web form submissions change that state. This can cause the set of valid inputs to change over time. For example, a user registration web form will ask for a login ID that has not already been chosen. Submitting the form twice with the same login ID will result in a rejection on the second attempt. This is problematic because NOTAMPER submits many different inputs to check for different classes of potential vulnerabilities, yet the login ID is both required and must be unique across inputs.

To address this issue, the Input Generator takes as an optional argument a list of variables required to have unique values and ensures that the values assigned to those variables are distinct *across* submissions. In our evaluation, generating inputs where certain variables all have unique values has been sufficient to address server-side state changes, though in general more sophisticated graybox mechanisms will be necessary (e.g., the ability to roll-back the server-side databases between test cases).

Summary. In total, the Input Generator expects the following arguments (1) the formula logical f_{client} (representing the set of inputs accepted by the client), (2) a list of required variables, (3) types for variables, (4) a manually supplied set of constraints (hints), and (5) a list of unique variables ((4) and (5) are optional). It generates hostile inputs (a set of I such that $f_{client}(I) = false$) and benign inputs (a set of I such that $f_{client}(I) = true$) such that all required variables have values, all values are of the right type, all manual constraints are satisfied, and each unique variable has a different value across all inputs. All arguments to the Input Generator are computed by the HTML/JavaScript Analyzer (as described in Section 4).

3.3 Opportunity Detector

The Input Generator 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 NOTAMPER must ascertain whether or not a given hostile input is accepted by the server while treating the server as a black box.

NOTAMPER 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 `copies` is assigned a negative number. If the server fails to verify that `copies` is a positive number, 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 `copies`, the hostile response will be an error page, which likely differs significantly from the confirmation screen.

4. ALGORITHMS & IMPLEMENTATION

This section details the core algorithms employed by NOTAMPER. All but one of them manipulate a logical language for representing restrictions on user-data enforced by the client. Currently, the language employed by NOTAMPER is built on arithmetic and string constraints. It includes the usual boolean connectives: conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). The atomic constraints restrict variable lengths using $<, \leq, >, \geq, =, \neq$ and variable values using $\in, \notin$ in addition to the above operators. The semantics for the only non-obvious operators, $\in$ and $\notin$, express mem-

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

<*sent*> ::= <*atom*> | <*conj*> | <*disj*> | <*neg*>
<*conj*> ::= (<sent> ∧ <*sent*>)
<*disj*> ::= (<sent> ∨ <*sent*>)
<*neg*> ::= (¬<sent>)
<*atom*> ::= (<*term*> <*op*> <*term*>)
<*op*> ::= < | ≤ | > | ≥ | = | ≠ | ∈ | ∉
<*term*> ::= <*var*> | <*num*> | <*str*> | <*len*> | <*reg*>
<*reg*> ::= `perl regexp`
<*len*> ::= `len` (<*var*>)
<*str*> ::= "<*var*>"
<*var*> ::= `?[a-zA-Z0-9]`⋆
<*num*> ::= `[0-9]`⋆

Table 1: Language of formulas generated by NOTAMPER

bership constraints on regular languages. For example, the following constraint requires x to be a non-negative integer: $x \in$ `[0-9]+`. Table 1 shows a Backus-Naur Form (BNF) grammar defining the constraint language.

Below we describe algorithms in the order they are executed by NOTAMPER: (1) extracting client constraints from HTML and JavaScript, (2) generating the additional inputs accepted by the Input Generator component, (3) constructing logical formulas whose solutions are hostile and benign inputs, (4) solving such logical formulas, and (5) identifying similarity between hostile and benign server responses.

4.1 Client Constraint Extraction

Extracting the constraints enforced by the client on user-supplied data and representing them logically as f_{client}, is done in two steps. First, an HTML analyzer extracts three items from a given web page: (1) constraints on individual form fields, enforced through HTML (2) a code snippet representing JavaScript executed on loading the web page as well as JavaScript executed for parameter validation performed by the client, and (3) a DOM representation of the form. Second, our concrete / symbolic JavaScript evaluator uses (3) during the symbolic evaluation of (2) to extract additional constraints that it then combines with (1). The result is the formula f_{client}.

Step 1: HTML analyzer.

Table 2 summarizes the constraints imposed by each HTML input control through examples. In our running example, there is a drop-down list for the `payment` control that includes two credit card values. The resulting constraint requires `payment` to be assigned one of the values in that list, as shown below:

$payment \in$
`(1234-5678-9012-3456 | 7890-1234-5678-9012).`

The construction of a JavaScript snippet representing the parameter validation performed by the client is accomplished by collecting all the event handlers (and associated scripts) and generating a single function that invokes all those event handlers, returning `true` exactly when all the event handlers return true. All the inlined JavaScript in the web page is then added as a preamble to the above script to initialize environment for the form validation JavaScript. The DOM representation for the form is constructed by recursively building the `document` object in the above JavaScript snippet i.e., the form being analyzed is initialized as a property of the `document` object which captures input controls as properties. Further, the `document` object simulates a small set of core methods that were necessary for processing forms e.g., `getElementById`. Currently, we do not support `document.write` or `document.innerHTML` and we are working towards adding support for these.

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** type=text/password>	**len(x)** ≤ **10**
readonly	<input name=**x readonly** value="20">	**x = 20**

Table 2: Constraints imposed by HTML form controls.

Step 2: JavaScript symbolic evaluator. 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 extract validation constraints, 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 (∧).

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

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

$$\bigwedge \begin{array}{l} \neg(copies < 0 \vee copies2 < 0)) \\ directions \in \texttt{[a-zA-Z]}\star \end{array}$$

The above is then combined with constraint on variable `payment` mentioned before to generate f_{client}.

4.2 Hostile Input Guidance

NOTAMPER's overall success depends crucially on generating interesting hostile inputs. Below we discuss the heuristics the HTML / JavaScript component uses to compute these values from a given web page. These heuristics were tested and refined by manually

examining two of our test applications (SMF and LegalCase) but were left unchanged for the remainder of our experiments.

Initial values. While generating f_{client}, NOTAMPER uses a heuristic to determine the intentions of default values for form fields. Some form fields are initialized with values that are simply illustrative of the kind of input expected, e.g., the value 1 for the number of product copies. Other form fields are initialized with a value that cannot be changed if submission is to be successful, e.g., a hidden field initialized to a session identifier. Currently, NOTAMPER uses the default value for a hidden field as a constraint included in f_{client} and considers the default value for all other fields as illustrative of the expected value. In either case, the list of initial values is provided to the input generator and used for other heuristics as described below.

Types. The type for each variable controls the set of possible values occurring in both the hostile and benign inputs. Choosing appropriate types can greatly improve the odds of success. In our running example, if the type of `copies` were the positive integers, the input generator would never find the vulnerability that appears when `copies` is less than zero. Similarly, if the type of `copies` were all strings, the likelihood that the generator randomly chooses a string that represents a negative integer is unlikely. Currently, NOTAMPER chooses a type for each variable based on (i) its occurrence in arithmetic constraints, (ii) the HTML widget associated with that variable, and (iii) its initial value. Occurrence in an arithmetic constraint implies a numeric type. An HTML widget that enumerates a set of possible values implies a value drawn from the set of all characters in the enumerated values. An initial value that is numeric also implies a numeric type. Integers are assumed unless there is evidence that real values are required.

Required variables. The list of required variables ensures that every hostile input includes a value for every variable in the list. Choosing too small a list risks hostile inputs being rejected because they did not pass the server's requirements for required values, and choosing too large a list can cause the server to reject hostile inputs because unnecessary variables are given invalid values. NOTAMPER employs two techniques for estimating the required variables. One is analyzing the HTML for indications that a variable is required, e.g., asterisks next to field labels. The other is extracting the variables from f_{client} that are required to be non-empty, e.g., the variable cannot be the empty string or the variable must be assigned one of several values (from a drop-down list).

Unique variables. When a variable appears in the unique variable list, every pair of hostile inputs differs on that variable's value. This is useful, for example, when testing user registration pages, where submitting the same user ID twice will result in rejection because the ID already exists. Choosing too large a list, however, can result in fewer hostile inputs being generated and therefore fewer vulnerabilities being found. For example, if a field can only take on one of three values and is required to be unique across all hostile inputs, at most three inputs will be generated. Currently, NOTAMPER is conservative in the variables it guesses should be unique. If there is any indication that a variable can only take on a small number of values, it is not included in the unique list.

4.3 Input Generation

The Input Generator constructs a series of formulas in the constraint language whose solutions correspond to hostile and benign inputs. Here we detail how the construction of formulas for benign and hostile inputs differ.

Benign inputs. To generate benign inputs satisfying f_{client}, NOTAMPER converts f_{client} to DNF [1], augments each disjunct with the user-provided constraints σ and required-variable and type constraints, and finds one solution per disjunct.

$len(\text{<var>}) = len(\text{<var>})$	$\text{<var>} \otimes \text{<var>}$
$\text{<var>} \neq \text{<var>}$	$\text{<var>} \otimes len(\text{<var>})$
$\text{<var>} \neq len(\text{<var>})$	$len(\text{<var>}) \otimes len(\text{<var>})$
$len(\text{<var>}) \neq len(\text{<var>})$	$\text{<var>} \oplus \text{<reg>}$

Table 3: The reduced constraint language: $\wedge$ and $\vee$ over the above atoms. $\otimes$ is one of $<, >, \leq, \geq$. $\oplus$ is either $\in$ or $\notin$.

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

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

NOTAMPER finds one solution for $copies > 0 \wedge directions \in \texttt{[a-zA-Z]}\star$ and another for $copies = 0 \wedge directions \in \texttt{[a-zA-Z]}\star$. If the type of $copies$ is `[0-9]+` and the type of $directions$ is `[a-zA-Z0-9]`$\star$, NOTAMPER includes the constraints $copies \in$ `[0-9]+` and $directions \in$ `[a-zA-Z0-9]`$\star$. If the variable $name$ is required and has type `[a-zA-Z]`$\star$, NOTAMPER includes the constraint $name \in$ `[a-zA-Z]`$\star$. If σ is nonempty, NOTAMPER includes it as well.

Satisfying the unique variable constraint is accomplished by keeping track of the values assigned to each variable for each generated input and adding constraints that ensure the next value generated for each unique variable is distinct from those previously generated.

Hostile inputs. To generate hostile inputs, NOTAMPER starts with $\neg f_{client}$ instead of f_{client} and then proceeds as for the benign case with one exception: filling in values for required variables. Consider any disjunct δ in the DNF of $\neg f_{client}$. If all the required variables occur within δ, NOTAMPER simply finds a variable assignment satisfying δ and returns the result; otherwise, NOTAMPER augments that assignment with values for the required variables not appearing in δ. To do so, it finds values that satisfy f_{client}. The hope is that if the server rejects the input it is because of the variables appearing in δ, not the remaining variables; otherwise, it is unclear whether or not the server performs sufficient validation to avoid the potential vulnerability δ.

In the example above, the disjunctive normal form of $\neg f_{client}$ produces a formula with two disjuncts.

$$\bigvee \begin{array}{l} \neg(copies > 0) \wedge \neg(copies = 0) \\ \neg(directions \in \texttt{[a-zA-Z]}\star) \end{array}$$

Suppose that both $copies$ and $directions$ are required. The first disjunct does not include $directions$, and the second does not include $copies$. After solving the first disjunct with, for example, $copies = -1$, NOTAMPER assigns $directions$ a value that satisfies the original formula, i.e., that satisfies $directions \in$ `[a-zA-Z]`$\star$. Likewise, after solving the second disjunct producing a value for $directions$, NOTAMPER assigns $copies$ a value that satisfies the original formula, e.g., $copies = 1$.

4.4 Constraint Solving

To solve formulas in the constraint language, NOTAMPER uses a custom-written constraint solver built on top of HAMPI [13], a solver that handles a conjunction of regular language constraints on a *single* variable of a fixed length. Our formula involves multiple variables, and therefore we developed our own procedure that uses HAMPI as described below.

NOTAMPER handles disjunction by converting a given formula to DNF [1] and solving each disjunct independently. For a given

Algorithm 1 SOLVE(vars, ϕ, asgn, BOUNDS)

```
 1: if vars = ∅ then return asgn
 2: values := ∅
 3: var := CHOOSE(vars, φ, asgn, BOUNDS)
 4: for all i in LOW(BOUNDS(var)) .. HIGH(BOUNDS(var)) do
 5:    if NUMERIC-VAR(var) then
 6:       if SAT(φ, asgn ∪ {var → i}) then
 7:          newasgn := SOLVE(vars–{var}, φ, asgn ∪ {var→ i},
             BOUNDS)
 8:          if newasgn ≠ unsat then return newasgn
 9:    else
10:       if not SAT(φ∧ len(var)=i, asgn) then goto next i
11:       loop
12:          val := HAMPI(φ|var ∧ var ∉ values, i)
13:          if val = unsat then goto next i
14:          values := values ∪ {val}
15:          if SAT(φ, asgn ∪ {var → val}) then
16:             newasgn := SOLVE(vars–{var}, φ, asgn ∪
                {var→val}, BOUNDS)
17:             if newasgn ≠ unsat then return newasgn
18: return unsat
```

disjunct (which is a conjunction), NOTAMPER performs type inference to determine which variables are numeric and which are strings, extracts bounds on the size of all variables, and simplifies the disjunct to produce a conjunction of atoms from Table 3. Then applies Algorithm 1 to search for a variable assignment satisfying the resulting conjunction.

Algorithm 1 takes as input a list of variables that require values, a logical formula, a partial variable assignment, and a function that maps each variable to that variable's bounds. It either returns $unsat$ (denoting that no satisfiable assignment is possible) or an extension of the given variable assignment that satisfies the logical formula.

The first step of the algorithm is choosing a variable to assign. Currently, NOTAMPER chooses the variable with the smallest range of possible lengths. Then search commences. String variables and numeric variables are treated differently. For numeric variables, NOTAMPER loops over possible values and for each one checks that assigning the variable the current loop value satisfies the constraints. If satisfaction holds, the variable is assigned the loop value.

For strings, NOTAMPER loops over possible lengths (as opposed to possible values), and for each one satisfying the length constraints invokes HAMPI to generate a variable assignment. HAMPI takes as input a logical formula with one variable and a length for that variable. It either returns $unsat$ or a value satisfying the formula. Reducing the given formula ϕ with multiple-variables to a formula with just the chosen variable, denoted $\phi|_{var}$, is performed by selecting the subset of constraints where only the chosen variable occurs. If HAMPI finds a satisfying value, the algorithm checks that the value satisfies the relevant constraints HAMPI does not check: those constraining multiple variables. Additionally, the algorithm keeps a list of values HAMPI returns so that if the search fails at a later point in the search, and another value needs to be generated for the current variable, we can augment the logical formula given to HAMPI to require a value not already chosen.

Once a variable has been assigned a value, Algorithm 1 recurses on the original variable list after having removed the chosen variable, the original logical formula, the original variable assignments augmented with the chosen variable's assignment, and the original variable bounds. When the variable list becomes empty, the algorithm returns the given variable assignment, indicating that all constraints are satisfied by that assignment. If no such assignment can be found, the algorithm returns $unsat$.

4.5 HTML Response Comparison

In order to determine whether hostile inputs were accepted by the server, our approach compares the server's response against a response that is known to have been generated by benign (valid) inputs. Since the server's responses are in HTML, we have to 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. [5]). 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 Obershelp algorithm [16] 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.

Complexity. Our comparison strategy for HTML files is based on the gestalt pattern matching procedure [16], which itself finds the longest common subsequence between HTML files, and then recursively finds the common elements to the left and right of the

Application	Forms	Hostile Inputs	Pote. Oppo.	Conf. Exploit?	Conf. FP
SMF	5	56	42	✓	8
Ezybiz	3	37	35	✓	16
OpenDB	1	10	8	✓	1
MyBloggie	1	8	8	✓	7
B2evolution	1	25	21		2
PhpNuke	1	6	5	✓	4
OpenIT	3	28	27	✓	0
LegalCase	2	13	9	✓	0
smi-online.co.uk	1	23	4		2
wiley.com	1	15	4		2
garena.com	1	4	4		1
selfreliance.com	1	5	1	✓	0
codemicro.com	1	6	1	✓	0

Table 4: Summary of NOTAMPER results (Opportunities:169, Examined: 50, Confirmed exploits: 9, False Positives:43).

Application	Lines of Code	Files	Client-Side	Use
Ezybiz	186,691	1,103	HTML+JS	Busn Mgt
Mybloggie	9,431	59	HTML+JS	Blog
OpenDB	92,712	273	HTML+JS	Inventory
SMF	97,304	166	HTML+JS	Forum
OpenIT	114,959	335	HTML+JS	Support
Legalcase	58,198	195	HTML	Inventory
PHP-Nuke	228,058	1,745	HTML+JS	Content Mgt
B2evolution	167,087	531	HTML	Blog
smi-online.co.uk			HTML	Conference
wiley.com			HTML+JS	Library
garena.com			HTML	Gaming
selfreliance.com			HTML	Banking
codemicro.com			HTML+JS	Shopping

Table 5: NOTAMPER analyzed 8 open source applications and 5 live websites

common sequence. Our procedure has linear complexity in its best case and has quadratic worst-case complexity.

4.6 Implementation

The HTML analysis was implemented on top of the APIs provided by the HTML Parser[2], specifically using visitors for <form> and <script> tags. The JavaScript analysis was performed using a modified Narcissus JavaScript engine-based symbolic evaluator. Narcissus is a meta-circular JavaScript interpreter that uses SpiderMonkey JavaScript engine's interfaces.

The Input Generator was built as a wrapper around the solver HAMPI[13] using the subroutine library Epilog[3] for manipulating logical expressions written in KIF[4]. It consisted of 1700 lines of Lisp code.

The Opportunity Detector was primarily implemented in Java. Based on inputs generated by the constraint solver, a Java-based module relayed HTTP requests to the test server, saved the responses for processing, and implemented algorithm to compute the difference rank.

5. EVALUATION

Test suite and setup. We selected 8 open source applications and 5 live websites. To choose the open source applications, we visited http://opensourcescripts.com and found applications that are heavily reliant on web forms (mainly blogs, business and management applications) and do not use AJAX. To choose the live websites, we selected forms we used personally that seemed likely to contain flaws (e.g., one of the authors has an account at the exploited bank). Table 5, provides some background details for these applications. For open source applications, columns 2 and 3 show the lines of code and number of files, respectively. Column 4 shows the type of constraints enforced by the evaluated forms and the last column shows the functionality provided by the application. We deployed the applications on a Linux Apache web server (2.8GHz Dual Intel Xeon, 6.0GB RAM) and our prototype implementation NOTAMPER ran under Ubuntu 9.10 on a standard desktop (2.45Ghz Quad Intel, 2.0GB RAM).

[2]http://htmlparser.sourceforge.net/
[3]http://logic.stanford.edu/
[4]http://www-ksl.stanford.edu/knowledge-sharing/kif/

5.1 Summary

Our experimental findings are summarized in Table 4. For each application (column 1), the table includes the number of forms analyzed (column 2), the number of hostile inputs NOTAMPER generated (column 3), the number of tampering opportunities (column 4), and whether or not we were able to confirm a vulnerability for that application (column 5). The last column lists the number of confirmed false positives.

When deployed by a web developer to analyze a web application, column 4 is of primary interest. A developer need only look through those hostile inputs that were accepted by the server, and for each one manually decide whether or not the server is actually vulnerable. When deployed by testers (blackhat team), they may confirm exploits by further experimenting with the accepted hostile inputs. In a similar spirit, we tried to confirm at least one exploit in each application. The effort involved to examine 50 of the total 169 opportunities was moderate and required an undergraduate student only a week of effort. We anticipate seasoned developers and testers familiar with their applications to take much less time. During this effort, we developed working exploits in 9 out of 13 applications. Below we highlight some of the exploits we discovered.

5.2 Details of Exploits

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 menu to choose the source account for the transfer, and uses another drop-down menu to choose the destination account. Both drop-down menus 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 drawn from the drop-down menus. 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 drawn from the drop-down menus). 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; how-

Application	Formu. Comp.	Pote. Oppo.	HT-ML	JS	Hid-den
SMF	17	42	28	4	10
Ezybiz	28	35	19	11	5
OpenDB	29	8	8	0	0
MyBloggie	23	8	8	0	0
B2evolution	47	21	8	0	13
PhpNuke	6	5	4	0	1
OpenIT	20	27	21	3	3
LegalCase	13	9	3	0	6
smi-online.co.uk	36	4	2	1	1
wiley.com	20	4	4	0	0
garena.com	10	4	4	0	0
selfreliance.com	9	1	1	0	0
codemicro.com	12	1	0	1	0

Table 6: Details of NOTAMPER results.

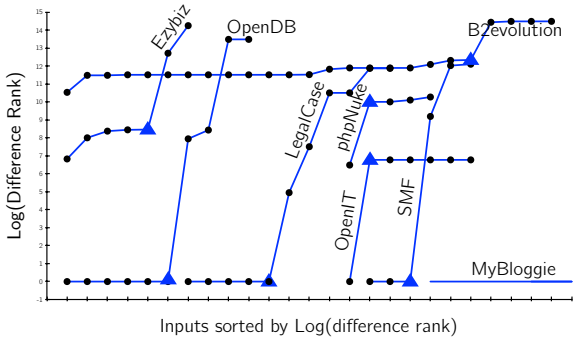

Figure 4: Graph illustrating the importance of hostile input ranking, with bold triangles denoting thresholds used.

ever, 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 cheques. 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 supplied a negative number for one of the quantity fields (and submitted through a proxy). The resulting HTML page, while containing a different total and quantity than the benign input, was otherwise identical, and thus NOTAMPER ranked it as a parameter tampering opportunity.

We were able to further develop this into another serious exploit: we were able to add an item with negative quantities by disabling JavaScript in the browser. When JavaScript was re-enabled, the application computed the total purchase price 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 hours.

Privilege escalation. The `OpenIT` application stores user profiles and employs a web form to allow users to edit their profiles. After logging in, the application provides the user with a web form for editing her profile. Included in that form is the hidden field `userid`, where the application stores the user's unique identifier. When the form is submitted, the server updates the profile for the user identifier corresponding to `userid`. By changing `userid` to that of another user, it is possible to update any user's profile.

When NOTAMPER analyzed this form, it generated a hostile input where the value for `userid` was the number 2 (as opposed to the initial value 1). The server's response was virtually identical to the benign input response (where the value was set to 1), and was therefore reported as a tampering opportunity.

After confirming this vulnerability, we enhanced the exploit so as to modify the profile of an admin user to include a Cross-site Scripting (XSS) payload. Every time the admin user logged in, the script would execute and send the admin cookie to a server under our control. With the help of the stolen `cookie` we then re-constructed and hi-jacked the admin session, thus gaining all the privileges of the admin. This experiment demonstrates that parameter tampering vulnerabilities could be used as a launch pad for other privilege escalation attacks.

Summary of other exploits. The supplemental website [1] provides details of the above exploits and the others found by NOTAMPER. In the `phpNuke` application, tampering of a hidden `name` field allowed us to bypass a CAPTCHA challenge and a confirmation page during the registration process (work-flow attack). In the `OpenDB` application, an XSS script was injected through a tampered `country` field. In the `SMF` application, tampering of vote `option` radio button violated integrity of the voting results.

5.3 Other Experimental Details

False positives. All FPs were either (a) pertaining to the `maxlength` constraints on form inputs that couldn't be exploited to any serious vulnerability or (b) rewritten by the server without any observable difference in HTML output (12 for the `Ezybiz` application).

Categorizing potential vulnerabilities. Table 6 provides more details of our experiments, categorized by application. Column 2 shows the average formula complexity for the client-side constraints, i.e., the average number of boolean connectives and atomic constraints. Column 3 shows the total number of tampering opportunities. Column 4 shows the number of potential vulnerabilities derived from HTML input controls other than hidden fields; Column 5 shows the number of potential vulnerabilities due to JavaScript; and Column 6 shows the number derived from hidden fields.

Hostile input ranking. For each form input NOTAMPER issued an HTTP request to the appropriate application and computed the difference rank (edit distance in bytes) of the response as described previously. A sorted list of the difference rank is produced for each application. In our experience, it is easy to identify the threshold limits for a potential parameter tampering opportunity, as the difference rank between inputs potentially accepted by the server tend to be at least an order of magnitude smaller than the ones potentially rejected by the server.

We use the graph in the Figure 4 to illustrate the thresholds. For space reasons, we only chose one form from each application to be represented in this graph, although our approach tested several forms in every application. Since we are only interested in showing a threshold, the graph plots the logarithm of the difference rank in the Y-axis, with the X-axis representing the various input points sorted according to their difference ranks. We identify the thresholds for various forms using a bold triangle, and we classify those inputs below the threshold as parameter tampering opportunities. It is clear from the graph that such thresholds exist as denoted by steep rises in the difference ranks.

Manual intervention. For each web form, we manually provided certain kinds of hints to NOTAMPER pertaining to information not present on the client but that a human tester might provide. For example, in the `SMF` application, the server required a *valid* login name to access the form, and so we provided such a name to NOTAMPER. Throughout all the forms, we added one of three hints: credentials or session cookies, inputs required by the server (required variables list), and variables required to be unique across invocations (unique variables list). (See Section 3 for more details.)

To discover such restrictions, we used NOTAMPER to generate an input satisfying the client-side constraints (f_{client}). If this input was rejected, we examined why and provided hints that ensured NOTAMPER could generate a benign input accepted by the server.

A total of 3 unique-variable hints were added in our experiments (`SMF`: 2, `phpNuke`: 1). For every application except `phpNuke`, we supplied a cookie with a valid session id. Further, a total of 12 required variable hints were supplied in all forms (`SMF`: 5 in 3 forms, `phpNuke`: 4, `B2evolution`: 1, `garena.com`: 2). This manual intervention is bounded by the number of input fields on a form and typically required less than 5 minutes per form. We expect this process to be simpler for a real tester who is familiar with the application being tested.

Performance. The most computationally expensive component of NOTAMPER was the Input Generator. The HTML / JavaScript Analyzer ran in under a second for the most elaborate form in our test suite. The Opportunity Detector ran in sub-second time for each application, ignoring the delays between consecutive HTTP requests built-in to avoid overloading the server. The most expensive step of Input Generation was constraint solving; the remainder of the Input Generation component ran in under a second. Over the 22 forms, the constraint solver solved 315 formulas in a total of 219 seconds, giving an average time of 0.7 seconds per input. Such performance is acceptable for an off-line analysis tool such as NOTAMPER.

6. RELATED WORK

Symbolic evaluation. A number of research approaches have used symbolic execution to address a wide range of security problems, e.g., automated fingerprint generation [7] and protocol replay [15]. Our own recent work [6] also applied this technique to eliminate SQL injection attacks in legacy web applications by retrofitting PREPARE statements through automated code transformation.

Research on input validation methods. The lack of sufficient input validation is a major source of security vulnerabilities in web applications. 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 [14, 12, 21, 4] and Cross-site Scripting [20, 23, 22] are well studied examples in which untrusted data can result in unauthorized actions in a web application.

Vulnerability analysis. There has been intense interest in analyzing JavaScript code for the purpose of detecting security flaws. Kudzu [18] reduces JavaScript to string constraints for the purpose of detecting *client-side* attacks, whereas our focus is utilizing JavaScript analysis to discover *server-side* flaws. Our problem setting has enabled us to specialize our concrete / symbolic evaluation and constraint solving with many aspects of form processing, e.g., processing client-side formulas to generate logical queries that are likely to succeed as tampering vulnerabilities and the development of many practical heuristics. There are also approaches that perform white-box analysis of server side code for identifying such vulnerabilities [2, 3]. However, there is little work on systematic analysis of the kind of parameter tampering problems that were addressed in this paper.

Fuzzing/Directed testing. Fuzz and directed testing approaches [9, 10, 19] aim to apply random/guided mutations to well-formed inputs to discover vulnerabilities in a blackbox [19] or a white-box [10] fashion. In that sense, NOTAMPER is similar to these approaches as it generates hostile inputs to discover vulnerabilities. However, our formulation of the parameter tampering problem as one checking the consistency of the server and the client code bases and development of methods specialized to this problem makes it different from these approaches.

Prevention architectures. New browser architectures [11, 17, 25] propose to sandbox the client side code of applications to prevent undesired interactions. Recent works have also aimed at ensuring that the server side of a web application remains protected from malicious clients. Ripley [24] aims to detect malicious activities at the client by replicating the client execution in a trusted environment. SWIFT [8] uses information flow analysis during the development of new applications to ensure that constraints regarding information flow confidentiality and integrity will be met in client side code. NOTAMPER's goals are very different from these approaches as we focus on discovering vulnerabilities in existing (legacy) applications.

7. CONCLUSION

In this paper, we described NOTAMPER, a novel approach for detecting server-side HTTP parameter tampering vulnerabilities in web applications. We formulated our problem in terms of the constraints implied on user data by client-side code, advocated program analysis as a way of extracting those constraints, and employed constraint solving to generate tampering opportunities. Our work exposed several serious exploits in existing open source web applications and web sites, and we expect the number of discovered vulnerabilities to grow as we analyze more applications. Our results highlight a significant gap between the server-side parameter validation that *should* occur and the server-side validation that *does* occur in today's web applications.

NOTAMPER currently employs black-box server-side analysis, but in the future we expect to add white-box analysis. White-box analysis will reduce false positive/negative rates and the manual labor required to run the tool and analyze its results; however, the white-box capability will be an optional feature, allowing NOTAMPER to continue being applicable to web forms for which white-box analysis is infeasible.

Acknowledgements

This work was partially supported by National Science Foundation grants CNS-0716584, CNS-0551660, CNS-0845894 and CNS-0917229. Thanks are due to Mike Ter Louw and Kalpana

Gondi for their helpful comments. Finally, we thank the anonymous referees for their feedback.

8. REFERENCES

[1] NoTamper Supplementary Website. `http://sisl.rites.uic.edu/notamper`.

[2] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., 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, California, USA, 2008).

[3] Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. Multi-Module Vulnerability Analysis of Web-based Applications. In *CCS'07: 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA, 2007).

[4] Bandhakavi, S., Bisht, P., Madhusudan, P., and Venkatakrishnan, V. CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications security* (Alexandria, Virginia, USA, 2007).

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

[6] Bisht, P., Sistla, A. P., and Venkatakrishnan, V. 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).

[7] Brumley, D., Caballero, J., Liang, Z., Newsome, J., and Song, D. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *SS'07: Proceedings of 16th USENIX Security Symposium* (Berkeley, California, USA, 2007).

[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] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. *SIGPLAN Not. 40*, 6 (2005), 213–223.

[10] Godefroid, P., Levin, M. Y., and Molnar, D. A. Automated Whitebox Fuzz Testing. In *NDSS'08: Proceedings of the 16th Annual Network and Distributed System Security Symposium* (San Diego, California, USA, 2008).

[11] Grier, C., Tang, S., and King, S. T. Secure Web Browsing With the OP Web Browser. In *SP'08: Proceedings of the 29th IEEE Symposium on Security and Privacy* (Oakland, California, USA, 2008).

[12] Halfond, W. G., Viegas, J., and Orso, A. A Classification of SQL-Injection Attacks and Countermeasures. In *ISSE'06: Proceedings of the International Symposium on Secure Software Engineering* (Washington, DC, USA, 2006).

[13] Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P., and Ernst, M. D. HAMPI: A Solver for String Constraints. In *ISSTA '09: Proceedings of the 18th international symposium on Software testing and analysis* (Chicago, Illinois, USA, 2009).

[14] Livshits, V. B., and Lam, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *SS'05: Proceedings of the 14th USENIX Security Symposium* (Baltimore, Maryland, USA, 2005).

[15] Newsome, J., Brumley, D., Franklin, J., and Song, D. Replayer: Automatic Protocol Replay by Binary Analysis. In *CCS'06: Proceedings of the 13th ACM conference on Computer and communications security* (Alexandria, Virginia, USA, 2006).

[16] Ratcliff, J. W., and Metzener, D. Pattern Matching: The Gestalt Approach. *Dr. Dobbs Journal* (July 1988), 46.

[17] Reis, C., and Gribble, S. D. Isolating Web Programs in Modern Browser Architectures. In *EuroSys'09: Proceedings of the 4th ACM European conference on Computer systems* (Nuremberg, Germany, 2009).

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

[19] Saxena, P., Hanna, S., Poosankam, P., and Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium* (San Diego, California, USA, 2010).

[20] Saxena, P., Song, D., and Nadji, Y. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS'09: Proceedings of 16th Annual Network & Distributed System Security Symposium* (San Diego, California, USA, 2009).

[21] Su, Z., and Wassermann, G. The Essence of Command Injection Attacks in Web Applications. In *POPL'06: Proceedings of the 33rd symposium on Principles of programming languages* (Charleston, South Carolina, USA, 2006).

[22] Ter Louw, M., and Venkatakrishnan, V. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *SP'09: Proceedings of the 30th IEEE Symposium on Security and Privacy* (Oakland, California, USA, 2009).

[23] Van Gundy, M., and Chen, H. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-site Scripting Attacks. In *NDSS'09: Proceedings of the 16th Annual Network & Distributed System Security Symposium* (San Diego, California, USA, 2009).

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

[25] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H. The Multi-Principal OS Construction of the Gazelle Web Browser. In *SS'09: Proceedings of the 18th USENIX Security Symposium* (Montreal, Canada, 2009).