Finding Client-side Business Flow Tampering Vulnerabilities

I Luk Kim
kim1634@purdue.edu
Department of Computer Science, Purdue University
West Lafayette, Indiana, USA

Yunhui Zheng
zhengyu@us.ibm.com
IBM T. J. Watson Research Center
Yorktown Heights, USA

Hogun Park
hogun@purdue.edu
Department of Computer Science, Purdue University
West Lafayette, USA

Weihang Wang
weihangw@buffalo.edu
University at Buffalo, SUNY
Buffalo, USA

Wei You
youwei@ruc.edu.cn
Renmin University of China
Beijing, China

Yousra Aafer
yaafer@purdue.edu
Department of Computer Science, Purdue University
West Lafayette, USA

Xiangyu Zhang
xyzhang@cs.purdue.edu
Department of Computer Science, Purdue University
West Lafayette, USA

ABSTRACT

The sheer complexity of web applications leaves open a large attack surface of business logic. Particularly, in some scenarios, developers have to expose a portion of the logic to the client-side in order to coordinate multiple parties (e.g. merchants, client users, and third-party payment services) involved in a business process. However, such client-side code can be tampered with on the fly, leading to business logic perturbations and financial loss. Although developers become familiar with concepts that the client should never be trusted, given the size and the complexity of the client-side code that may be even incorporated from third parties, it is extremely challenging to understand and pinpoint the vulnerability. To this end, we investigate client-side business flow tampering vulnerabilities and develop a dynamic analysis based approach to automatically identifying such vulnerabilities. We evaluate our technique on 200 popular real-world websites. With negligible overhead, we have successfully identified 27 unique vulnerabilities on 23 websites, such as New York Times, HBO, and YouTube, where an adversary can interrupt business logic to bypass paywalls, disable adblocker detection, earn reward points illicitly, etc.

CCS CONCEPTS

• **Security and privacy → Web application security**.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380355

KEYWORDS

JavaScript; vulnerability detection; business flow tampering; dynamic analysis;

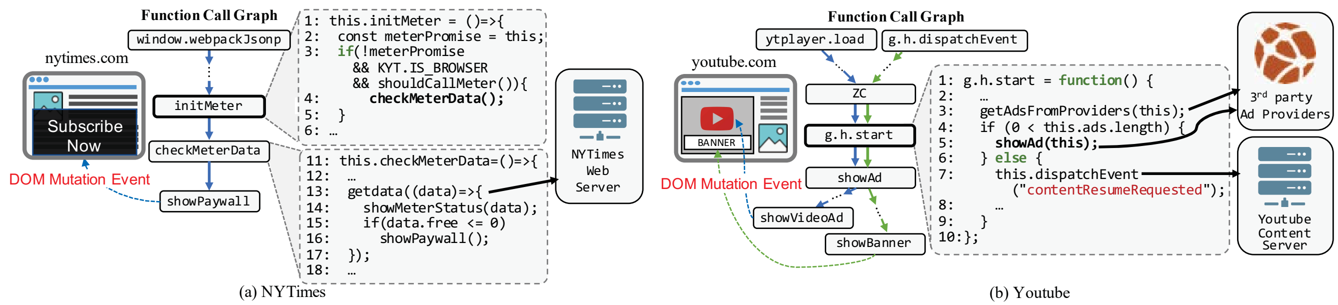
ACM Reference Format:
I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding Client-side Business Flow Tampering Vulnerabilities. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380355

1 INTRODUCTION

The intrinsic complexity of the web ecosystem has created an attractive attack surface for manipulation and exploitation. Adversaries have exploited many common flaws that plague various entities in the ecosystem. Of particular interest are client-side business logic flaws. If exploited, they may lead to devastating consequences.

As a side effect of exposing partial business logic to the client-side, by perturbing the internal control flow of events, adversaries are able to change the intended behavior of a website and cause various kinds of damages. For example, suppose an application's ad delivery mechanism is developed with the intention of playing a sponsor's video before streaming the actual content. Malice can directly skip the first step to circumvent the business model of the website. Similarly, a website rewards airline miles after a participant fills out a survey. An attacker can illegitimately earn miles without finishing the survey. A plausible approach to achieving this is to disable the condition check and force the execution of rewards logic, with the help of userscript manager utilities like Greasemonkey [13] or Tampermonkey [12].

Although OWASP strongly recommends enforcing business logic on the server-side [8], client-side implementations are commonly seen in practice. Sometimes, developers find it is easier to do so on the client-side without thinking too much about the consequences. But more importantly, it is unavoidable to devise some portion



(a) NYTimes (b) Youtube

Figure 1: Motivating examples

of the logic on the client-side to connect the dots in some scenarios. Web applications nowadays commonly integrate third-party content or services. In such cases, the client-side logic plays an important role in coordinating the internal states with those of multiple parties. For example, a web store may integrate a third-party payment service and implement the client-side logic to drive the checkout procedure [40]. For websites that serve a huge number of anonymous users (e.g., YouTube), it is very expensive to rely on the server-side to maintain the comprehensive states of all users.

Threat Model. We assume adversaries can manipulate client-side execution on the fly. For instance, they can trigger web page events in arbitrary orders. They can modify client-side scripts, change event handlers, bypass condition checks and alert, and send HTTP requests to servers. However, we assume they do not have access to servers so they cannot modify server-side logic.

Problem Statement. Web applications extensively incorporate code from multiple sources and thus it is desirable to understand the risks hidden in the client-side implementations. However, given the size and the complexities caused by JavaScript (JS) dynamic features and event-based executions, it is extremely hard to audit client-side scripts (including those from third parties) and identify the locations that are vulnerable to business flow tampering. To this end, we propose a dynamic analysis based approach to help developers focus on places that are more likely to be real vulnerabilities. By reporting the locations and the concrete tampering instances, our method can help developers effectively evaluate if actions should be taken to either relocate the logic to the server-side, deploy some runtime attack detection technique or incorporate additional client-side defense techniques.

In this paper, we investigate the pervasiveness of the *client-side DOM-related business flow tampering vulnerabilities*, To the best of our knowledge, this is the first study to characterize the impact of the client-side business flow tampering vulnerabilities. As they are commonly caused by insufficient process validation, we propose an automatic detection method that addresses the following challenges:

- Pinpointing places vulnerable to business flow manipulation is difficult in multi-functional web applications.
- Dynamic web application features should be handled as code can be injected and generated on the fly.
- Code modification techniques and event-based dynamic executions make static analysis difficult.

In particular, our method systematically examines websites as follows: (1) Starting with business operation descriptions, we navigate the website and collect a set of functions that may be relevant to the business logic. (2) We analyze each candidate function and look for potential tampering locations, which may perturb the intended behavior if modified. (3) We develop techniques to select functions that are more likely to be vulnerable and generate tampering proposals for each selected function. (4) We revisit the website with the tampering proposals and confirm if the detection results are indeed business flow tampering vulnerabilities.

To understand the scope and magnitude of the vulnerabilities in practice, we evaluate our method on 200 real-world websites. We are able to detect client-side business logic tampering vulnerabilities on popular websites. Specifically, attackers can bypass paywalls and read an unlimited number of articles without paying on NYTimes and WashingtonPost. Detected flaws on Youtube and CWTV enable attackers to skip in-stream video ads. We also discover a flaw in the popular reward-earning website InboxDollars; attackers can illegitimately earn rewards points without finishing the required steps (e.g. watch videos). In our experiments, we are able to stack $3.44 reward for an hour attack with a single machine without watching videos, and if we continue this attack, we could steal around $80 per day.

In summary, we make the following contributions:

- We investigate the pervasiveness of the DOM related client-side business flow tampering vulnerabilities.
- We develop a novel dynamic analysis based approach to automatically identifying client-side business flow tampering vulnerabilities.
- We evaluate our method on 200 popular real-world websites. With negligible page loading/rendering overhead, we found 27 unique vulnerabilities, where an adversary can interrupt the business logic to bypass paywall, disable adblock detection, earn rewards illicitly, etc.

2 MOTIVATION

In this section, we use two real-world examples to show (a) how business logic can be tampered with on the client-side, (b) why such vulnerabilities are common, and (c) why identifying these vulnerabilities is challenging.

2.1 Bypassing a Metered Paywall

New York Times [37] (NYT) is a well known news publisher. Its main business model is a metered paywall. It allows users to read a limited number of articles for free. After that, paid subscription is required. NYT developers implemented the metered paywall in March 2011. Within three months, the system generated 224,000 subscribers [29]. With the paywall, after a user reaches the quota of 5 free articles, a subscription message box with a black-colored background covering most of the screen will be displayed. Inspired by NYT's success, many publishers (e.g., Washington Post, The Boston Globe, and Chicago Tribune) adopted a similar paywall system. Fig.1(a) describes part of the simplified paywall implementation. Each time a news article is accessed, the article page is loaded as if there was no paywall. The paywall logic is implemented in a JS file loaded as part of the article page. In the JS file, function `window.webpackJsonp` invokes function `initMeter`, which further calls `checkMeterData` (line 4) that implements the paywall logic. In the function, the meter data is first accessed (line 13). If the current user exceeds the free quota, function `showPaywall` (line 16) inserts the subscription message box. To bypass the paywall, the attacker can disable the function call `checkMeterData()`. Consequently, the subscription message box is elided and the attacker can continue to access articles for free. A demo video of the attack (hosted on an anonymous website) can be found at [1].

While OWASP recommends that critical access control should be performed solely on the server-side to avoid any client-side tampering, NYT's design simply delivers all the content to the client and relies on client-side access control to protect the content. Further inspection suggests that there are reasons for such a flawed design. It looks like the paywall system was introduced as a well-encapsulated and stand-alone add-on (i.e., a self-contained JS file) to avoid any complex interference with the previous code-base. Properly implemented access control has to monitor each page load from the server-side, requiring substantial code changes. Furthermore, the number of free-readers is orders of magnitude larger than that of the subscribers. A correct design requires maintaining some profile for each free-reader on the server-side (e.g., the number of free articles accessed by the reader in a time duration), which would be much more expensive than the current design that only needs to maintain subscribers' profile. The current design relies on the client-side resources to deal with a large number of free readers. Such dilemmas are typical, leading to many flawed design and implementation as shown by our results in Section 5.

2.2 Skipping In-stream Ads

Ads revenue is crucial for business sustainability of streaming services like YouTube. Although service providers try to make money from other sources such as membership subscriptions, it turned out they often have to scale back and rely mostly on ads [11]. In particular, YouTube inserts ad videos before and in the middle of content videos. Recently, it even started showing Hollywood movies with ad breaks for free [34]. To implement this, YouTube has to use client-side logic because it needs to coordinate states among multiple parties and dynamically load videos from ad networks.

Fig. 1(b) shows a simplified version of the process. The ad videos are controlled by functions connected by blue arrows, while banner ads are managed by functions linked by green arrows. Function `ytplayer.load` is invoked during page load and eventually invokes function `g.h.start`. The function first gets available ads from third-party ad providers (line 3), then decides if ad videos should be played by checking if the list `this.ads` is empty (line 4). If yes, functions `showAd` and `showVideoAd` are called to play the in-stream video ads. Otherwise, it skips and plays the actual content (lines 7 – 8). Filling up *this.ads* is by a separate thread in the background, controlled by a timer. Similarly, function `g.h.dispatchEvent` is invoked regularly to deliver ad banners via function `showBanner`. By enforcing the false branch outcome at line 4, the ads are skipped and the user can watch the content video without watching the ads. Note that this is different from using an ad blocker to block ads. Many modern web applications are equipped with anti-adblocker mechanism, including Youtube [38]. Anti-adblockers often work by monitoring DOM object changes after ads are loaded. If no change is observed (meaning the ads are not displayed), it gets into a blocking mode requiring the user to turn off the adblocker. An anti-adblocker has to be closely coupled with the ad display function. In this case, the anti-adblocker is part of the function `shownAd()` (line 5). As such, by tampering with the JS code (i.e., line 4) directly, the ads, together with the anti-adblocker logic, are silently evaded.

A key feature of these business models is that the content publisher (or service provider) wants to ensure certain operations must be performed on the client-side, which cannot be trusted. This is not a new problem. There are many other rigorous business logics such as online shopping, credit card transactions, and online bank transactions. The key enabling technique in those business models is to associate a crypto-protected credential (e.g., token) with a user [5]. The token is shared by the multiple parties in the business model such that client-side operations can be remotely verified. For instance, an online shopping application has to interact with at least a remote payment service provider and a remote product provider. The payment is recorded by the payment service provider with the credential of the user. The product provider can *independently* check with the payment service provider to make sure the payment is in place before the product is sent. Such distributed integrity protection mechanism is heavyweight and often deployed in applications where users are properly profiled (e.g., users with accounts).

However, many web applications serve a vast number of users with most of them not properly profiled and hence do not have associated credentials. Nonetheless, the content publishers want their interest to be protected on those un-profiled users (by limiting their quota like in NYT or forcing them to watch ads like in YouTube). Such light-weight business models usually have to rely on client-side logic to conduct access control. In the YouTube example, ad networks are designed in such a way that any third party, including individuals, can bid for an ad slot (on YouTube). Most such third parties do not have the capacity to support remote authentication (like the payment service provider in online shopping), which entails saving credentials of individual users. While a better scheme may be possible for lightweight business models in the future (e.g., through some centralized service like Google DoubleClick), client-side tampering is a realistic and prevalent vulnerability. As shown in the above examples, such vulnerabilities can lead to financial

loss. If such attacks were launched at a large scale, websites may go out of business. Hence, it is in websites' best interest to identify such vulnerabilities so that they can take action to secure their interest, for example, by employing more expensive authentication schemes, deploying on-the-fly attack detection on the server-side, or even performing sophisticated client-side obfuscation.

Identifying Client-side Tampering Vulnerabilities Is Challenging. Client-side code usually comes from multiple parties. In addition, as suggested by the results in Table. 2, there are 8, 307 JS functions on average in a single page load. Due to the overwhelming size and complexities (i.e., JS dynamic features, code minimization, and obfuscation) of the client-side code, it is impractical for developers to manually locate the vulnerable points. Developing an automated tool to expose such problems is necessary.

Therefore, we propose a dynamic search-based approach that applies a set of pre-defined tampering operations on client-side JS code. These tampering operations include enforcing branch outcomes, skipping or repeating functions. To reduce the search space, we develop an analysis technique to identify JS code elements that are likely to be business logic related and focus on tampering those. In particular, we observe that client-side business operations are usually correlated to DOM mutations. Hence, we intercept such events and collect the corresponding *candidate* JS functions. To deal with the prominent dynamic features of JS code, our technique is dynamic, modifying the underlying JS engine to instrument the internal intermediate representations of JS code on-the-fly, instead of directly instrumenting JS source code. For the aforementioned Youtube ad banner case, our technique selects 159 code elements as potential tampering candidates from 8,191 functions. The vulnerability disabling ad banners is identified after 10 trials. We have reported such problem to developers along with the NYT case.

3 SYSTEM OVERVIEW

Fig. 2 provides an overview of our vulnerability detection procedure, which can be largely divided into three phases.

Site information collection. By recording and inspecting user interactions, we collect basic information about the targeted websites. In particular, we identify DOM objects that should be monitored for mutation events and generate browsing automation scripts that allow automatic website navigation. They will be used as inputs to the whole procedure. Details can be found in Sec. 4.1.

Identify potential JS code elements to tamper with. We analyze the website and generate candidates for tampering. In particular, we monitor DOM mutation events and collect the corresponding call stacks. By inspecting the functions on the stack, we identify candidate functions that may be business logic related. The candidates are further ranked based on the estimation how likely they are vulnerable to tampering attacks. For each candidate, we generate potential *tampering proposals* that include the tampering points and the corresponding tampering operations. We explain the components and algorithms in Sec. 4.2- 4.4.

Vulnerability scanning by tampering testing. We repeatedly run the websites according to the generated tampering proposals to filter out proposals that cannot lead to tampering attacks. In order to reduce manual efforts to confirm if the outcomes are real attacks, we develop automated techniques to group test results, based on DOM event tracking and clustering. Instead of examining all outcomes, testers only need to check one representative from each cluster. We produce a vulnerability report to explain the attack for each exploit. We explain the details in Sec. 4.5.

4 DESIGN

In this section, we describe each component in detail and reason about our design choices.

4.1 Website Information Collection

Our system requires two pieces of information about the target website to start the procedure: (a) identifiers of DOM objects that are related to business logic and (b) browsing automation scripts. They are collected automatically by recording testers' interactions with the targeted websites. In the YouTube example discussed in Sec. 2, testers can record browsing activities to automate the operations such as "play video". In the meantime, testers can also specify elements or regions on the webpage that might be related to the business logic by simply clicking a button provided by our tool. Our system automatically collects DOM selectors that identify the DOM objects involved. Note that our technique does not require good code coverage of the application. Any test case that triggers the business model is sufficient. Due to the essential role of business model, a typical use case would easily cover it. Furthermore, such manual efforts are one-time. Our tool records the user operations in an automatic script that can be repeatedly executed in the scanning phase. Hence, the manual efforts required are minimal.

4.2 Identifying Potential Business Logic Related Functions

As testers already specified their areas of interest on the web page, we intercept the mutation events on the DOM objects and collect the corresponding (asynchronous) call stack. Then, we consider the functions on the stack that are more likely related to business logic and give them high priorities. In particular, when mutation events such as attribute updates, node modifications, or child DOM tree changes happen, a hook function will be invoked to collect the function call trace containing the functions that directly/transitively trigger the changes. We exclude common JS libraries since they are unlikely tampering vulnerability candidates. We remove them using a whitelisting approach.

Note that we may observe different call stacks for the same mutation event. We hence construct a call tree, by merging the same functions in stack traces through a preprocessing step. Take the YouTube case in Fig. 1(b) as an example. The function "`showAd`" appears in two different traces. It has two different callees ("`showVideoAd`" and "`showBanner`") in the two traces. They are hence the two children of "`showAd`" in the call tree. *All nodes in the call tree of a relevant DOM mutation are considered candidates and given high priorities.*

4.3 Dynamic Page Data Collection

In order to overcome the challenges introduced by JS dynamic features, in this step, we collect runtime information about the candidate functions obtained in the previous step. In particular, we dynamically construct a *Business Control Flow Graph (BCFG)*

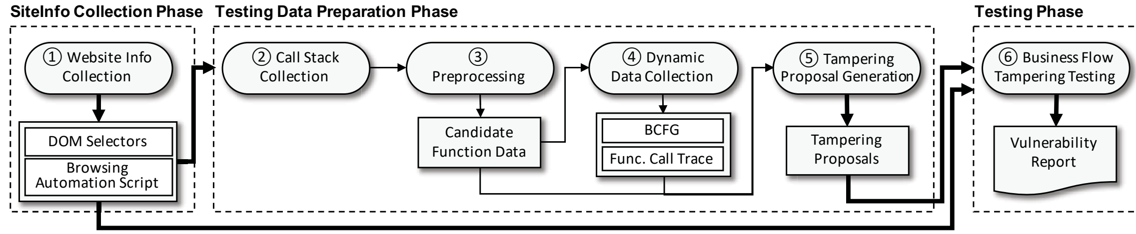


Figure 2: Approach overview

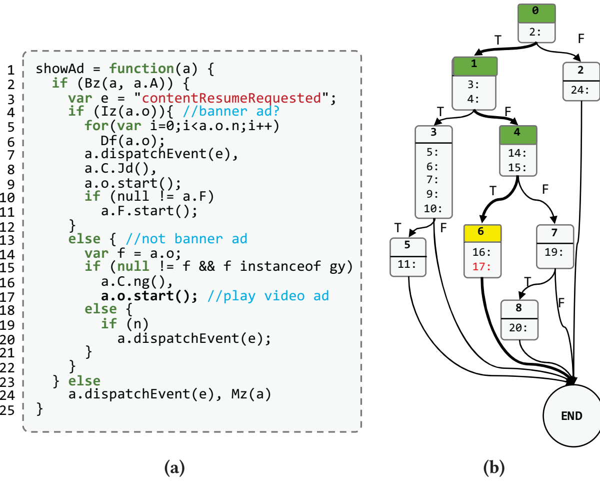


Figure 3: Source code and Business Control Flow Graph (BCFG) of function showAd **with each node representing a basic block with a unique id followed by the statements in the block**

for each candidate, which abstracts away path conditions that are unlikely to do with access control in business logic.

4.3.1 Business Control Flow Graph (BCFG). The abstraction focuses on precluding predicates that are not related to business logic access control. Specifically, loop predicates are abstracted away as we consider loop predicates are unlikely to perform access control. While abstracting away loop predicates, we retain the loop body which may contain important function calls. *Note that BCFG is not intended to be compiled and executed. It is more a representation for us to enumerate the possible tampering schemes (called tampering proposals).* After abstraction, BCFG mostly contains the following conditional statements: `if-then-else`, `switch-case`, and `conditional ternary operator`. A more heavy-weight analysis (e.g., one that leverages data-flow analysis) may have difficulty dealing with the various complex language features and the extremely dynamic nature of JS code, and hence is less desirable.

Example. Fig. 3a depicts a simplified version of the function showAd discussed in the YouTube motivating example. Function "showAd" renders ads differently based on the ad types (e.g., video ads or banner ads). At line 4, it checks the ad type (variable "`a.o`") and verifies if it's a banner ad. If so, the function resumes the player ("`a.dispatchEvent(e)`" at line 7) and displays the banner ad by invoking function "`a.o.start`" at line 9. Otherwise, the function plays the video ad by invoking function "`a.o.start()`" at line 17.

Table 1: Ten features for function ranking

ID	Features of Candidate Function `fn`
F1	Domain similarity between the website URL and `fn`'s script URL
F2	The loading order of the script containing `fn`
F3	The number of appearance of `fn` among all call stacks
F4	The position of `fn` on its call stack
F5	The collecting order of the call stack with `fn`
F6	The length of the call stack with `fn`
F7	The number of times `fn` is called
F8	The number of times `fn`'s callee is called
F9	The number of branches in `fn`
F10	`fn`'s callee directly mutates DOM (1: yes, 0: no)

Fig. 3b shows the BCFG of the function "`showAd`", where each box represents a block of instructions and each arrow denotes control flow. In particular, the execution path (BB_0, BB_1, BB_4 and BB_6) represents the video ads delivery procedure. The yellow-colored block BB_6 contains the function call linking to the next function on the call stack. Besides, BB_6 is control-dependant on the green-colored blocks (BB_0, BB_1 and BB_4). Observe that the loop predicate at line 5 is abstracted away.

In addition to stack traces and BCFG, we collect DOM mutation types, function execution frequencies, positions in source code, and the source URL. Such information is needed in the later scanning phase. Note that we convert the position information (of a code element) in the row and column format (in the source code) to its IR offset used inside the JS engine as tampering is performed by the modified engine. For example, the position of function call statement "`showAd(this)`" in Fig. 1 is "`row:5, column:9`". We convert it to "`offset:89`" with 89 the IR identifier of the statement.

4.4 Tampering Proposal Generation

With the call trees of DOM mutations and the BCFGs of the functions in the call trees, the next step is to generate a set of tampering proposals that specify the code location to tamper with and the tampering operation. Although these functions and predicates have a higher priority compared to others, due to the large search space, we develop additional techniques to further rank the functions and predicates. In particular, we first rank functions using a learning-based method. Then the BCFGs of ranked functions are traversed in order to derive tampering proposals.

4.4.1 Candidate Function Ranking. As we will show in Section 5, the number of functions in the call trees of DOM mutations is still very large. Ideally, we would like to develop a technique to determine which of these functions are more likely to contain business access control. However, a solely program analysis based solution may not have the desirable effectiveness as runtime information provides strong hints. For example, a business access control JS

file tends to be loaded before many other JSs; the function that performs access control often has high execution frequency than the content delivery function guarded by the access control; the URL of a business model related JS file tends to share common domain name as the main page, etc. Unfortunately, these properties are uncertain and their importance is difficult to determine by humans. Therefore we propose a learning-based method to predict the likelihood of a function containing business access control. We then rank the functions based on their likelihood.

Feature selection. Based on our observation of the properties that are possibly important, we select 10 features, as shown in Table 1. We use tampering locations we already know (in a small number of web applications) to evaluate the significance of the features. To refine selected features, ANOVA F-test [30] was leveraged. The null hypothesis of the test is that the feature takes the same value independently of the output value to predict. As a result, highly significant features were chosen for our classification task. All features are normalized individually by subtracting the mean and scaling to the unit variance.

Estimation of likelihood by learning a classifier. After selecting features, a classifier is learned from the training data. As our data may be biased, we explore using the Balanced Random Forest (BRF) [14], the weighted-SVM (w-SVM) [10], and the weighted-Logistic Regression (w-LR) [46], which are more interpretable than other classifiers (e.g., neural networks) and more robust when they have a small scale of training data. Balanced Random Forest (BRF) is an ensemble algorithm by a balanced bootstrapping method. In particular, we use 1,000 as the number of estimators. In addition, the gini impurity is used for split criterions. W-SVM (with RBF kernel) and w-LR use the ratio of class labels in their cost functions and put more weight on the rare cases to alleviate bias. In the testing stage, probabilities or regression values are used to estimate the likelihoods and are ranked by the scores.

To collect the training set for selecting features and learning the classifier, we first prepare a few confirmed tampering cases. After the training, we keep using the trained classifier to rank the candidate functions without any additional training. The result of the feature selection and learning will be discussed in Sec. 5.3.3.

4.4.2 Tampering Proposal Generation. Tampering proposals are generated by Algorithm 1. It takes two inputs: 1) the ranked candidate functions, where each function has its abstracted BCFG, the call site to its callees in the call tree, and the URL of the source code. 2) the *tampering strategy*, which can be either *bypass* or *repeat*. Intuitively, *bypass* skips a function call to see if the logic can be altered in the desired way, while the *repeat* strategy generates proposals that repeatedly invoke the callee.

The output is a list of tampering proposals indicating where and how the execution should be tampered with. In particular, a tampering proposal consists of (1) the URL of the script containing the candidate function, (2) the source code *offset* of the tampering point, (3) the *branch index*, and (4) the tampering *action*. The *branch index* specifies a branch outcome that should be enforced. For example, a basic block ended with an `if` statement may have two outgoing branches. If we want to execute the true branch next, we assign 0 to the branch index. Otherwise, we set the branch index to 1. Beside predicates, we may also tamper with the execution of

Algorithm 1 Tampering Proposal Generation

Input:
F: candidate functions sorted by the likelihood having tampering points. $f \in F$ is a function with the BCFG, the call site to its callee on stack, and the URL of the script having f.
ts: the tampering strategy, which can be $bypass$ or $repeat$
Output:
T : tampering proposal (script_URL, offset, branch index, action) $\in T$

```
1:  function GenerateTamperingProposals(F, ts)
2:      T ← []
3:      for each f ∈ F do
            // f.callsite is the call site in f to its callee on stack
4:          co ← GetOffset(f.callsite)
5:          if ts is bypass then
6:              T ← T ∪ (f.url, co, none, "disable callee")
7:              B ← GetControlDepBasicBlocks(f.callsite)
8:              for each b ∈ B do
                    // b.branch_cnt is the number of outgoing paths of basic block b
9:                  for i ← 0 to b.branch_cnt do
10:                     if HasPath(b, i, f.callsite.basic_block) then
                            // skip the existing path from b to the callsite
11:                         continue
                        // b.branching_stmt is the last stmt before branching in b.
12:                     bco ←GetOffset(b.branching_stmt)
                        // generate a non-existing path starting from b.branching_stmt
13:                     T ← T∪ (f.url, bco, i, "force branch outcome")
14:             fo ← GetOffset(f)
15:             T ← T∪ (f.url, fo, 0, "disable caller") // disable function f
16:         else
                // repeatedly invoke the callee of f on stack
17:             T ← T∪ (f.url, co, none, "repeat callee")
```

non-predicate statements. In this case, the proposal simply specifies the location of the statement without the branch index information. The *action* indicates how the execution should be tampered with at a particular tampering point. The action can be *disable callee*, *disable caller*, *force branch outcome*, or *repeat callee*. Details of each action can be found in Sec. 4.5.1.

For each candidate function f, we first locate the locations where f invokes its callees observed on the stack (line 4). For example, in Fig. 1(b), the invocation statement at line 5 is the call site where function `g.h.start` invokes its callee `showAd`. Although a candidate function may invoke multiple callees in the execution, we separate them and create a trace for each invocation. Therefore, in our representation, a candidate function only has one call site to its callee in a trace. Under the strategy *bypass*, we generate a proposal that skips the invocation of the callee function (line 6). Then, we obtain the basic blocks that the call site control-depends on (line 7), where each basic block returned has a number of outgoing branches (i.e. 2 branches for basic blocks with an `if` statement, and n branches for basic blocks ended with a `switch-case`).

Now we want to generate proposals that follow paths that are different from the one on the stack. To do so, we check if the call site is reachable through a particular path. Among them, we skip the path connecting the predecessor block and the call site (line 11). Since we want to explore the remaining paths even the path conditions are not met, we generate proposals for such paths (line 13) so that we can force the branch outcome.

The algorithm also creates a proposal to skip executing the entire function f (line 15). On the other hand, if the generation strategy is *repeat*, the algorithm creates a proposal that repeatedly invokes the callee function (line 17).

4.5 Business Flow Tampering Testing

After the tampering proposals are generated, we use a testing-based approach to confirm the real vulnerabilities. For each proposal, we leverage the automated script (recorded in the earlier phase) to load the target website. When the modified JS engine gets a script specified by the tampering proposal, it mutates the bytecode IR on-the-fly according to the action specified in the proposal. After a batch of tests, we gather test results and cluster them based on similarities. Finally, a tester confirms the success of the testing by checking the clustered results, which are usually just a few screen shots showing if the access control is circumvented. In this section, we first discuss how our system manipulates the business flow. Then, we describe how we filter out test results using DOM event tracking and clustering techniques in order to minimize manual efforts.

4.5.1 Tampering Actions. As mentioned before, there are four possible actions: *disable callee*, *disable caller*, *forced branching*, and *repeat callee*. Next, we explain how they are supported.

Disable callee. When the interpreter encounters the function call expression specified by the tampering location, it skips the call.

Disable caller. We disable the bytecode generation for statements in the function, which is equivalent to generating an empty function. This is because we still need the definition of the disabled function. Otherwise, the interpreter may crash if the disabled function is referred to somewhere. Disabling caller can be beneficial because it disables all function calls from other locations even not in the call stacks we collected. For example, in Fig. 1(b), if the website also plays the ad in the middle of playing the content, it can be turned off by disabling the execution of function `showAd`, instead of disabling all the function calls. Furthermore, callback functions triggered by native functions (e.g. event handler) or by external JS libraries can only be disabled by this method since call statements are not accessible.

Forced Branching. The branching target is forcefully set regardless of the result of the predicate condition. However, we still interpret the condition expression because it may have sub-operations (e.g. function calls).

Repeat callee. This tampering action is for duplicating desirable behavior (e.g. getting rewards) by repeatedly invoking the function. A naive approach to repeating callee would be to interpret the function call statement twice. However, business logic normally requires network interaction between the client and web servers. So, it is very likely the duplicated requests without interval will be ignored or considered as an error. We could add intervals by calling the `sleep` function at runtime in the JS engine. However, this may block the single-threaded JS engine and substantially interrupt the normal execution. To solve this problem, we use `setTimeout` function and register the function to be repeated as a callback function of the timer event.

4.5.2 Test Result Screening. After finishing each test trial, we need to check if the tampering proposal successfully alters the original business flow in an intended way. Since our approach relies on DOM changes, a simple solution is to track the existence of DOM mutation events. For the NYT example mentioned in Sec. 2, if the DOM mutation event that is triggered when displaying the subscription message box is reproduced in testing, this tampering proposal is not successful. However, even if the DOM mutation event is not triggered, it does not mean that this test trial succeeds for various reasons. For instance, the tampered execution may stop showing the subscription message box, but it also blocks other DOM objects, such as the article content. It is also possible that the event is not triggered because the page is crashed. A more complex scenario is that the text message disappears, but the black box still exists. Since there could be countless outcomes depending on web applications, a tester's intervention is inevitable to make the final decision. In order to minimize the manual efforts, we group test results using a similarity-based clustering technique. Instead of asking testers to check every result, they can just check one in each cluster. Such number is smaller according to our experiment in Section 5. Furthermore, for the rest of the testing batches, the tester only needs to check when a new cluster is found. To be specific, for each test trial, when the original DOM mutation event is not triggered, we take a screenshot, get the corresponding HTML source code, and store as a test result entry. After testing a batch of tampering proposals, we group the collected test results with a similarity-based clustering algorithm. We use the Structural Similarity Index Method (SSIM) [41] for screenshots, and Tree Edit Distance (TED) [45] algorithm for HTML files to compute the structural similarity between DOM trees. As a metric of the clustering algorithm, we combine the two similarity scores since they complement each other. Specifically, clustering with the image similarity metric usually generates fine-grained clusters especially if a screen changes frequently. In the ad banner case from the motivating example, screenshots might vary depending on the screenshot taking time since the video content is being played; therefore, there would be many clusters. In this case, DOM tree similarity metric would reduce the number of clusters if we combine them together. On the other hand, image similarity shows better performance if DOM structures change dynamically, such as a front page of newspaper websites loading dynamic contents. As a clustering algorithm, we select Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [18]. The advantage of DBSCAN is it does not require the number of clusters as an input unlike other algorithms such as k-means. It is an important factor since we do not have any clues about how many clusters exist in the test results.

Note that we do not need this step for the *repeat* tampering strategy, instead, we can simply check if the original DOM mutation event is triggered after the specified timeout.

5 EVALUATION

5.1 Implementation

Our system[1] is implemented in Python and Node.js, and the modified JS engine is based on V8 6.6.74. The testing module leverages the Puppeteer library[4], which provides high-level APIs to control Chromium over the DevTools Protocol[2]. To collect dynamic data and perform the business flow tampering testing, we instrument the target JS code by modifying V8 engine [6] in Chromium. In particular, the instrumentation works as follows: once the V8 engine loads

[1]We plan to make our system available at https://github.com/yirugi/JSFlowTamper

Table 2: Statistics of websites from 5 categories

Category	Total JS Size (KB)	# of JS	# of Functions	# of Branches
Newspapers	6,313	451	11,403	9,700
Magazine	4,334	258	8,046	6,884
Online Media	4,761	240	7,902	6,737
Surfer Rewards	2,521	132	3,931	3,330
Travel	4,402	220	7,031	5,854
Average	4,814	281	8,307	7,049

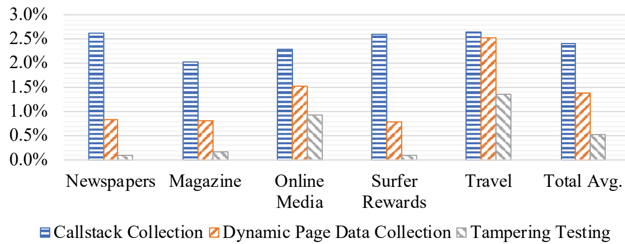


Figure 4: Normalized execution overhead

Table 3: Result of our testing on 200 websites

Case No.	Website	T.A.*	Vulnerable Operation	Case No.	Website	T.A.*	Vulnerable Operation
C-01	BostonGlobe	DCE	Paywall	C-14	CNBC	DCE	Anti-Adblock
C-02	NYTimes	DCE		C-15	CWTV	FE	
C-03	CWTV	DCE	Video Ad	C-16	CBS	FE	
C-04	FoxNews	DCE	Anti-Adblock	C-17	SeattleTImes	DCE	
C-05	NewsWeek	DCR	Offer Notification	C-18	MiamiHerald	DCE	
				C-19	DenverPost	DCE	
C-06	CBS	DCR	Video Ad	C-20	ETOnline	DCE	
C-07	Youtube	FE		C-21	AMC	DCE	
C-08	ETOnline	DCE		C-22	DallasNews	DCE	Paywall
C-09	AMC	FE		C-23	WashingtonPost	FE	
C-10	CartoonNetwork	FE		C-24	ChicagoTribune	DCE	
C-11	Fox	FE	Offer Notification	C-25	Youtube	DCE	Etc
C-12	PCMag	FE		C-26	HBO	FE	
C-13	Business-Standard	DCE		C-27	Inboxdollars	RC	

T.A: Tampering Action
*: FE = Forced Branching, DCE = Disable Callee, DCR = Disable Caller, RC = Repeat Callee

a script file, an Abstract Syntax Tree (AST) is built for each function and further translated to bytecode by the bytecode generator. We modify `InterpreterCompilationJob` class to generate BCFGs for JS functions after the ASTs are built. The `BytecodeGenerator` class is also modified to collect dynamic data and mutate execution. Comparing to code rewriting approaches [44], we modified the JS engine because it brings in additional benefits. First, it can easily handle dynamically generated codes as well as other sophisticated code modification techniques discussed in Sec. 2. Second, it has fewer side-effects. For example, it can work with code integrity checking techniques (e.g., Subresource Integrity (SRI) features [7]).

5.2 Research Questions

We investigate the following research questions in order to evaluate the effectiveness of our system:
RQ1. How much overhead does our system introduce?
RQ2. How many real-world business flow tampering vulnerabilities can our system find?
RQ3. How well do we estimate the likelihood of vulnerable functions? In particular, what are the results of the feature selection and the learning algorithm?
RQ4. How effective are tampering testing and result screening?
RQ5. How effective is our system on reducing search space?

To answer these research questions, we run our system on 200 real-world websites. The websites are collected from 5 different categories in Alexa Top 500 since they use the most common business models, such as advertisement, paywall, and point reward.

5.3 Experimental Methodology and Results

5.3.1 RQ1: Performance Overhead. Table 2 shows the benchmark statistics clustered by categories. On average, 8, 307 functions can be observed in a single page load, which points to the needs of our approach. We measure three kinds of overhead, the first one is to collect the stack trace of DOM mutation events, the second is caused by the instrumentation to collect dynamic page information such as function execution frequencies, and the third is the tampering testing overhead. To reduce non-determinism caused by dynamic page content (e.g., ads), we crawl the pages and resources to a local directory, and then load the local pages and resources with and without our technique. The former is consider the baseline. We run each of the 200 websites 10 times and average the execution time. Fig. 4 depicts the normalized overhead. The first 5 sets of bars show the overhead observed in each category and the last set denotes the average overhead. The overhead for the call stack collection step is 2.41% (90ms) on average. We observed the average number of DOM mutation events triggered during page loading is 60.55. Hence, the overhead of handling one mutation event is around 1.5ms. Similarly, the overhead for dynamic page data collection step is 1.39% (70ms). We do not measure the overhead of executing tampering proposals for all 200 websites as it causes exceptions and early termination in many cases, skewing the real overhead. From the cases that terminate normally, the average overhead is 0.53% (4ms) which is lower than dynamic page data collection.

5.3.2 RQ2: Effectiveness in Finding Vulnerability. Our technique discovers 27 vulnerable cases from 23 websites as shown in Table 3. The first and the fifth columns show the case number while the second and the sixth columns describe the website. The third and the seventh columns indicate the tampering action, and the last ones contain the vulnerable business logic. Observe many websites are mainstream content publishers. We use the first 5 cases to produce the training set for the function ranking model. After we trained the classifier, we found 22 more cases. Besides the NYT case in the motivation section, we found 4 more vulnerable paywall systems (C-01, 02, and 22 - 24). Instead of using paywall, some websites show an offer notification popup at the front page. We found 4 cases that the offer popup can be disabled (C-05, and 11 - 13). From the websites in online media category, most of the findings are about skipping the ads before or in the middle of video playing (C-03, and 06 - 10). We also try to tamper with the protection method for their ad-related business logic against adblockers. We found 9 cases (C-04, and 14 - 21) in which the anti-adblock techniques can be bypassed. Youtube shows ad banners in the middle of video playing, and this can be skipped by disabling callee (C-25). HBO prompts a user to provide personal information in order to watch free episodes right before video starts. This can be skipped by forced branching (C-26). Inboxdollars gives points to users at the end of watching a video. Our system found a way to repeat the rewarding operation using the repeat callee tampering action (C-27). We have uploaded

Table 4: Function ranking with classifiers

(a)

Case No.	Avg. Rank of Func. w/ Tampering Point		
	w-LR	w-SVM	BRF
C-01	3.5	13.2	2.1
C-02	26.1	16.2	13.4
C-03	4.4	5.7	6.3
C-04	1.4	5.2	1.8
C-05	2	1.5	1.1
Average	7.48	8.36	4.94

(b)

Case No.	Rank of Func. w/ Tampering Point		Case No.	Rank of Func. w/ Tampering Point	
	BRF	Random		BRF	Random
C-06	1	82.5	C-17	3	10.4
C-07	4	90.4	C-18	1	3.8
C-08	8	47.4	C-19	1	9.3
C-09	1	31.4	C-20	9	46
C-10	6	25.2	C-21	4	69.5
C-11	5	19.4	C-22	1	7.1
C-12	1	4.3	C-23	2	63.2
C-13	3	4.6	C-24	1	9.1
C-14	1	5.6	C-25	1	3.5
C-15	1	8.5	C-26	2	12.3
C-16	8	26.3	C-27	11	12.4
Average				3.41	26.92

Table 5: Function ranking and screening results

Case No.	# of T.P.	# of Tests to Success	# of Results after E.S.	# of Results after Clustering
C-01	264	170	32	4
C-02	191	150	107	17
C-03	176	90	42	16
C-04	150	20	13	4
C-05	208	10	8	2
Average	197.80	88.00	40.40	8.60
C-06	486	10	10	4
C-07	281	20	8	3
C-08	440	20	16	2
C-09	99	10	10	3
C-10	460	20	12	2
C-11	45	20	14	6
C-12	209	10	8	4
C-13	93	10	3	1
C-14	66	10	10	3
C-15	211	10	5	3
C-16	225	50	27	3
C-17	173	10	2	1
C-18	35	10	4	1
C-19	623	10	10	1
C-20	722	20	18	4
C-21	113	10	7	2
C-22	53	10	1	1
C-23	529	10	7	2
C-24	933	10	10	3
C-25	159	10	2	2
C-26	57	10	6	3
C-27	42	19	-	-
Average	275.18	14.50	9.05	2.57

T.P.: Tampering Proposal, E.S.: Event-based Screening

demos of our findings (recorded screens of the successful tampered cases) to a private website[2]. We only use these findings for research purpose. We have responsibly reported the vulnerabilities to the victim websites and are in communication with them for possible defense solutions.

5.3.3 RQ3: Feature selection and learning algorithm. We use candidate functions from the first 5 cases in Table 3 as the training set. For these cases, we test every tampering proposals. If a vulnerability is found, the candidate function containing the tampering proposal is marked as a positive sample. The others are marked as negative samples. At the end, we acquire 56 positive samples and 402 negative samples. Using the training set, we perform the feature selection process for the 10 features in Table 1. We conduct the ANOVA test to find out which features are significant. As a result, the first 5 features whose p-value is less than 0.1 are selected (F1, F4, F5, F7, and F8). Next, in order to check which learning algorithm works best for our scenario, we tested 3 classifiers discussed in Section 4.4.1 using the training set with the 5 websites. In particular, we learn each classifier on 3 randomly picked websites and test them on the rest of the websites for its cross-validation. In order to evaluate the learned models, we order candidate functions for each website based on the likelihood scores and got the rank of the first function containing the real vulnerability. We perform this evaluation test 10 times, then calculate the average rank values, and Table 4a describes the results of the 3 classifiers, as a result, BRF shows the best average rank value.

Table 4b shows the function ranks of the 22 successful cases we find after we apply the trained classifiers (BRF). In order to evaluate its efficacy, we also select functions 10 times randomly, then get the averaged rank of functions containing the real vulnerability. As we can see in the table, the rank values with the classifier show significantly better performance than the random method. In 10 of the 22 cases, we find the vulnerability at the first candidate function using the ranking.

5.3.4 RQ4: Effectiveness of Tampering Testing and Result Screening. Table 5 shows the effectiveness of tampering testing and test result screening. The second column shows the total number of tampering proposals. The third column describes the number of tests until we find a successful case. The last two columns show the effects of the test result screening, the number of test results after DOM event-based screening, and the number of results after similarity-based clustering. The last column also indicates that the number of results requiring a tester's confirmation. In this experiment, we test 10 tampering proposals in one batch. As we mentioned, the first 5 cases are collected from randomly picked candidate functions, and the rest 22 cases are found with the help of the candidate function ranking method. In the first 5 cases (C-01 to C-05), the numbers of tests vary from 10 to 170. The worst case is almost 80% of the total tampering proposals (C-02), and on average, we test around 50% of the tampering proposals. The clustering and screening significantly reduce manual efforts such that testers only need to check 8 results on average, in comparison to the hundreds proposal executions. In the 22 cases found later (C-06 to C-27), the number of tests needed to expose the real vulnerabilities is tremendously reduced using the function ranking method. The average is 14.50, which is only 5% of the total tampering proposals. Because of the reduction, testers only need to check 2.57 results on average. In addition, we investigate the cases that do not have vulnerabilities, and we observe that testers have to check 27.93 clusters on average. As checking a cluster is as simple as inspecting a screenshot, we consider such manual efforts are manageable.

5.3.5 RQ5: Effectiveness in Reducing Search Space. In order to reduce search space, we collect call stacks by observing DOM mutation events, and we remove redundant functions and those from JS libraries. Moreover, the functions have zero call count during dynamic data collection are also removed in the tampering proposal generation. To evaluate the effectiveness of our filtering method, we collect statistics for the 27 successful cases. Specifically, we collect the total number of 3 types of data (JS files, functions, and branches)

[2]https://sites.google.com/view/tampering-cases/

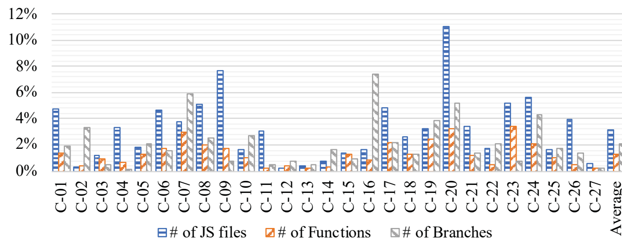


Figure 5: Effectiveness in reducing search space

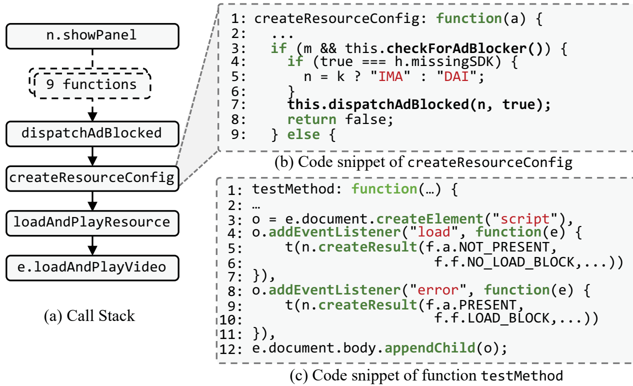


Figure 6: Bypassing adblock detection in cbs.com

we have to consider before and after filtering. Fig. 5 shows the normalized numbers of each data types from 27 cases, and the last bar denotes the average number. As we can see, we could reduce the number of each data type substantially. For instance, we need to investigate 18,658 functions without filtering if we want to find the case C-22. However, after filtering, there are only 84 functions left, and this is only 0.45% of the original number. On average, we only need to inspect 3.18% of the JS files, 1.31% of the functions, or 2.13% of the branches of those in the original execution.

5.4 Case Study

In this section, we show two case studies to demonstrate how our system finds the business flow tampering vulnerabilities.

5.4.1 Bypassing Adblock Detection. The website we use in this case study (C-16) is `cbs.com` which is one of the biggest television networks in the US. They provide the subscription-based online streaming service, and some of their episodes can be watched for free with video ads from sponsors. They also protect their business logic using the anti-adblock technique. If a user tries to watch a free episode with an adblocker-enabled web browser, the website blocks the actual contents with a warning message.

In order to find if their anti-adblocker technique can be bypassed using our system, we collect call stacks by tracking the warning message. After preprocessing, we have 95 functions, which are only 0.86% of the total functions on the page. There are 225 total tampering proposals, and after 5 batches, which is 50 trials, we found the vulnerability. Note that after the screening, only 3 test results required a manual check.

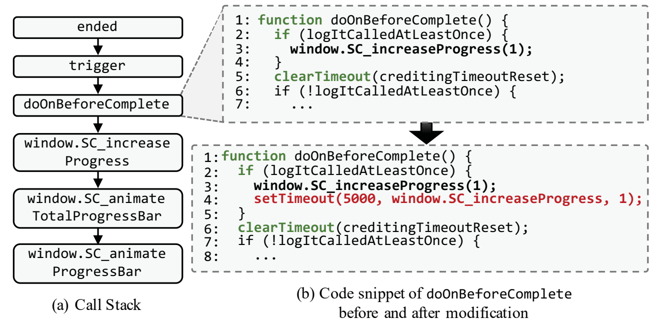


Figure 7: Repeating point reward in inboxdollars.com

To analyze how our system found the success case, we checked the vulnerability report. Our system found the tampering location in function `createResourceConfig`. Its call stack and code snippet are described in Fig. 6(a) and (b). It checks the presence of adblocker using `checkForAdBlocker` (line 3), and it calls `dispatchAdBlocked` to show the warning message (line 7). If we follow the function `checkForAdBlocker`, the function `testMethod` in Fig. 6(c) tries to inject a script containing "*ad*" string in its url (line 12) since the adblock applications usually block those scripts. The tampering proposal that forces the false branch of the if statement at line 3 succeeds. As shown, our system successfully found a way to bypass the anti-adblocker technique. With the tampered business flow, users can watch free episodes without watching video ads, and this would affect the business model of the website.

5.4.2 Repeating Point Reward. Inboxdollars (C-27) is an online marketing company that connects consumers and advertisers, and consumers can earn cash rewards for engaging in a variety of web activities. According to their website, total cash paid to members surpasses $50 million in 2016[3]. One of the services they provide is video reward; they offer points after a user watches a video containing ad. To be specific, when a video player reaches the end of the video, it increases a progress bar indicating the current reward status. In this case, we tried to repeat the rewarding activity. In order to start testing, we first recorded browsing interactions for logging in and clicking a play button, then gathered a DOM identifier by selecting the progress bar. The total number of functions that appeared during testing is 12,642, and we could reduce it to 27 which is only 0.21% of the total number. Our system successfully found the tampering location with 19 trials out of 42 tampering proposals. As we discussed in Section 4.5.2, it did not require the manual work to check the success case.

Fig. 7 illustrates the call stack and the code snippet of the function `doOnBeforeComplete` that has vulnerability. Specifically, when the video is finished, the player calls the function `ended`, then the function `trigger` calls the function `doOnBeforeComplete`. In the function, it first checks if the video has finished (line 2). If so, it calls the function `SC_increaseProgress(1)` to send a reward request to its server as well as to increase the progress bar. In order to repeat the call, the modified JS eingne added the setTimeout function containing the name of the function and a parameter, indicating the function will be called after 5,000 msec. Using this vulnerability, we could get multiple rewards after watching a single video. As

mentioned, the rewarded points can be exchanged to actual cash. This directly causes financial damage to the website. We were able to stack $3.44 reward points for an hour attack with a single machine, and if we continue this attack, we would get around $80 per day. We did not exchange the points we got from the vulnerability, and we are in communication with Inboxdollars so that they can deploy defense mechanism.

6 THREATS TO VALIDITY

There are a number of threats to the validity of our conclusion. Part of our technique (i.e, function ranking) requires training. Although our training task is quite simple with well defined features, we only use 458 training samples. While the training set and the test set are strictly separated and our results indicate the effectiveness of the trained model, it may be possible that the training set is not representative and hence the ranking model may not be optimal. We will study the effect of including more cases in the training set in our future work. Our results are only acquired on 200 top-ranked websites as our technique is heavyweight testing-based, requiring processing thousands of dynamically loaded JS files and substantial dynamic contents. It is possible that these 200 websites are not representative. We plan to test on more websites. Checking the final results requires human efforts. It is possible that we may miss some real vulnerabilities. We currently only support simple tampering operations, which may not disclose complex business model flaws.

7 RELATED WORK

Our work builds on extensive previous work on automatically testing web applications for vulnerabilities. We briefly describe relevant approaches, as well as previous works that detect business logic vulnerabilities in web applications.

Multiple path execution. Our work shares some similarity with recent work to explore execution paths by forcing program execution on JS programs [21], native binary programs [28], mobile apps [16, 20], and kernel rootkits [42]. Forced execution was first proposed in [42], which brute-forces control-flow at branches to explore program paths. X-Force [28] moves forward by designing a crash-free engine. In our work, forcing branch outcome is one of the tampering actions. However, our technique addresses a much broader problem. Guided mutation testing for JS web applications develops generic mutation testing approaches based on common mistakes made by JS programmers [23, 24]. Our technique mutates places specific to business models. It features sophisticated methods to narrow down the candidates of such mutation. Symbolic and concolic execution based techniques [22, 31–33] have also been proposed to analyze JS programs. Despite their great potential, handling substantial dynamic features in complex websites remains a challenge.

Business logic vulnerability detectors. Recently, researchers have proposed a number of techniques to test web applications for business logic vulnerabilities [15, 17, 19, 27, 35, 36, 39, 40]. These techniques focus mostly on the detection of web-based single sign-on systems and third-party payment systems. Wang et al. are the first to analyze logic vulnerabilities on merchant websites [40], and [39] studied logic flaws on popular web single sign-on systems. These techniques follow an API-oriented methodology that dissects the workflow in a particular application by examining how individual parties affect the arguments of related API calls. Sun et al. proposes a static detection of logic vulnerabilities in e-commerce web applications by combining symbolic execution and taint analysis to detect invariant violations of correct payment logic [36]. [15] proposes an approach to invoking static verification of the safety property of multiparty online services.

Dynamic JS code analysis. Dynamic analysis is commonly used to deal with the highly dynamic nature of JS applications. [25] builds a call graph on client-side codes embedded in server-side codes as string literals. It handles all possible client-side JS code variation by symbolically executing server-side code. AjaxRacer [9] detects AJAX event race errors in JS web applications by testing pairs of user events that are potentially AJAX conflicting. [43] proposes a dynamic slicer providing a comprehensive analysis to identify data, control, and DOM dependencies for client-side JS code. ConfictJS [26] finds conflicts between JS libraries by identifying potentially conflicting libraries and testing them with generated client applications that may suffer from the corresponding conflicts.

8 CONCLUSION

We present a novel dynamic analysis approach that scans web applications and automatically detects client-side business flow tampering vulnerabilities. We overcome technical challenges that hinder the detection process to make the technique practical. Our evaluation shows that our method has small overhead, and discovers 27 unique vulnerabilities in popular websites, which allow an adversary to interrupt the business logic of web applications such as skipping ads at the beginning of videos, bypassing ad blocking checking and even illicitly earning reward points.

In the future, we will further investigate our approach by calculating precision and recall metrics for known vulnerabilities so that we can show more accurate coverage evaluation results for possible missing cases.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by DARPA FA8650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] [n.d.]. Business Flow Tampering Success Cases. https://sites.google.com/view/tampering-cases.
[2] [n.d.]. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.
[3] [n.d.]. InboxDollars - about us. http://corporate.inboxdollars.com/about-us/company/.
[4] [n.d.]. Puppeteer. https://pptr.dev/.
[5] [n.d.]. Security Token. https://en.wikipedia.org/wiki/Security_token.
[6] [n.d.]. V8 JavaScript engine. https://v8.dev/.
[7] 2016. W3C Recommendation - Subresource Integrity. https://www.w3.org/TR/SRI/.
[8] 2019. OWASP AJAX Security Cheat Sheet. https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/AJAX_Security_Cheat_Sheet.md#dont-rely-on-client-business-logic.

[9] Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX race detection for JavaScript web applications. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 38–48.

[10] Rehan Akbani, Stephen Kwek, and Nathalie Japkowicz. 2004. Applying support vector machines to imbalanced datasets. In *European conference on machine learning*. Springer, 39–50.

[11] Julia Alexander. 2018. YouTube Premium is changing because it has to. https://www.theverge.com/2018/11/29/18116154/youtube-premium-free-ads-subscription-red.

[12] Jan Biniok. 2019. Tampermonkey project homepage. https://tampermonkey.net/.

[13] Aaron Boodman. 2019. Greasemonkey project homepage. https://www.greasespot.net/.

[14] Chao Chen, Andy Liaw, and Leo Breiman. 2004. Using random forest to learn imbalanced data. *Tech. Rep* (2004), 1–12.

[15] Eric Y Chen, Shuo Chen, Shaz Qadeer, and Rui Wang. 2015. Securing multiparty online services via certification of symbolic transactions. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 833–849.

[16] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 44–56.

[17] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. 2011. Fear the EAR: discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 251–262.

[18] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[19] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2014. An expressive model for the Web infrastructure: Definition and application to the Browser ID SSO system. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 673–688.

[20] Ryan Johnson and Angelos Stavrou. 2013. Forced-path execution for android applications on x86 platforms. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 188–197.

[21] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 897–906.

[22] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking internet malware. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 443–457.

[23] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. Efficient JavaScript mutation testing. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 74–83.

[24] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering* 41, 5 (2015), 429–444.

[25] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 518–529.

[26] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 741–751.

[27] Giancarlo Pellegrino and Davide Balzarotti. 2014. Toward Black-Box Detection of Logic Flaws in Web Applications.. In *NDSS*.

[28] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 829–844. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng

[29] Peter Preston. 2011. A paywall that pays? Only in America. https://www.theguardian.com/media/2011/aug/07/paywall-that-pays-only-in-america.

[30] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. 2007. A review of feature selection techniques in bioinformatics. *bioinformatics* 23, 19 (2007), 2507–2517.

[31] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 513–528.

[32] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 488–498.

[33] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 842–853.

[34] Garett Sloane. 2018. YouTube is now showing ad-supported Hollywood movies. https://adage.com/article/digital/youtube-starts-showing-free-hollywood-movies-ad-breaks/315631/.

[35] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, Luca Compagna, et al. 2016. Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications.. In *NDSS*.

[36] Fangqi Sun, Liang Xu, and Zhendong Su. 2014. Detecting Logic Vulnerabilities in E-commerce Applications.. In *NDSS*.

[37] The New York Times. 2019. Breaking News, World News & Multimedia. https://www.nytimes.com/.

[38] Jordan Valinsky. 2016. Some Adblock Plus users are reporting problems with YouTube. https://digiday.com/social/youtube-adblock-problems/.

[39] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 365–379.

[40] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. 2011. How to Shop for Free Online–Security Analysis of Cashier-as-a-Service Based Web Stores. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 465–480.

[41] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[42] Jeffrey Wilhelm and Tzi-cker Chiueh. 2007. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Christopher Krügel, Richard Lippmann, and Andrew J. Clark (Eds.), Vol. 4637. Springer, 219–235. https://doi.org/10.1007/978-3-540-74320-0_12

[43] Jiabin Ye, Cheng Zhang, Lei Ma, Haibo Yu, and Jianjun Zhao. 2016. Efficient and precise dynamic slicing for client-side javascript programs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 449–459.

[44] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 237–249.

[45] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.

[46] Zhaohui Zheng, Xiaoyun Wu, and Rohini Srihari. 2004. Feature selection for text categorization on imbalanced data. *ACM SIGKDD Explorations Newsletter* 6, 1 (2004), 80–89.