ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities

PRESENTED BY SAI TEJ KANCHARLA

Content

- Introduction
- Motivation And Threat Model
- System Overview
- Invariant Detection
- Invariant Enforcement
- Evaluation
- Related Work & Conclusion

Introduction

- Modern Web Apps are increasingly using JavaScript to move program code to client side.
- With increase in use of HTML5 API's such as *postMessage* client side Validation vulnerabilities are becoming increasingly important to address.
- But most detection and prevention techniques focus on sever side and less on client side.
- Hence there is a need for a system on client side which can protect against these vulnerabilities.

Threat Model

- We consider a webmail service that contains code and resources of both the application and ads from multiple origins.
- The webmail communicates with the ad networks via *postMessage* to get ads for target profiles.
- Since origins of ads are distinct Same Origin Policy applies, so these ads cant communicate with each other.
- Since the ad component uses *onMessage* and *postMessage* to send and receive messages from webmail component and responses, it is vulnerable to client side validation attacks.

Threat Model

- To tamper with the ad network, the attacker must be able to invoke *postMessage* in the same context.
- This can be achieved by using *Cross Site Scripting(XSS)* vulnerabilities from user content, framing the webmail service or exploiting logic vulnerability.
- *Cross-Site Scripting (XSS)* attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. **XSS attacks** occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.
- Hence the attacker has to send an email to victim user that contains the XSS code or lure the victim into a site that will frame the webmail service.

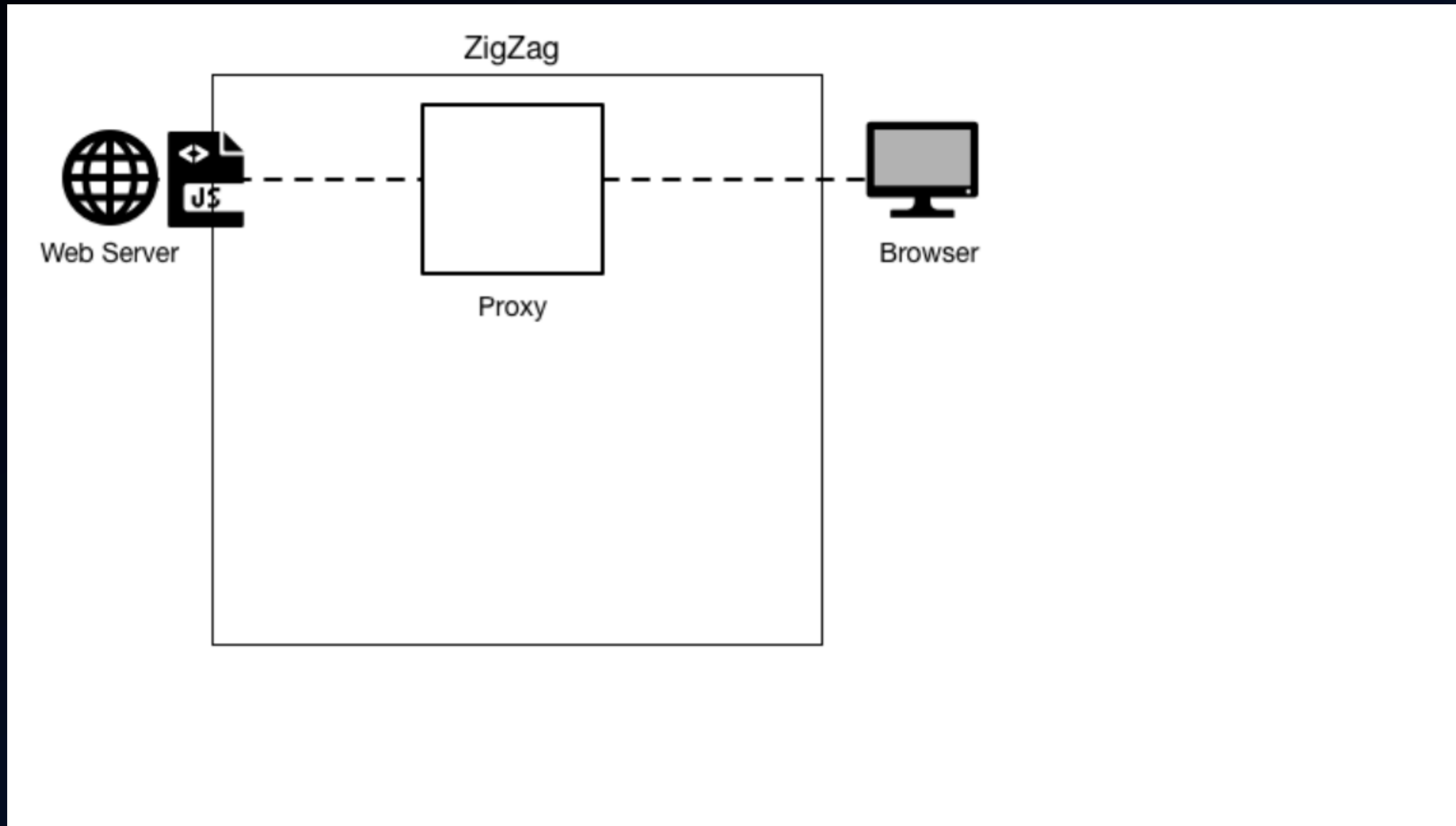
System Overview

- ZigZag is an in-browser anomaly detection system that defends against Client Side Validation attacks in JavaScript Applications.
- It works by interposing between web servers and the browser in order to transparently instrument JavaScript programs.
- The instrumentation process works in two phases:
- Learning Phase
- Enforcement Phase

Learning Phase

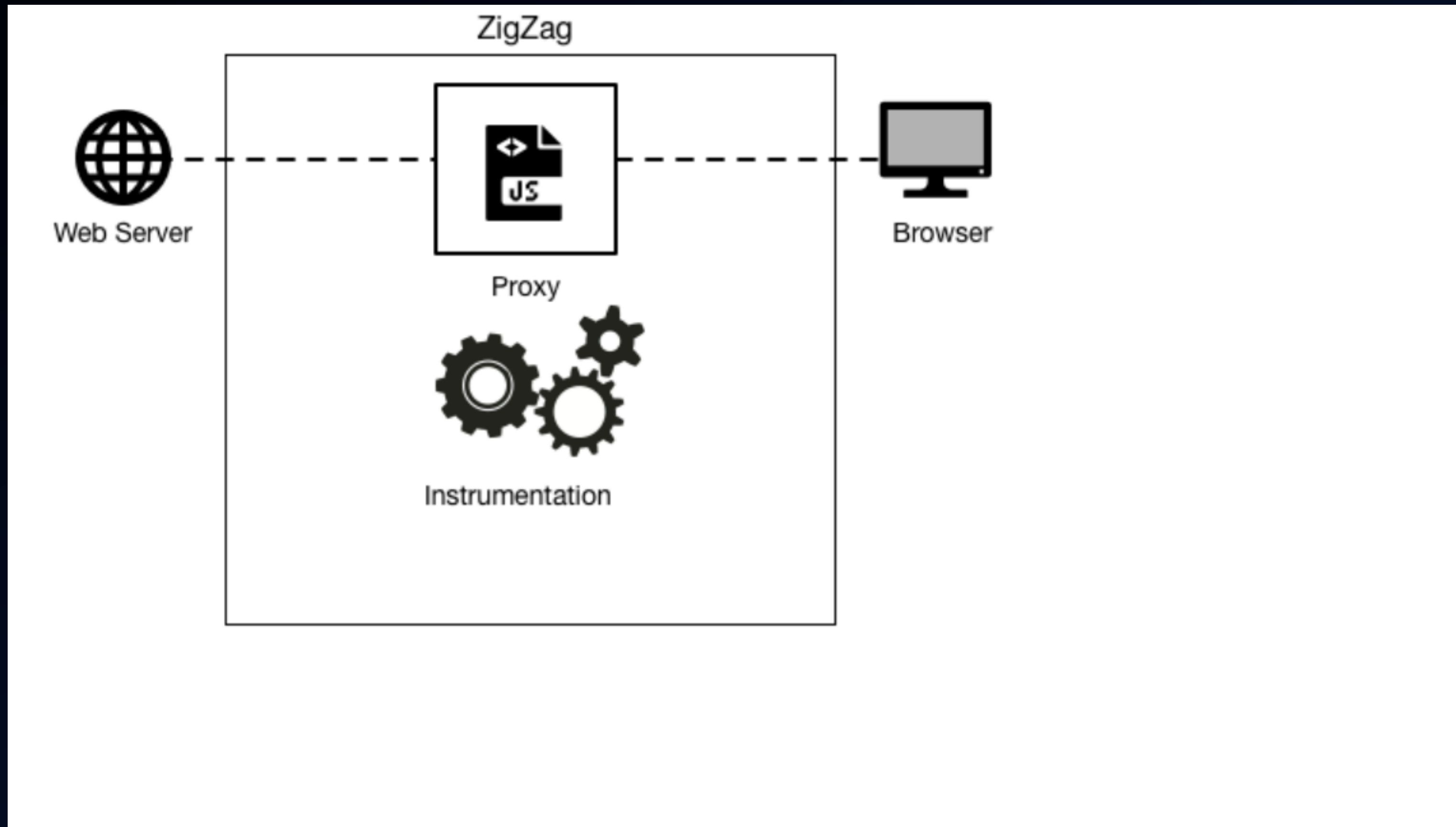
- Zigzag rewrites program with monitoring code to collect execution traces of client side code.
- The traces are fed to Dynamic Invariant Detector that extracts likely invariants or models.
- **Invariant** is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution
- The invariants extracted are learned over data such as function parameters, variable types and function caller.
- However, there is no guarantee that invariants will also hold in the future. Therefore, ZigZag only uses invariants which should hold with a high probability

Learning Phase



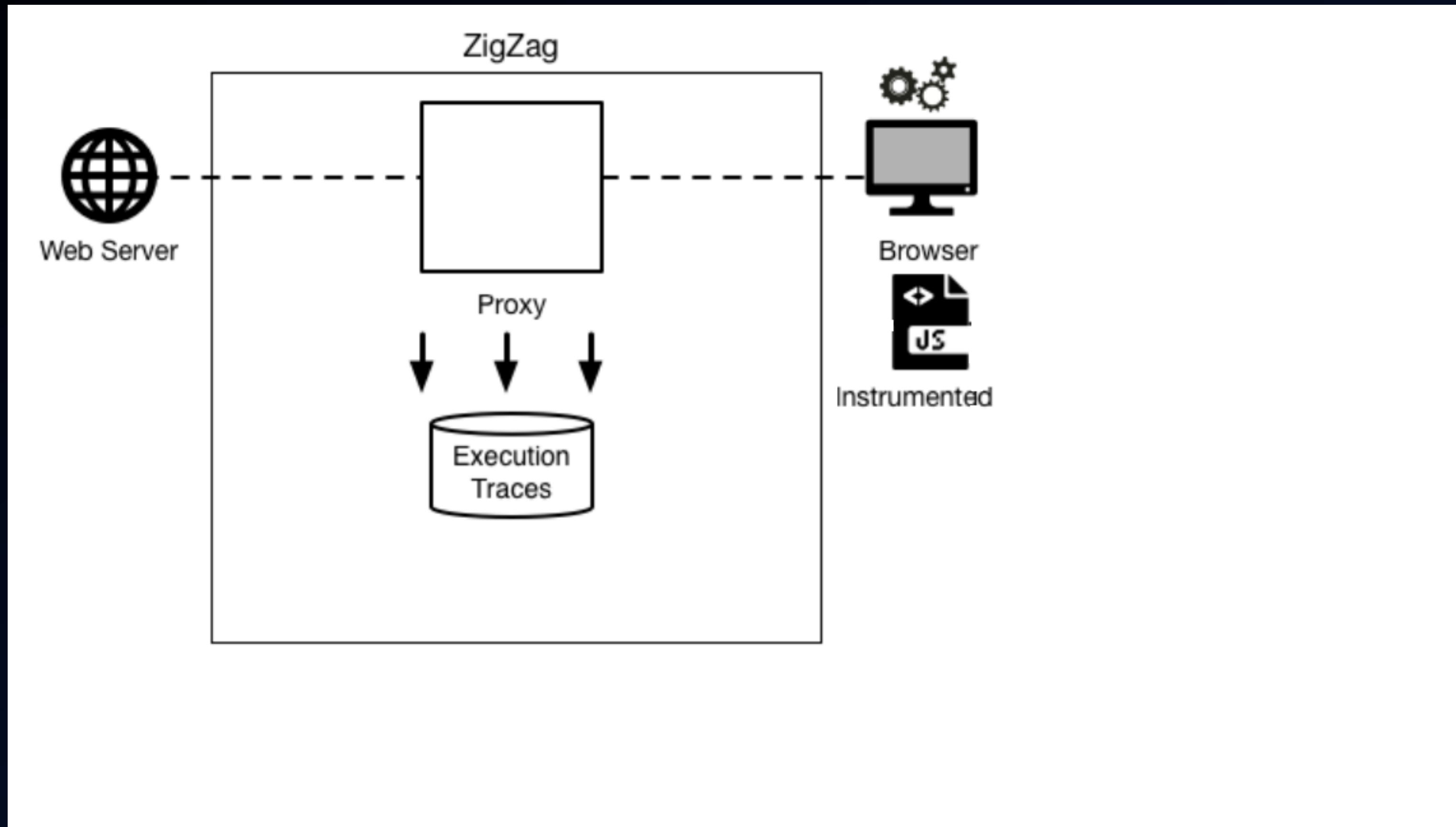
With ZigZag, the webmail service is used through a transparent proxy to monitor the code.

Learning Phase



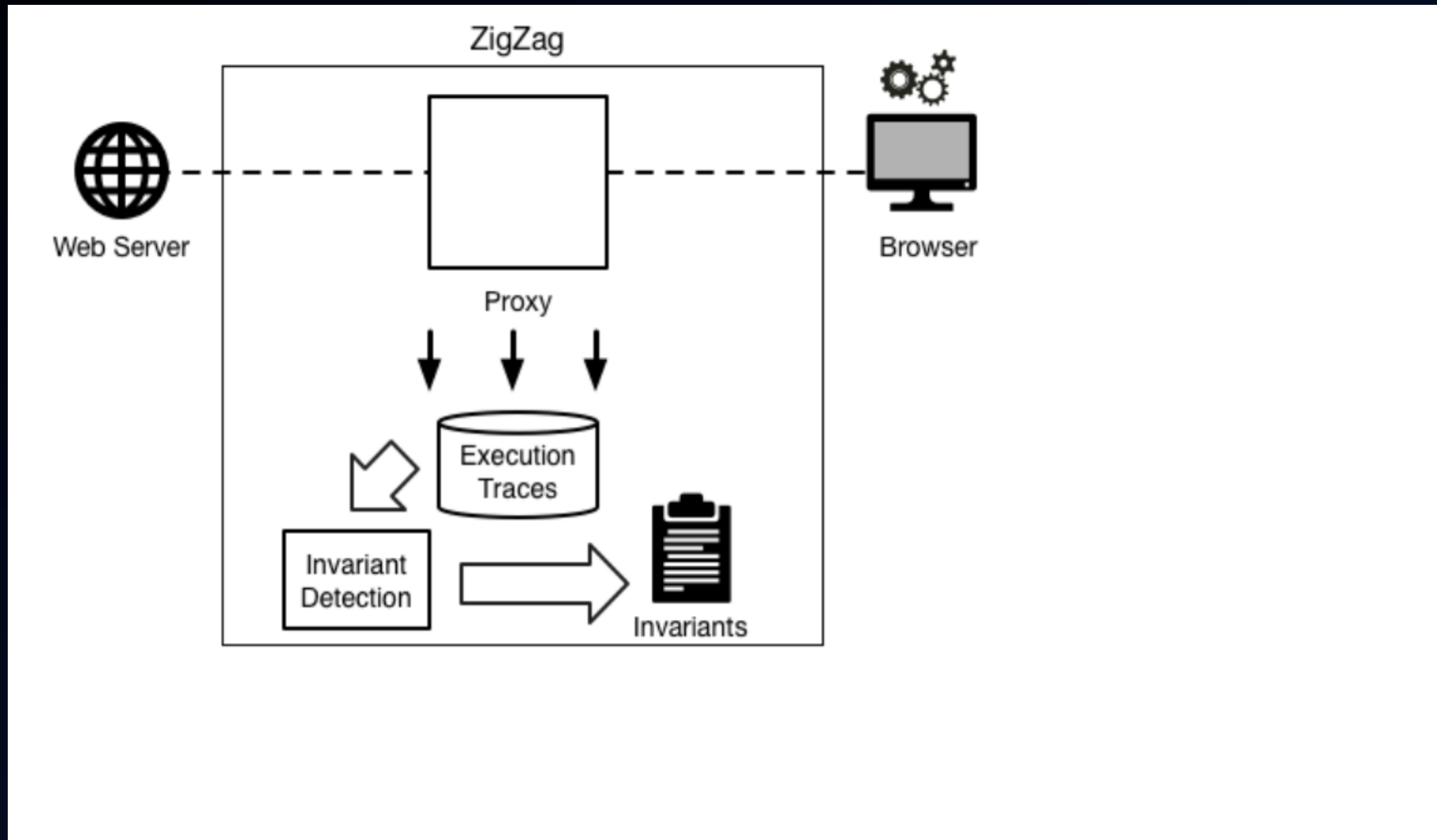
ZigZag uses a transparent proxy that instruments the JavaScript code, augmenting each component with monitoring code

Learning Phase



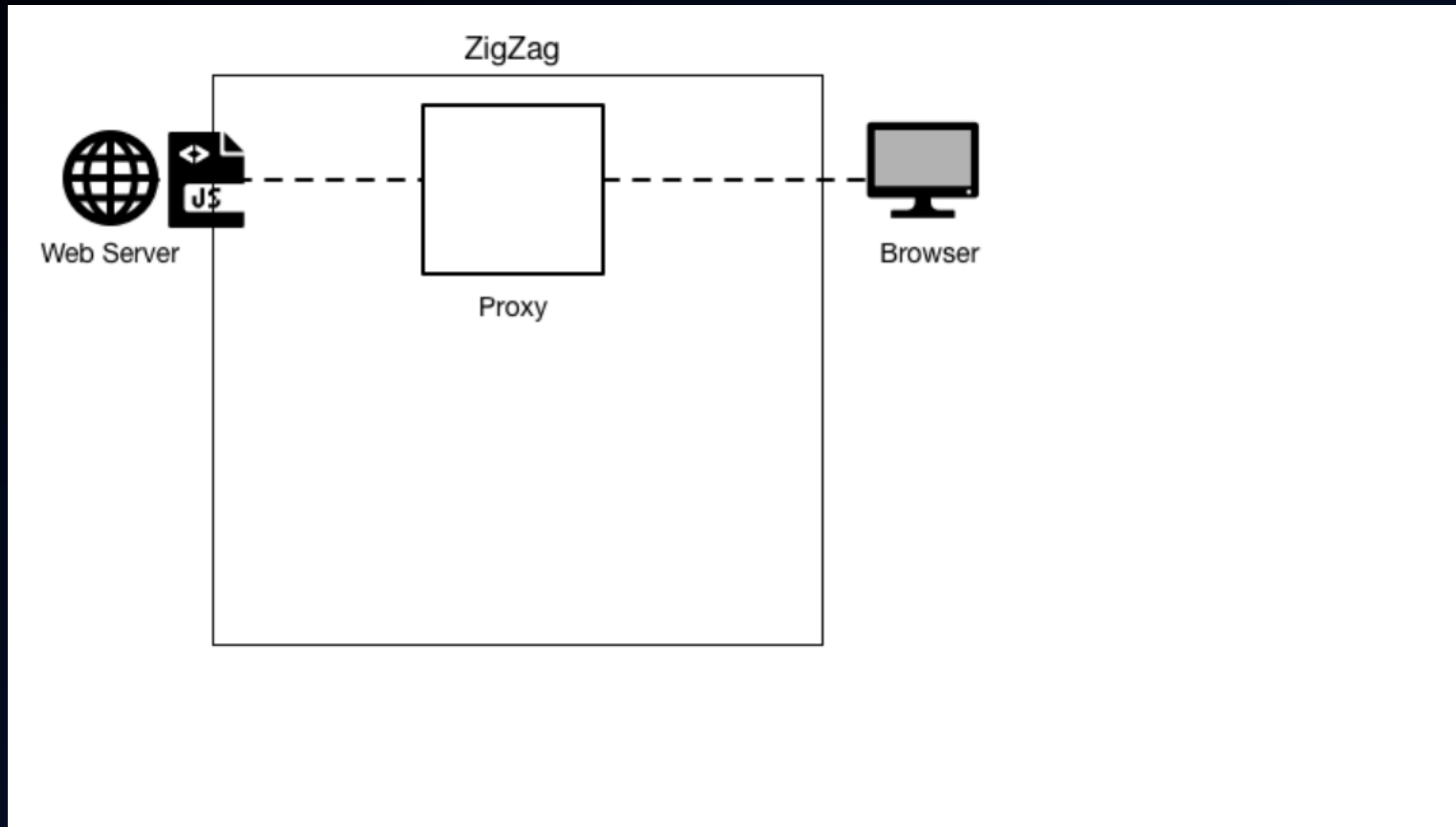
The webmail service then runs in a training phase where execution traces of the JavaScript programs are collected.

Learning Phase



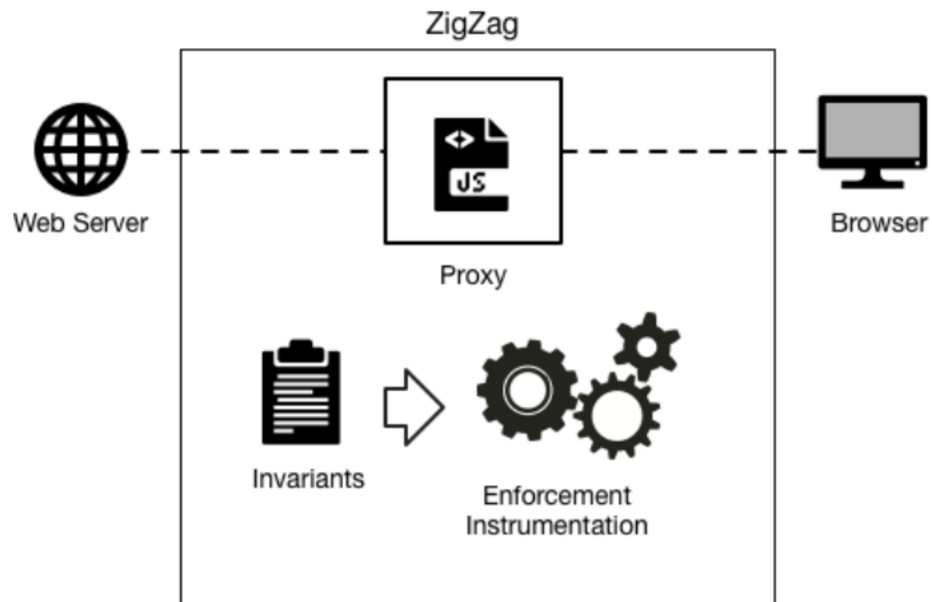
Once enough execution traces have been collected, ZigZag uses invariant detection to establish a model of normal behavior.

Enforcement Phase



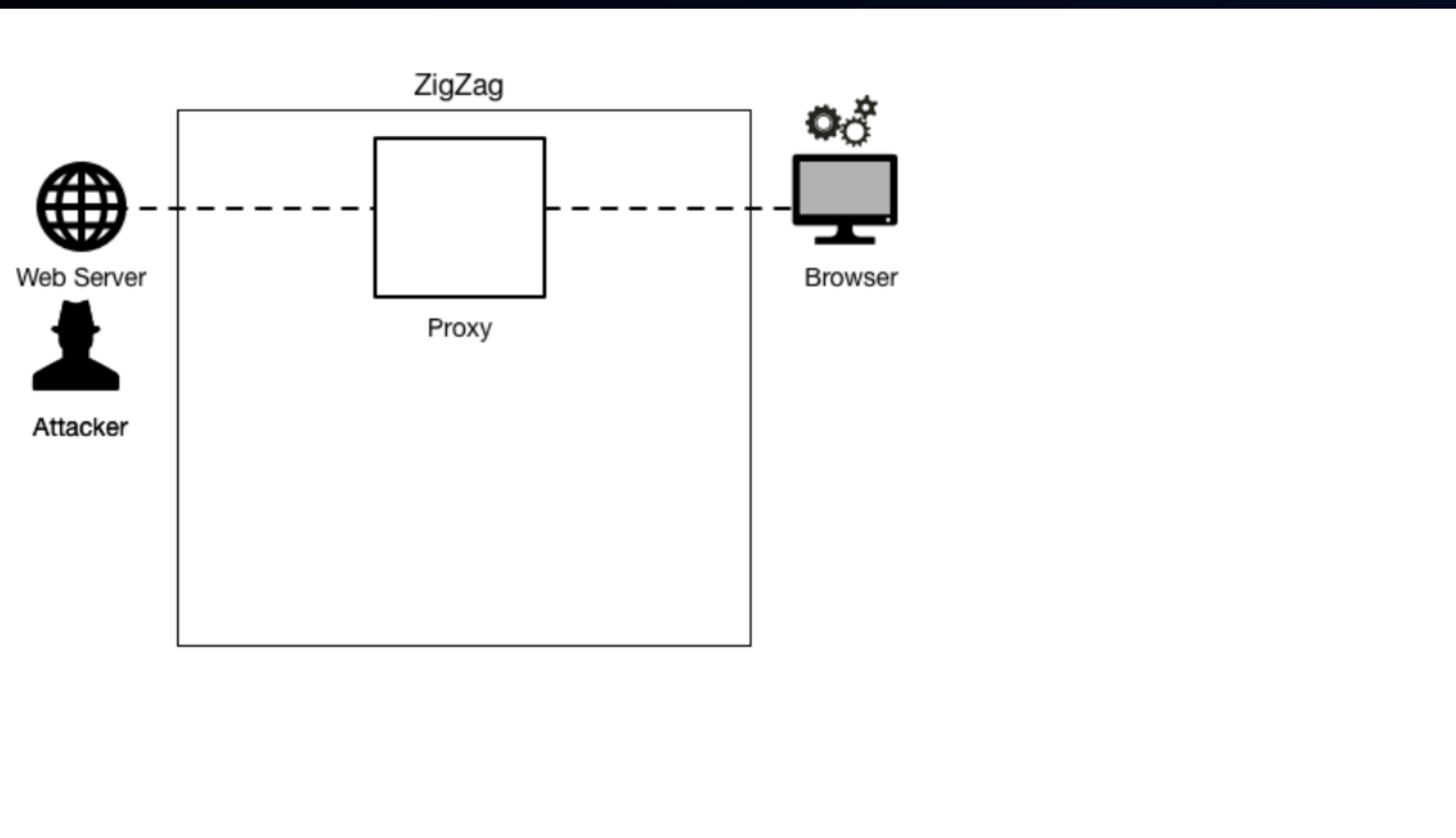
In the enforcement phase the invariants learned in the initial phase are used to harden the client side components

Enforcement Phase



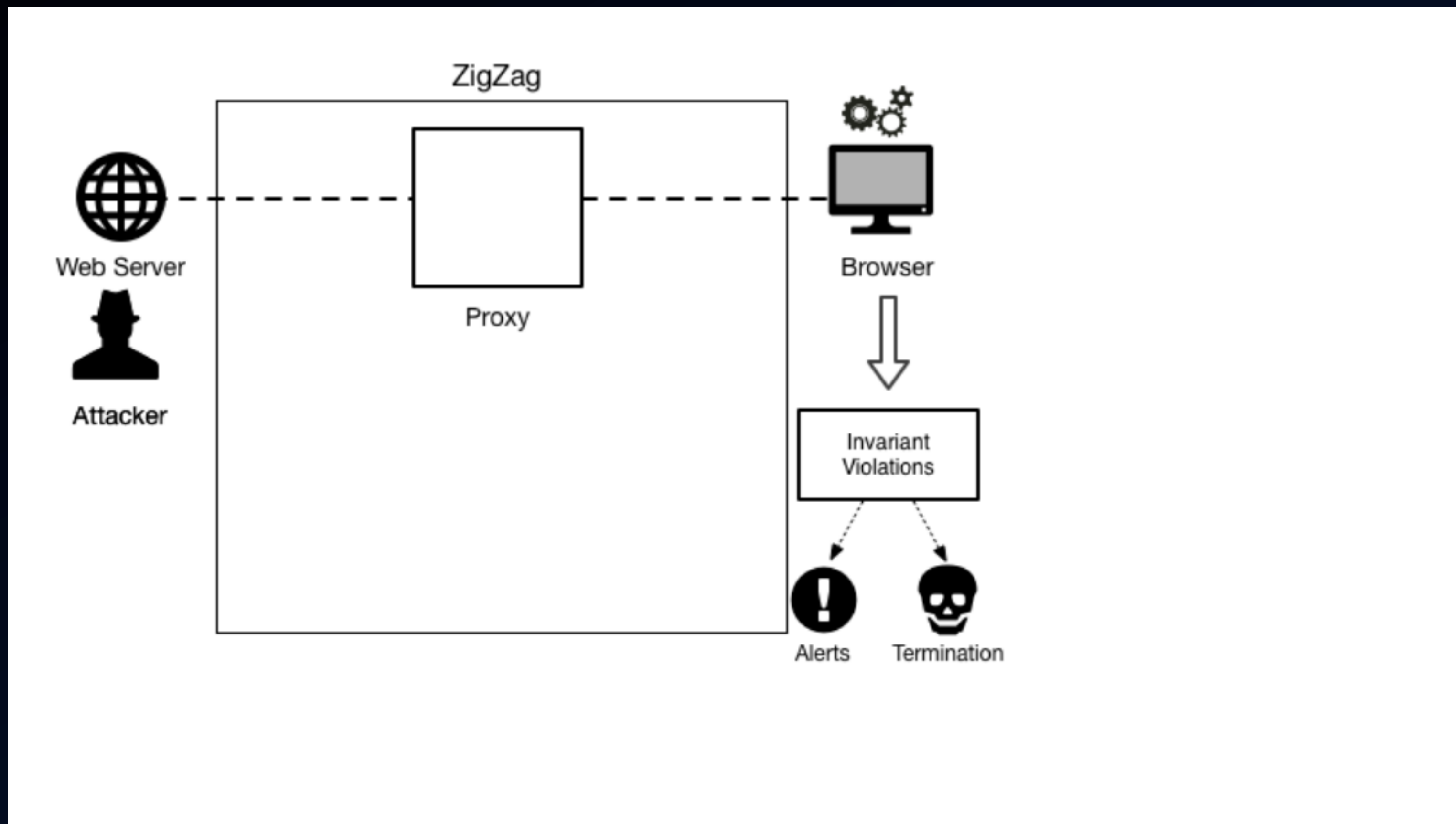
The hardened version of the web application preserves the semantics of the original for comparison to check for deviations

Enforcement Phase



Zigzag also incorporates runtime checks to enforce that execution does not deviate from what was observed during the initial learning phase

Enforcement Phase



If a deviation is detected, the system assumes that an attack has occurred and execution is either aborted or the violation is reported to the user

Invariant Detection

- Types of Invariants supported by ZigZag are

Data Types	Invariants
All	Types
Numbers	Equality, inequality, oneOf
String	Length, equality, oneOf, isPrintable, **isJSON, isEmail, isURL, isNumber**
Boolean	Equality
Objects	All of the above for object properties
Functions	**Calling function**, return value

Invariant Detection

- Dynamic program invariants are statistically-likely assertions established by observing multiple program executions.
- Trace collection and enforcement code is inserted at program points called checkpoints, We capture program state at checkpoints and compare subsets of these states for each individual checkpoint.
- There is no guarantee that invariants will hold in the future, Therefore, ZigZag only uses invariants which should hold with a high probability.
- These invariants are later used to decide whether a program execution is to be considered anomalous.

Invariant Detection

- ZigZag uses program execution traces to generate Daikon dtrace files. These dtrace files are then generalized into likely invariants with a modified version of Daikon developed.
- Daikon is capable of generating both univariate and multivariate invariants.
- Univariate invariants describe properties of a single variable, like the length of a string, the percentage of printable characters in a string, and the parity of a number.
- Multivariate models, on the other hand, describe relations between two or more variables, like example x == y or x<y.
- To reduce the overhead on system, ZigZag checks function coverage before issuing invariants for enforcement. It only allows for enforcement of a particular function after execution traces from four or more training sessions were collected, but this is easily configurable.

Program Instrumentation

- Trace collection and enforcement code is inserted at program points called checkpoints.
- Checkpoints are inserted at function prologues and epilogues, since events such as receiving cross-window communication are handled by functions.
- ZigZag performs a lightweight static analysis on the program's abstract syntax tree (AST) to prune the set of checkpoints that must be used.
- Functions which contain eval sinks, XHR requests, access to the document object, and other potentially harmful operations are labeled as important and only these functions are used in data collection and enforcement mode.

Program Instrumentation

- Each function labeled important is instrumented with pre and post function body hooks called *calltrace* and *exittrace*.
- Identifiers used are functionid uniquely identifies functions within a program, codeid labels distinct JavaScript programs, and sessionid labels program executions.
- Every invocation of calltrace increments and returns a global callcounter vaiable to provide a unique identifier such that calltrace and exittrace invocations can be matched.

Invariant Enforcement

- Given a set of invariants collected during the learning phase, ZigZag then instruments JavaScript programs to enforce these invariants.
- In our implementation, the calltrace and exittrace functions perform a call to an enforcement function generated for each function labeled important during the static analysis step.
- calltrace examines the function input state, while exittrace examines the return value of the original function
- These functions are generated automatically by ZigZag for each important function marked.
- Should an assertion be violated, a course of action is taken depending on the system configuration. Options include terminating execution by navigating away from the current site, or alternatively reporting to the user that a violation occurred and continuing execution

Deployment Models

- First, application developers or providers could perform instrumentation on-site, protecting all users of the application against CSV vulnerabilities.
- It is possible to deploy ZigZag as a proxy. In this scenario, network administrators could transparently protect their users by rewriting all web applications at the network gateway or individual users could tunnel their web traffic through a personal proxy.

Limitations

- The system was not designed to be stealthy or protect its own integrity if an attacker manages to gain JavaScript code execution in the same origin.
- So we presume the presence of complementary measures to defend against XSS-based code injection like Content Security Policy (CSP) or any auto-sanitization frameworks that prevent code injection in web applications.
- If the training set contains attacks, the resulting invariants might be prone to false negatives.
- If the training set is too small, false positives could occur.

Evaluation

- The ZigZag is evaluated using 4 real world case studies. First ZigZag is trained manually by browsing the application with one user for five minutes, starting with a fresh browser state four times. Next ZigZag is switched to the enforcement phase and attempted to exploit the applications.
- The attacks were caught by the ZigZag system while the functionality is retained at the same time.
- The median overhead measured over Alexa Top 20 sites is 2.01s(112%) while the microbenchmark was 0.66s.
- This is a fair trading off of some performance for improved security. It is acceptable for high assurance web applications and security-conscious users.

Conclusion

- ZigZag can be deployed by either the website operator or a third party. Website owners can secure their JavaScript applications by replacing their programs with a version hardened by ZigZag, thereby protecting all users of the application.
- Third parties, on the other hand, can deploy ZigZag using a proxy that automatically hardens any website visited using it. This usage model of ZigZag protects all users of the proxy, regardless of the web application.
- Evaluation shows that ZigZag can successfully instrument complex applications and prevent attacks while not impairing the functionality of the tested web applications. Furthermore, it does not incur an unreasonable performance overhead and, thus, is suitable for real-world usage.

THANK YOU