Marshall Plan Scholarship Final Report: Defence- and Detection-techniques against Malware with focus on Advanced Persistent Threats

Michael Weissbacher
mweissbacher@iseclab.org
Eduard Kunz str. 33
3032 Eichgraben

to:
Marshall Plan Foundation
Ungargasse 37
1030 Wien

April 10, 2012

Contents

1 General Information

Scholarship holder
Michael Weissbacher
Eduard Kunz str. 33
3032 Eichgraben
http://iseclab.org/people/mweissbacher

Duration of stay: 10 months
Field of study: Computer Science
Research topic: Defence- and Detection-techniques against Malware with focus on Advanced Persistent Threats

Home institution
Technical University of Vienna
Karlsplatz 13
1040 Wien, Austria
Advisor: Christopher Kruegel

Host institution
University of California, Santa Barbara
Department of Computer Science
Santa Barbara, CA 93106, USA
Advisor: Christopher Kruegel

2 Overview

2.1 Introduction

This report summarizes the scholarship holder's 10 month stay at the Computer Security Group of the University of California, Santa Barbara. I contributed to the project "Malicious Code Analysis and Detection", working on a JavaScript attack mitigation framework. The stay was financially supported by the Marshall Plan foundation. Other than the primary research target the seclab is active in hacking competitions, both on the competitors side as well as on the organizing side (UCSB iCTF.) These competitions will be described in the Events section.

2.2 Malware

Malware is short for malicious software, it is the root cause of most computer security threats today. In the past people have been working on such programs alone in their cellars for fun and were only seldom causing harm. Nowadays Malware has become a business, criminal organizations can profit through it in various ways, i.e.: extortion through DDoS (Distributed Denial of Service) attacks, credit card fraud with stolen data and spam. This group of software includes viruses, trojans, spyware, worms and others. The common denominator in these programs is that their intent is contrary to the users' belief about it, given that the person even knows of it being running on its system. Often Malware is propagated through vulnerabilities in software which is already installed on the users' computer. Prominent targets have been Microsoft Outlook, Internet Explorer and Adobe's Acrobat Reader. The user can get infected while surfing on some website (drive-by-downloads) or reading an email. Once the Malware is operative in the users' environment it may become part of a botnet where it can be used to send spam, participate in DDoS attacks, host phishing sites or steal passwords and credit card data. These botnets can consist of millions of computers (Conficker, Zeus, Mariposa) and pose a serious threat to the internet as a whole.

2.3 Advanced Persistent Threats

A subtype of Malware are so called *Advanced Persistent Threats* (APT). These don't target hosts at random, their goal is to steal inside information from companies or governments and stay undetected to be able to operate over long periods of time. Institutions which have data that is worth being stolen in such a clandestine way usually have no generic setup. Hence this Malware has to be tailored to the institutions' network and host security specifics. Techniques used include zero-day exploits of previously unknown vulnerabilities, strong encryption, polymorphism, binary obfuscation and

others. Once the network has been penetrated by an APT it can start collecting data and leak it to the outside attacker. Since the main goal of an APT is to remain undetected it is hard to ascertain how many networks are actively leaking data through such threats. The advanced nature of this Malware means that teams of expert attackers need to be well funded. Developing APTs belongs to the domain of major companies and governments.

A recent discovery of APT was *Operation Aurora*, an attack that targeted various high-tech companies including Google, Adobe, Juniper, Rackspace and potentially others. Attackers have gained access to their networks around mid-2009, they were accessing the companies' internal information including source code and potentially modifying it.

Goals of the research stay

The goal of the stay was to create new methods and algorithms to detect APT - for example by hardening JavaScript programs against CSV attacks. My approach is to combine hidden/dormant code analysis [8, 15, 29] with long term network monitoring which makes it possible to reverse engineer the network protocols [17, 39] and detect hidden botnet communication channels [20].

The UCSB security group has developed state-of-the-art intrusion detection and Malware analysis tools which I will refine and extend with new capabilities regarding APT. My work will contribute to the research project "Malicious Code Analysis and Detection".

Reason of relocation

The security group at UC Santa Barbara is a global leader in the fight against malicious code. Their tools were the first to create an in-depth analysis of the aurora exploits making Santa Barbara the ideal location for this project. Furthermore they have developed a number of tools for Malware analysis and tracking, that allowed them to collect a unique data set that I can work with.

Moreover, there are many people with deep insights into cybercrime, the internet underground economy and excellent connections with internet service providers. This allows me to apply my new analysis and detection algorithms on large, real-world data sets. These possibilities for an empirical evaluation are not available at Vienna University of Technology. Finally, the group is well-established, allowing me to perform world-class research and involvement in the security community.

The UC Santa Barbara security group is part of the International Secure Systems Lab, like the Viennese Seclab where I wrote my bachelor's thesis

and was working on a practicum. My work contributed to the emulation-based Malware detection software Anubis [10, 11, 12, 26].

3 Events

This section describes one-time events, conferences and CTF hacking competitions.

3.1 2011/04 Seclab retreat

The Seclab retreat was a one-time event at which every member of the lab had to present her current project covering problem statement, approach, evaluation, missing components, the next steps plus an idea for a new project. Two additional talks were held, "How to have success as a graduate student" by Christopher Kruegel and "How to succeed as an academic" by Giovanni Vigna.

3.2 2011/04 Plaid CTF

Capture the flag (CTF) competitions are hacking competitions in which teams have to compete for points by breaking challenges or hacking each other. These competitions test skills in areas such as computer forensics, cryptography, binary analysis, web security and others. Plaid CTF [4] was the first competition held by the CMU hacking team Plaid Parliament of Pwning. One particularly interesting challenge covered a cold boot attack, we had to gain access to an encrypted TrueCrypt volume by recovering the key from a memory dump. The scholarship holder summarized the approach in a blog post [1]. EpicFail (the UCSB hacking team) scored 7^{th}.

3.3 2011/05 IEEE Symposium on Security and Privacy

The IEEE Symposium on Security and Privacy in Oakland is one of the best conferences in the field and is a platform where high impact papers get presented. Most of the lab members were able to join the event. A paper that was directly related to my research was Taly et al.'s *Automated Analysis of Security-Critical JavaScript API* the talk helped to gain new insights. Johnson et al.'s paper *Differential Slicing: Identifying Causal Execution Differences for Security Applications* also gave me good input on some problems.

3.4 2011/06 DEFCON CTF qualifications

The DEFCON CTF is considered being the hacking world championship. Many teams want to compete, but only the best twelve teams from the qualification round can actually participate in the CTF hosted at the DEFCON

conference. The qualifications are a 53 hour non-stop online competition in which teams have to solve challenges of various complexity levels to gain points. Team Shellphish includes the UCSB team (EpicFail), UCSB alumni and some other individuals close to the seclab. Shellphish secured the 8^{th} place and qualified.

3.5 2011/08 DEFCON CTF

DEFCON CTF is hosted at the DEFCON conference which had its 19^{th} anniversary this year and takes place right after Black Hat in Las Vegas. The competition was run the third time by ddtek [3]. As a notable event, at least one team (lolersk8erz) gained access to the competitors machines and was able to enter the jailed operating systems of all other teams. We finished on the 9^{th} place. Although DEFCON hosts interesting talks and events all the participants of the lab were stuck behind the screen, fighting for points, from the beginning to the end of the event.

3.6 2011/08 USENIX Security

This conference is similar in impact to Oakland S&P. Two interesting JavaScript papers that are close to my field of research were presented: *ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection* by Curtsinger et al. and *ADsafety: Type-Based Verification of JavaScript Sandboxing* by Politz et al.

3.7 2011/12 UCSB iCTF

2011 was the 10^{th} edition of the UCSB iCTF [2] which is the biggest CTF competition worldwide. Teams of over 80 universities (more than 1000 individual players) competed for bragging rights and money.

I contributed several of the challenges, two of them covered JavaScript security. Furthermore I was in charge for administering the Internet Relay Chat (IRC) server during the competition that was used as primary means of communication between the teams and the organizers.

The UCSB iCTF was the first competition to allow participants to control objects in the physical space (in 2010.) In 2011 participants were able to control an unmanned drone over the internet. A write-up to some of the challenges and a visualization of the progress on how teams solved the challenges over time can be found on on the scholarship holders website [1].

4 Repeated Events

4.1 Hackmeetings

Hackmeetings occurred about every other week. In these meetings we would train for hacking competitions by breaking challenges of previous events or other available targets.

4.2 SRG - Security Reading Group

Every SRG member can propose papers. The elected paper will be read and discussed in the weekly session. SRG is an important instrument to keep the lab members updated on state of the art papers that may be outside of their direct research scope.

4.3 Flashmeetings

During a flashmeeting all members of the lab get two to three minutes to talk about advances in their research and problems they were struggling with since the last meeting. Open problems lead to discussions in the group which can help to solve the problem right away. Topics that would consume too much time get rescheduled to separate meetings with the advisors. Flashmeetings were held twice a week when I joined the lab and later reduced to a weekly schedule.

4.4 Meetings with advisors

The scholarship holder was meeting with the advisors either at follow-up meetings after the flashmeetings, or individual appointments on request.

5 Research Project - ZigZag

This section presents details on project ZigZag, my main task during the stay. ZigZag is the name of the prototype of instrumentation framework for CSV attack detection for JavaScript programs. Some of these attacks can be classified as APT.

5.1 Introduction

In recent years, through the introduction of technologies such as AJAX and Web 2.0, there has been a major paradigm shift in the way that software is developed and deployed. Ten years ago, most software was written for and executed on desktop machines (PCs) of end users. However, with the growth of the Internet and the recent push towards cloud computing, the landscape has changed. These days, many applications are no longer running

on the client alone. Instead, there exists a hybrid deployment model where a part of the application is running on a remote server (often in the cloud), while the other part of the application is running in the browser of the user. This client-side part is typically written in JavaScript, and it makes use of asynchronous calls to connect to the server (AJAX).

Compared to traditional desktop software, the security of Web 2.0 applications is not well-understood. This is a significant problem as one can expect that an increasing number of applications will be written following this model. One important class of vulnerabilities for Web 2.0 applications are *client-side validation (CSV) vulnerabilities*. CSV vulnerabilities are programming mistakes in the client-side portion of web applications. These vulnerabilities can be exploited by an attacker who can send inputs to these programs, using a number of channels that allow sites (scripts) in different browser windows, and from different origins (web sites) to communicate and exchange data. Examples include the cross-window communication interface, referrer data, and reflected flows. By sending inputs to a vulnerable application, the goal of the attacker is to perform actions through this application on behalf of the victim user. For example, the attacker can send emails through a vulnerable web mail program, obtain cookie data, or display a phishing site to obtain a user's credentials. What makes these attacks special is that they cannot be detected on the server side – they are only visible on the client (web browser). Hence, security advances and hardening at the server side do not help to mitigate client-side validation vulnerabilities. Instead, novel, client-side solutions are required.

The goal of this project is to develop novel techniques and tools to protect the execution of the client-side parts of web applications. In particular, we propose to automatically instrument client-side scripting code so that its execution can be monitored. We use this monitoring framework to (i) build models that capture normal program runs, and (ii), subsequently leverage these models to detect (and prevent) attacks as they unfold.

The programming language that is used for most client-side components is JavaScript. Until recently, JavaScript was mostly used for small scripts that enhance the appearance and user experience of web sites. While security implications of small, isolated scripts were studied in the past, interacting and complex components, which are typical for new Web 2.0 applications, pose a different threat. Moreover, new HTML features, such as `postMessage` do not adhere to the "same origin policy (SOP)." The SOP enforces constraints on data accesses based on the origin of the request: For example, a cookie written by a script loaded from `example.com` cannot be accessed by code loaded from `attacker.com`. This policy is implicitly enforced by the browser, not only on cookies but also on JavaScript methods and web page contents. It is well-understood by developers, and hardens the client-side against attacks.

`postMessage` was introduced to allow scripts from *different* origins to communicate directly with each other. This significantly eases the process of creating web mashups and makes response times shorter. However, for a safe use of this function, the recipient of a message must check that data was sent by a legitimate, expected script. Unfortunately, these checks are often forgotten, which allows an attacker to write malicious scripts that send messages to vulnerable applications. To this end, a malicious web site can include the vulnerable application in a hidden `iframe` and directly send messages to it. Such vulnerabilities are not simply theoretical, but have already had real-world impact. In a recent attack against Google Gmail, such a vulnerability was exploited to allow attackers to access victims' mails or to send mails on their behalf.

As a typical example of a CSV vulnerability, consider the following simple code that first opens a popup window (Line 10) and then uses this window to display messages received from a backend server at `csv-example.com` (Lines 11 to 15):

```
 1:  var messagesURL = "http://csv-example.com";
 2:  popURL = "messages_popup.html";
 3:  var popup;
 4:  ...
 5:  function receiveMessages(msg) {
 6:     var message = getMessage();
 7:     popup.postMessage(message,messagesURL);
 8:  }
 9:  ...
10:  popup = window.open(popURL);
11:  for(;;) {
12:     ...
13:     receiveMessages(msg);
14:     ...
15:  }
```

The problem here is that it is the programmer's responsibility to validate that the messages that the popup window receives are indeed coming from the legitimate domain `csv-example.com`. For example, consider the following code that the popup window might use to receive and display messages:

```
1.  function displayMessage (evt) {
2.    var message;
3.    message = "I got " + evt.data;
4.  }
```

Unfortunately, the developer does not validate the origin, and hence, this code is vulnerable! Any malicious website could send a message to the popup window that is then displayed to the user. To fix this vulnerability, the developer would have to insert a piece of code that validates the origin, such as the following:

```
1.    if (evt.origin !== "http://csv-example.com") {
2.       message = "You are not worthy";
3.    }
4.    else {...
```

Although the example we provided is simple, similar to cross-site scripting (XSS) attacks, the possibilities of exploitation of a CSV by an attacker are only limited by her imagination. Among the possible attacks are origin misattribution, code injection, command injection, and cookie-sink attacks [31].

Given the strong trend towards web applications, and the increasing support for complex, client-side components by new HTML standards and browser improvements, we expect to see more and more critical applications developed following the Web 2.0 paradigm. Of course, this means that client-side vulnerabilities will become more widespread, and the severity of exploits will increase. This makes the development of new techniques to protect client-side code critical. Zigzag represents the first step towards this goal.

5.2 State of the Art

There are two main approaches [19] to test software applications for the presence of bugs and vulnerabilities: white-box testing and black-box testing. In white-box testing, the source code of an application is analyzed to find flaws. In contrast, in black-box testing, input is fed into a running application and the generated output is analyzed for unexpected behavior that may indicate errors.

When analyzing web applications for vulnerabilities, black-box testing tools [6, 13, 24, 37] are the most popular. Some of these tools (e.g., [6]) claim to be generic enough to identify a wide range of vulnerabilities in web applications. However, recent studies [9, 18] have shown that scanning solutions that claim to be generic have serious limitations, and that they are not as comprehensive in practice as they pretend to be.

Two well-known, older web vulnerability detection and mitigation approaches in literature are Scott and Sharp's application-level firewall [32] and Huang et al.'s [21] vulnerability detection tool that automatically executes SQL injection attacks. Scott and Sharp's solution allows to define fine-grained policies manually in order to prevent attacks such as parameter tampering and cross-site scripting.

With respect to white-box testing of web applications, a large number of static source code analysis tools (e.g., [23, 34, 40]) that aim to identify vulnerabilities have been proposed. These approaches typically employ taint tracking to help discover if tainted user input reaches a critical function without being validated. Previous research has shown that the sanitization process can still be faulty if the developer does not understand a certain class of vulnerability [7].

Note that there also exists a large body of more general vulnerability detection and security assessment tools (e.g., Nikto [27], and Nessus [35]). Such tools typically rely on a repository of known vulnerabilities and test for the existence of these flaws.

With respect to scanning, there also exist network-level tools such as nmap [22]. Tools like nmap can determine the availability of hosts and accessible services. However, they cannot detect higher-level application vulnerabilities or CSV vulnerabilities.

CSV vulnerabilities were first highlighted by Saxena et al. [31]. In their work, the authors propose FLAX. FLAX is a framework for CSV vulnerability discovery that combines dynamic taint analysis and fuzzing into *taint enhanced blackbox fuzzing*. The system operates in two steps. First, a JavaScript application under test is executed to dynamically identify all data flows from untrusted sources to critical sinks (where critical sinks are functions such as cookie writes, eval, or XMLHttpRequest operations). This flow information is processed into small executable programs, called *acceptor slices*. These programs accept the same inputs as the original program but are reduced in size. Second, the acceptor slices are fuzzed using an input-aware technique to find inputs to the original program that can be used to exploit a bug. A program is considered to be vulnerable when a data flow from an untrusted source to a critical sink can be established. Later, the same authors improved their FLAX system by replacing the dynamic taint analysis component with a static analysis component [30]. Again, the goal of the static analysis is to find unchecked data flows from inputs to critical sinks. Because the system uses static analysis, its false negative rate decreases.

The main difference between our proposed work and the FLAX system is that FLAX can only detect attacks where foreign JavaScript *code* is injected into the vulnerable application (and executed by the sink function). Our approach, on the other hand, also handles situations where anomalous program executions are caused by malicious *data* that is injected into the vulnerable process. In this case, no new code is added to the application, but existing code is misused by executing existing functions in an unwanted order and/or with invalid arguments. To draw a similarity with more traditional attacks against programs written in C, FLAX would handle cases in which an attacker injects shellcode into the vulnerable process. Our solution, on the other hand, is more complete and, in addition to basic code injection

attacks, also covers return oriented programming (ROP) exploits [33] and non-control-flow attacks [14].

As we will see later, our solution requires that client-side JavaScript is instrumented. Previous research has examined various ways in which JavaScript instrumentation can be implemented. This is a non-trivial problem because JavaScript can be considered to be self-modifying code (as a running program can generate input code for its own execution). This renders (static) instrumentation without execution impossible since not all code can be processed by the initial instrumentation step. Hence programs have to be instrumented before execution and consequently all writes to the program's code have to go through another instrumentation step. An interesting proxy-based approach to JavaScript instrumentation was discussed by Yu et al. [25, 41]. For our work, we can leverage existing proposals for JavaScript instrumentation. The novel part of our research is to determine what needs to be instrumented, and how security violations can be detected.

5.3 Statement of Work

The objective of this project is to develop a system, called ZigZag, that performs automated code instrumentation of the client-side JavaScript portion of web applications. The purpose of this instrumentation is twofold: In a first step, *during a monitoring phase, the instrumentation code observes the execution of the client-side code and builds models that characterize and capture normal program runs.* Once these models are built, *ZigZag switches into detection and enforcement mode.* In this mode, the instrumentation code checks the program execution for anomalies. Such anomalies are indications of attacks. In a stricter setting, the model can also *enforce the execution so that it is never permitted to violate a model.* Of course, the instrumentation has to be complete (to cover different attack vectors) and efficient (minimizing penalties to code performance).

The project will be structured as the following complementary and closely-related main tasks, which fall into three main research activities:

- **Task 1: CSV Classification and Theory**

  Subtask 1.a: Study and collect well-known real-world CSV exploits and examples.

  Subtask 1.b: Construct a classification and taxonomy as a guideline for the development and evaluation of detection and prevention approaches designed to cover whole classes of CSVs.

- **Task 2: Data Collection**

  Subtask 2.a: Construct and evaluate basic data collection components for the monitoring and instrumentation of client-side JavaScript code. This component will focus on function calls and their argument values.

Subtask 2.b: Improve the basic collection component to create a more comprehensive (but possibly more costly) data collection framework. This framework will collect data from local variable assignments, data invariants, global context, and timing information.

Subtask 2.c: Investigate and develop distributed data collection techniques that divide up the load of building program execution profiles over multiple (many) program executions.

- **Task 3: Detection Models**

  Subtask 3.a: Construct models that use the information collected by the basic collection component to characterize benign program runs and detect anomalies as client-side attacks.

  Subtask 3.b: Generalize and improve the models to take into account the additional information provided by the advanced data collection framework.

  Subtask 3.c: Perform extensive experiments with real code to evaluate the detection models and prototypes for their ability to identify and prevent exploitation based on CSVs.

  Subtask 3.d: Develop techniques to lower potential false positives and integrate detection capabilities into the browser.

5.4 Detailed Research Plan

In the following paragraphs, we discuss the research plan in more detail. First, we provide a brief overview of the general approach. Then, we discuss key challenges. Finally, we present our approach to realize the individual components of ZigZag.

5.4.1 Research Overview

The goal of ZigZag is to protect client-side JavaScript execution against exploitation. To this end, we propose to leverage proven anomaly detection techniques. In particular, our approach helps to decide whether a monitored execution run deviates from models learned for legitimate executions. When such a deviation (an anomaly) is detected, the program can be terminated to avoid putting the user at risk. As a result, the application that executes on the client-side will be armored against CSV vulnerabilities.

Anomaly detection is an approach to that is complementary to the use of signatures in detecting attacks. It relies on models of the normal behavior of applications to identify anomalous activity that may be associated with attacks. The main advantage of anomaly-based techniques is that they are able to identify previously unknown attacks. By defining the expected,

normal behavior, any abnormality can be detected, whether it is part of a known attack or not.

As part of the anomaly detection process, we need to build models of normal program activity. This requires three steps: First, we need to determine features (or properties) that define the execution of a program. For example, we could record the sequence of function calls that a program makes, or we can infer invariants between variables that must hold for all benign executions. Second, we require a mechanism to record the activity of a specific program execution. In our case, this implies that we need a component *at the client-side* that can monitor the execution of scripts. The collected data is needed for the third step. In this third step, we use the information collected for legitimate program runs to build models that capture a notion of "normality." Of particular interest for the detection of attacks against web applications are machine learning-based techniques, which can build a model of the normal behavior of an application by observing the usage patterns of the application during a training period.

Once the model of normal behavior is established, the system switches to "detection mode" and compares the behavior of the application with respect to the model learned during the training period, with the assumption being that anomalous behavior is likely to be associated with an attack (and that an attack will result in anomalous behavior). Since we aim to protect client-side executions, the models need to be sent to the client together with the script that should be protected.

For both the training and the detection phase, it is necessary to monitor the execution of client-side JavaScript code. For this, we could instrument the browser or require users to install a plug-in. Of course, this severely restricts the ease of deployment. Thus, we propose to develop an instrumentation framework that can add the necessary data collection and detection/enforcement hooks directly to the client-side scripts before they are sent to the web browser. This instrumentation framework can be integrated with the server-side part of the web application or with a proxy. Using a proxy has the advantage that the instrumentation is fully transparent to the web application.

5.4.2 Research Challenges

Research Challenge: *How can we create detection models that can identify malicious client-side JavaScript activity?*

Our goal in this project is to create a system that can automatically monitor and identify attacks against client-side JavaScript code. We envision a system that can automatically produce attack alerts (and potentially block them) after a pre-defined training period. The key idea is to create a system that can use machine learning algorithms to train on a set of behavioral

features, and that can then detect executions that exhibit malicious, attack-like behavior.

The first step in creating effective detection models will be to identify behavioral features that are indicative of attacks. For example, we will investigate if function call sequences, parameters, timing information are useful features in detecting attacks.

Research Challenge: *How can we deal with dependencies of client-side code on third-party tools and systems?*

Web applications are highly interactive today, and they often live in a heterogeneous environment. For example, they may depend on backend servers and databases, and may even depend on other websites for delivering content.

When building an anomaly-based malicious activity detection system, it is, hence, very important to take such dependencies into account. The behavior of the client-side code may change not because there is an attack, but because there is a failed dependency at some point. For example, a crashed database may cause the client code to behave in an anomalous, suspicious way.

One of the challenges of this research is to devise novel techniques to deal with non-standard situations when instrumenting and monitoring the executions of the client code. That is, heuristics need to be developed that can identify anomalous behavior as being an error (e.g., a database crash) rather than an attack. One way of achieving this, for example, would be to analyze application logs for classic signs of error messages. For instance, correlating a "database connection refused" message with a corresponding non-standard behavior would eliminate the false positive.

Research Challenge: *How can we deal with the complexity of client-side code that typically consists of thousands of lines of code and decide on what to instrument?*

In the earlier days of the web, web applications typically contained a few lines of JavaScript code. Since then, there has been a dramatic change in the size and complexity of client-side web application code. Today, JavaScript code in a highly-interactive, sophisticated web application (e.g., GMail, Facebook) easily exceeds several thousand lines of code.

JavaScript code is critical and decisive for developers to be able to implement interactive web applications that are performant and that provide an appealing look and feel. At the same time, JavaScript code suffers from the same type of bugs found in traditional software such as race conditions, logic errors, performance bottlenecks, and even memory leaks.

In comparison to traditional software, one of the challenges of monitoring and analyzing JavaScript code is that there is a lack of isolation between

different parts of the code. Typically, the code is not developed in modules (e.g., by using namespaces) and the fact that the code is running on the user's (or potentially, the attacker's) machine means that it cannot be trusted.

When designing a monitoring and detection system that aims to protect applications against attacks that are based on CSVs, it is important to assume that the developed components on the client might be operating in an adversarial setting. Furthermore, novel techniques and approaches are required to deal with the complexity of the code, and to decide how and what to instrument.

We plan to study well-known examples of CSVs (e.g., such as the attack against GMail as discussed in the introduction), and use these case studies as a starting point into what and how to instrument. We will then generalize the instrumentation strategies that are based on specific attacks to cover more generic attacks.

Research Challenge: *How can we deal with client-side script execution environments that are non-standard?*

Web 2.0 application code that runs on the client shares most of the development challenges of traditional, complex software systems. One of the core challenges of analyzing and monitoring Web 2.0 applications is that they may sometimes rely on non-standard execution environments. For example, although the JavaScript language has been standardized in 1999 (i.e., ECMA 1999), the implementations of the JavaScript execution environments may differ. As a result, real-world Web 2.0 implementations may sometimes contain incompatibilities across different browsers.

For example, certain event handlers that have been registered for the same event (e.g., logging key presses) may fire in a different order on different browsers. Clearly, as the implementations of browsers differ, so do the implementations of the run-time environments. Note that these differences in browser behavior are well-known by attackers, and are sometimes used to launch attacks (e.g., in a cross site scripting attack, a script tag may be encoded as "script" on a specific browser in order to bypass filtering mechanisms. That is, while this may work on Firefox, it may fail on IE 6. Hence, a defender that is solely relying on the behavior of IE 6 when creating filters would be vulnerable to an attack if the user uses Firefox).

In order to deal with the potentially differing behaviors of browsers, we plan to deploy and test the monitoring techniques on different popular browsers (e.g., Firefox, IE, Safari).

Research Challenge: *How can we design instrumentation and monitoring approaches that are performant and efficient?*

One downside of code instrumentation for monitoring is that the instrumented code will inherently take a performance hit. Hence, one of the

challenges of monitoring and attack detection systems is that they do not significantly impact the performance and efficiency of an application. Clearly, an application that is secure, but not performant (and hence not usable by users) would not be effective in practice.

A technique we could use to to deal with performance issues is to make the instrumentation and monitoring adaptive and distributed. Web 2.0 applications typically support multiple users and hence, the instrumentation of code can be distributed across a large user population.

One of the research challenges in this work is to decide how to distribute the instrumentation load so that the instrumented web application does not suffer a significant performance hit. To make sure that the performance hit is acceptable, we plan to conduct regular performance experiments with real code, and then to use the results to improve the architecture and the implementation of the prototypes.

5.4.3 Architecture of ZigZag

As discussed previously, the deployment of the tool consists of two main phases: the Training Phase and the Detection Phase. In the Training Phase, the web application is instrumented and execution data is collected. Using this data, models for anomaly detection are generated. In the Detection Phase, these models are deployed on the web client, and they detect and prevent CSVs.

The ZigZag system consists of three main components: First, we require a framework that allows the instrumentation of JavaScript (JS) programs. Second, we require components to model the execution of JS. Third, we require an anomaly-based attack detection component to check and enforce the execution of a JS program, given a set of execution models. These components are discussed in more detail in the following sections.

5.4.4 Data Collection and Code Instrumentation

For the training phase, it is necessary to instrument a client-side script with code that can record certain properties of the execution. For example, this can be the fact that a specific function is called, or the assignment of values to variables as a certain program point. For the detection phase, it is necessary to instrument client programs with hooks into the detection models, which verify whether the current run is legitimate (i.e., conforms to the given the models).

For the instrumentation component of ZigZag, a possibility would be to use a source-to-source compiler for JavaScript. It is to be determined, however, whether the level of granularity is sufficient to gather all needed features. For deployment, the idea is to add a proxy in front of the web application so that all code that is sent from the server to the client can

be *transparently* instrumented by our system, without any changes to the back-end web application and without requiring any annotations from the application developers.

The idea of the instrumentation phase is to rewrite the JavaScript code of the web application, and to include instrumentation code for communicating with the data collection components, and to insert monitoring and attack detection code. All function calls of the original program are augmented with a call to the trace function as the first line of code.

One of the design issues that will be investigated in this work is how often the collected data should be sent to the analysis server. Clearly, uploading the collected data too often will have a performance impact. We plan to store the collected data in buffers, and to flush these buffers periodically to the analysis server or once the buffer reaches a predefined size. Timing functions are important to determine whether the monitored code is still being executed, or is inactive.

Note that establishing the communication between the client program and the analysis server is not a straightforward task. We have to work around the single origin policy of JavaScript which restrains the communication possibilities of the program to the host/protocol/port the JavaScript program has been loaded from. This problem can be resolved by installing a proxy between the client and the server. Messages can be sent that start with a key that does not occur in the application, and the proxy can intercept these and forward them to the analysis server.

To ensure the instrumented JavaScript programs are still performant after the instrumentation, we will consider several data transfer possibilities to the analysis server. The most favorable one seems to be a transfer that leverages JavaScript worker technology. With this functionality, we will be able to perform the upload to the analysis server in a separate thread. The only shortcoming is that this feature is only available in the Firefox and Google Chrome browsers. Microsoft Internet Explorer does not support JavaScript workers. However, if the browser the script is running in does not support workers, we can still drop back to data transfers that use XMLHttpRequest. Hence, we can make sure that we are compatible with many browsers, while at the same time making sure that the performance hit as small as possible.

Despite optimizations, it is possible that the data collection process is too costly (slow), especially when all possible features are collected for each run. To address this problem, we plan to investigate the idea of *distributed data collection*. That is, we leverage the fact that when the system is deployed, instrumented code is run by many users in many browsers. The key idea is that we do not need to collect all possible data for each run. Instead, we can collect only parts for each run, and then combine all collected data into a single model that reflects, as accurately as possible, all potential program executions. Of course, we cannot simply remove data collection points at

random. For example, when we want to check for the relationship between certain variables, we need to see both variables in a single execution run. Also, when we collect sequences of events, it is difficult to drop random events from this sequence, as crucial information would be missing.

5.4.5 Modeling Program Executions

To guide the collection of data, we need to define features that capture (JavaScript) program executions. The anomaly detection approach envisioned in this proposed work is based on the application-specific analysis of individual function calls of the JavaScript code running on the client. For each function invoked by the application, a distinct profile will be created. Each of these profiles captures the notion of a "normal" function invocation by characterizing "normal" values (e.g., the order in which the arguments are given, the time it takes to execute the function, etc.).

The expected "normal" executions of individual functions are determined by models. A model is a set of procedures used to evaluate a certain feature of a function, such as the length of its arguments and the number of local variables that it defines during execution.

To determine features that can capture and characterize the benign behavior of an application, we will initially study well-known instances of CSV attacks and investigate how the malicious executions of the sample code differ from benign runs. After this analysis period, we will start to experiment with features that are able to capture malicious executions. The goal in this step will be to define detection feature sets where each feature might be a tell tale sign for a CSV-based attack. Clearly, it is improbable that a feature by itself can indicate with certainty that an execution is malicious. Rather, a combination of these features will help determine the probability that we are observing a malicious JavaScript execution.

In the following, we describe some of the feature sets that we will investigate, and explain why we believe that they may be sufficient to capture malicious behavior. Note that the discussed list is not comprehensive, and that research and experimentation will be required to identify models that perform well in practice.

Sequences of Function Calls: One of the first features we can analyze are the sequence of function calls. The intuition here is that an attack may change the normal sequence of functions that is observed during benign runs. For example, imagine a case where the attacker is able to inject a malicious script into an application by exploiting a CSV. If we monitor the order in which the functions in the scripts are typically invoked, the injected script would deviate the execution and cause an anomaly.

Monitoring function sequences and creating training techniques that are based on the order in which functions are invoked is analogous to intrusion

detection techniques that are based on system call sequences.

Function Arguments: One interesting feature set to investigate consists of argument values that are passed to JavaScript functions. We are interested in the values that these arguments assume, their typical ordering, and their lengths.

For example, imagine the case length of the first argument of a function in the script code rarely exceeds a hundred characters and mostly consists of human-readable characters. Suppose, now, that malicious input is passed to this function that consists of several hundred characters in size as the attacker is launching a cross site scripting attack, and is injecting code. Our training on the normal length of the first argument would show that the attacker's input is anomalous, and we would be able to detect this attack.

As another example, recall the simple CSV vulnerability we presented in Section 5.1. In the example, an attacker was able to inject messages into a popup window. In this case, the argument value in benign runs would always consist of `http://csv-example.com`. However, if an attacker would inject, say, `http://www.attacker.com`, this input would be identified as being anomalous and potentially malicious.

When considering the argument length, the goal of the model is to approximate the actual but unknown distribution of the lengths of a string argument and detect instances that significantly deviate from the observed normal behavior. Clearly, one cannot expect that the probability density function of the underlying real distribution would follow a smooth curve. One also has to assume that it has a large variance. Nevertheless, the model should be able to identify obvious deviations.

When analyzing function arguments, one could also look at character distributions. The string character distribution model would capture the concept of a "normal" or "regular" string argument by looking at its character distribution. The approach is based on the observation that strings have a regular structure, are mostly human-readable, and almost always contain only printable characters.

Often, the manifestation of an exploit is immediately visible in function arguments as unusually long strings or strings that contain repetitions of non-printable characters. There are situations, however, when an attacker is able to craft her attack in a manner that makes its manifestation appear more regular. For example, non-printable characters can be replaced by groups of printable characters. In such situations, we need a more detailed model of the function argument. This model can be acquired by analyzing the argument's structure. For our purposes, the structure of an argument could be the regular grammar that describes all of its normal, legitimate values. Structural inference is the process by which this grammar is inferred by analyzing a number of legitimate strings during a training phase.

Execution Timing: A promising feature that may help detect CSV attacks is the timing information related to the execution of a script. The key insight here is that an attack will often cause a delay in the execution when the attacker deviates from the benign execution to the malicious one. By creating execution timing profiles of of the client-side code of a web application, we expect to be able to identity anomalous situations.

For example, imagine a situation where the attacker is able to inject JavaScript code into a web application by exploiting a CSV that is hosted on a remote server `http://www.attacker.com/attack.js`. The attacker's code reads the application cookie of the victim, and sends this to a remote server that the attacker has access to. By using execution timing profiles, there is a good chance that we would be able to spot an attack such as this one. That is, the time that it would take the application to complete a specific benign functionality would change as extra statements injected by the attacker (e.g., reading the cookie) would be invoked.

One of the challenges in using execution timing as a feature to identify anomalous behavior is that such timing profiles may be fragile and may cause false positives. For instance, the timing profile of code may also change because of errors, or delays in communication. Hence, one of the objectives of the proposed work will be to investigate what portions of the client-side code can be instrumented to record timing profiles while at the same time providing reliable signs for malicious activity.

Global Context and Variable Invariants: In a typical attack, the attacker is able to deviate the execution of the targeted code from normal behavior by injecting or using statements that implement her malicious intent. While doing this, the attacker will often need local or global variables. For example, she may access the contents of a global variable that stores the value of a sensitive parameter (e.g., session ID). As another example, she may define local variables that hold the value of the application cookie before it is sent to a remote server under the attacker's control.

Hence, suspicious, anomalous definitions of variables during execution, and abnormal patterns of access to them can be good indicators for detecting malicious activity. In this work, we plan to investigate how variable definitions and usage can be utilized to model program execution and to create a "normal" execution profile.

Moreover, for legitimate program executions, it is often possible to extract "relationships" between variables that always hold. Such invariants are frequently violated as part of an attack where the malicious code or data tampers with the intended application functionality.

5.4.6 Generating Models and Detecting Attacks

Based on the features described in the previous section and the data collected from client-side program runs, we can build detection models. The first step for building the detection models will be to construct the training set. Clearly, the quality of the results produced by a machine learning algorithm strongly depends on the quality of the training set [36]. Hence, our goal will be to develop a classifier that is able to label executions as being benign, or belonging to an attack. Thus, we require a training set that contains a representative sample of benign and malicious executions. To this end, one of our tasks will be to study many known examples of client-side vulnerabilities and attacks, and then to use these for constructing the training set.

For this work, we are planning to use machine learning techniques that have proven to be effective in practice. An important step in building the detection models will be to select a suitable classifier. A popular choice for binary classification is Support Vector Machine (SVM) [16]. SVM consists of a set of supervised learning methods that build a hyper-plane from the training set in order to divide the elements of the training set into different spaces with the maximum distance possible. Another choice is J48 [38]. J48 is an implementation of the C4.5 algorithm [28] that is designed for generating either pruned or unpruned C4.5 decision trees. It constructs a decision tree from a set of labeled training set by using the concept of information entropy.

Although SVM is the most successful classification method for binary classification, it has some disadvantages (e.g., such as the performance and having hidden decision factors). Hence, the reason why a decision has been taken is not always clear. Therefore, in our system, we plan to correlate the results of an SVM and a J48 decision tree classifier so that we are able to have good results (while at the same time, also being able to understand the reasons for the decisions).

After establishing models of benign behavior, we do not require additional information from clients. Instead, models are included in the code that is sent to clients. In particular, the client code will be instrumented to check whether the previously-established models (i.e., learned by training) are violated.

Note that CSV cannot be detected at the server, since it involves one malicious script attacking another one. Thus, it is necessary to insert detection capabilities at the client-side. Because we do not want to change browsers, the instrumentation of JavaScript code is the natural choice. Also, by learning models (and specifications) of normal program execution from a large number of legitimate runs, we can create comprehensive descriptions that tightly model permissible behaviors.

6 Acknowledgments

In Vienna I want to thank the Marshall Plan Foundation for making the trip possible. In Santa Barbara I want to thank my advisors Prof. Giovanni Vigna and Prof. Christopher Kruegel who invited me and provided lots of insights. The Computer Security group at UCSB proved to be a very productive and friendly environment. We had many discussions and were often working in the lab until late at night. The stay was a great experience.

References

[1] http://mweissbacher.com.

[2] https://ictf.cs.ucsb.edu.

[3] http://www.ddtek.biz/.

[4] http://www.plaidctf.com/.

[5] *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[6] Acunetix. Acunetix Web Vulnerability Scanner. `http://www.acunetix.com/`, 2008.

[7] Davide Balzarotti, Marco Cova, Vika Felmetsger, Davide Balzarotti, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, 2008.

[8] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, CA, USA*, 02 2010.

[9] Jason Bau, Elie Burzstein, Divij Gupta, and John. C. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of IEEE Security and Privacy*, May 2010.

[10] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*. The Internet Society, 2009.

[11] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 1871–1878. ACM, 2010.

[12] Ulrich Bayer, Andreas Moser, Christopher Krügel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[13] Burp Spider. Web Application Security. `http://portswigger.net/spider/`, 2008.

[14] Shuo Chen, Jun Xu, Emre Sezer, , Prachi Gauriar, and Ravishankar Iyer. Non-control-data attacks are realistic threats. In *Usenix Security Symposium*, 2005.

[15] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying dormant functionality in malware programs. In *IEEE Symposium on Security and Privacy*, pages 61–76. IEEE Computer Society, 2010.

[16] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[17] Weidong Cui. Discoverer: Automatic protocol reverse engineering from network traces. In *In Proceedings of the 16th USENIX Security Symposium*, 2007.

[18] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, 2010.

[19] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.

[20] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *NDSS* [5].

[21] Y. Huang, S. Huang, and T. Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *12th World Wide Web Conference*, 2003.

[22] Insecure.org. NMap Network Scanner. `http://www.insecure.org/nmap/`, 2010.

[23] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[24] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *World Wide Web Conference*, 2006.

[25] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript Instrumentation in Practice. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[26] Matthias Neugschwandtner, Christian Platzer, Paolo Milani Comparetti, and Ulrich Bayer. anubis - dynamic device driver analysis based on virtual machine introspection. In Christian Kreibich and Marko Jahnke, editors, *DMIVA*, volume 6201 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010.

[27] Nikto. Web Server Scanner. `http://www.cirt.net/code/nikto.shtml`, 2010.

[28] J.R. Quinlan. Learning with continuous classes. *Proceedings of the 5th Australian joint Conference on Artificial Intelligence*, Singapore: World Scientific:343 – 348, 1995.

[29] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC*, pages 289–300, 2006.

[30] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Stephen McCamant, Feng Mao, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[31] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Network and Distributed Systems Security Symposium*, 2010.

[32] D. Scott and R. Sharp. Abstracting Application-level Web Security. *11th World Wide Web Conference*, 2002.

[33] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[34] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages*, 2006.

[35] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. `http://www.nessus.org/`, 2010.

[36] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Academic Press, 2009.

[37] Web Application Attack and Audit Framework. `http://w3af.sourceforge.net/`.

[38] IH. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[39] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. Automatic network protocol analysis. In *NDSS* [5].

[40] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.

[41] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Symposium on Principles of Programming Languages (POPL)*, 2007.