See No Evil: Evasions in Honeymonkey Systems

Brendan Dolan-Gavitt & Yacin Nadji

May 7, 2010

Abstract

Client-side attacks have emerged in recent years to become the most popular means of propagating malware. In order to keep up with this new wave of web-based malware, companies such as Google routinely crawl the web, feeding suspicious pages into virtual machines that emulate client systems (known as *honeymonkeys* or *honeyclients*). In this paper, we will demonstrate that although this approach has been successful to date, it is vulnerable to evasion by attackers: by guarding exploit code with client-side checks to determine whether a visitor is a human or an automated system, attackers can ensure that only human users are exploited, causing the page to look benign when viewed by a honeyclient. Our 13 evasions, which include both observational and interactive proofs of humanity, are able to easily distinguish between honeyclients and human victims. We evaluate the strengths and weaknesses of these evasions, and speculate on even more powerful attacks and defenses.

1 Introduction

Attacks against end users have undergone a dramatic shift in the past several years, moving from attacks against listening network services to attacks on client software such as browsers and document viewers. These changes have created a need for new defensive tactics. Many new attacks can no longer be found by using honeypots that host vulnerable network services and monitor for signs of compromise. Instead, researchers have proposed using *honeyclients*, which actively seek out malicious content on the web, to find previously unknown client-side attacks.

We note in passing that, like detection of malicious software, detection of malicious web pages is theoretically undecidable. We provide a proof of this fact, in the spirit of Fred Cohen [1], in Appendix A. However, such theoretical impossibility need not deter us; just as a large industry has grown up around virus detection, honeyclients are in widespread use today, and it is both useful and necessary to consider *practical* evasions against them.

Just as honeypots can be classified as high- or low- interaction, so too can honeyclients be categorized based on the extent to which they emulate a full end user system. HoneyMonkey [14] and the MITRE Honeyclient [8], which are both high-interaction honeyclients, use a full virtual machine with a web browser to visit potentially malicious sites; if the visit results in a persistent change to the state of the VM outside the browser, one can conclude that a successful attack has occurred. By contrast, low-interaction honeyclients such as honeyc [13] merely emulate a full browser, and use heuristics to detect attacks rather than monitoring for changes in the state of the OS.

Honeyclients have recently gained wide use in the Google Safe Browsing API [3]. As Google crawls the web, it uses heuristics to detect potentially malicious web sites. These sites are then fed to a custom honeyclient to verify that they are, in fact, malicious [11]. If a web page is found to host attack code, users who attempt to visit the page from Google's search results are presented with the warning shown in Figure 1. The user must ignore this warning to view the potentially malicious page.

Due to the prevalence of client-side attacks and the increasing use of honeyclients to detect them, a likely next step for attackers is to consider honeyclient *evasions* — techniques that cause a web page to exhibit benign behavior when visited by a honeyclient while still exploiting real users. In this project, we will present a number of such evasions, assess their utility to attackers, and consider possible defenses honeyclients can use to counter our attacks.

2 Related Work

Honeypots have long been used to detect and analyze novel attacks. Efforts such as the Honeynet Project [12] and numerous academic papers [10, 6] have described the use of both real and virtual machines to pick up on network-based attacks on vulnerable services. As the use of honeypots became more common, evasion techniques were developed [5, 7] that used techniques such as virtual machine detection to prevent executing in a honeypot environment. As attacks shifted to the client side, honeyclients were proposed as a means of detecting attacks. Systems such as Strider HoneyMonkey [14] use virtual machines running a vulnerable version Internet Explorer to detect malicious sites.

Despite the widespread use of honeyclients, honeyclient evasion has received little attention in academic literature. Although Wang et al. discusses evasions, the attacks they consider are either easy to counter, impose a significant cost on the attacker, or run the risk of alerting the user. By contrast, many of the evasions we describe are transparent to the user and do not harm the attacker's chances of success.

3 Methodology

Honeyclients and honeypots are two similar solutions to fundamentally different problems. A honeypot attempts to capture *push-based* infection, malware that is forced onto unsuspecting servers. Once a server becomes infected, it furthers the campaign by pushing its infection on to other hosts. A honeyclient, however, attempts to capture *pull-based* infection, which requires some human involvement.

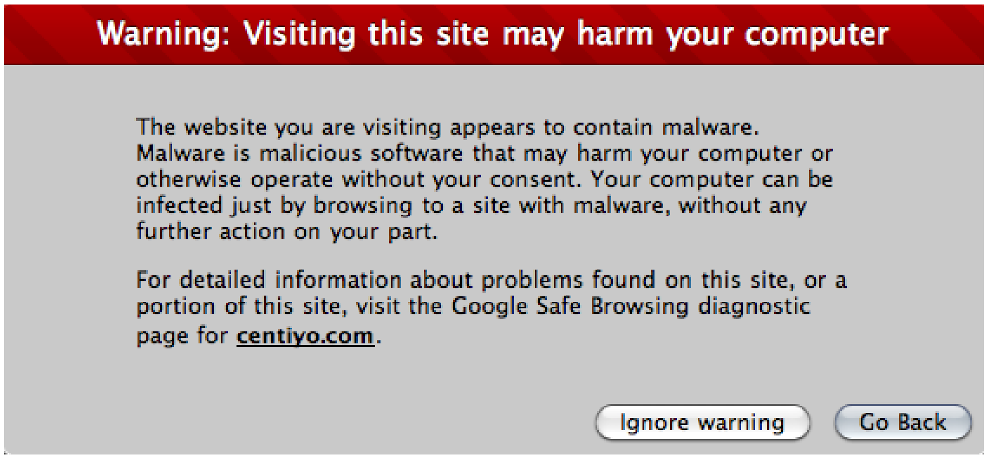
By detecting whether a client is a human or a honeyclient we can mount attacks on humans only, nullifying honeyclients as a tool for analysis. To demonstrate these attacks, we have designed and implemented a reverse Turing test that looks for a variety of features that would most likely be present in a true human client. In the interest of a conservative analysis, we have limited ourselves to ourselves to legal constructs in common languages seen on the web, such as HTML, JavaScript, Flash and Java. Our approach precludes, for example, the use of a security flaw in Flash to implement our reverse Turing test. This conservative approach also means that our evasions cannot be countered by patching known client-side flaws. If our features determine a client is a human, the exploit is launched. Otherwise, the site executes only benign code.

We have ranked our features by considering the following categories:

1. How difficult is the feature to counter?

2. How noticeable is the feature to a human client?

3. How costly is the feature's implementation for an attacker?


Warning: Visiting this site may harm your computer

The website you are visiting appears to contain malware. Malware is malicious software that may harm your computer or otherwise operate without your consent. Your computer can be infected just by browsing to a site with malware, without any further action on your part.

For detailed information about problems found on this site, or a portion of this site, visit the Google Safe Browsing diagnostic page for centiyo.com.

Figure 1: Google Malicious Site Warning

- Does the attack incur false positives by rejecting real humans?
- Does it increase the time required to execute the attack?

This ranking can be used by an attacker to determine the features to be used, and the order in which they should be executed to detect honeyclients. A subset of the features can be used depending on the attacker's needs. For example, if it is more important to prevent missed attacks on humans than defeating honeyclient analysis, a feature that introduces some false positives may be left out.

For each feature, we have created pages that demonstrate the evasion. When loaded by a human, client-side code on the page tests for some feature that indicates the presence of a human, and then performs an `XMLHttpRequest` back to the server with the test result. In a real attack, the server could then examine the result and, if the test indicates that the client is human, send the exploit code to the client. Thus, an attacker can avoid leaking valuable exploit code to a honeyclient, while remaining effective against human users.

We can evaluate our approach by comparing page execution behavior between the MITRE Honeyclient and a human client.

4 Attacks

Our attacks fall into two different categories: *state-based* and *behavior-based*. State-based attacks query the client's state to infer information about the user. For example, if the browser has recently visited sites like Facebook and Twitter, it makes it more likely that the client is an actual human, rather than a browser instance spawned from a VM snapshot. Behavior-based attacks interact with the user directly. CAPTCHA is a well known technique for differentiating humans and computers. Here, we use it in a similar fashion; if the CAPTCHA is properly solved, we know a human is in control of the browser and we can deliver our exploit safely. These two attack categories are shown in Figure 2. Both attacks occasionally rely on a predefined timeout or threshold that must be crossed to determine if an exploit should be launched or not. This values are in no way static, and will change as further testing is done to determine the "best" possible values. An overview of our attacks is presented in Table 1, with explanations of the intuition behind them presented in the subsequent sections.

It is important to note that all the attacks are written entirely in Javascript or Flash. All major browsers support Javascript and 99% of Internet-

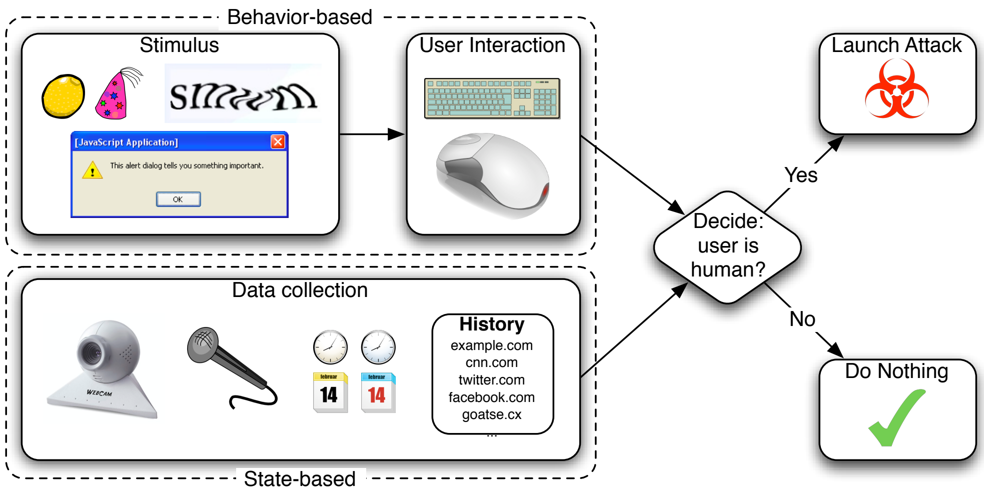


Figure 2: The two categories of evasions we describe. Behavior-based evasions use user interaction to distinguish between humans and honeyclients, while state-based evasions use only data gathered from the browser and operating system.

enabled computers have Flash installed[1], making these suitable languages to perform honeyclient evasion.

4.1 State-based Attacks

State-based attacks infer if a client is a honeyclient based on information harvested from the browser and operating system state.

Camera Intuitively, large-scale honeyclient web crawling will use virtualization for scalability and to make rollback after infection trivial. Customized honeyclient VMs can quickly be started from a snapshot, visit a malicious site, determine the guest OS's state was altered, and revert back to the original snapshot to scan the next potentially malicious site. These machines are unlikely to have a connected camera, a common feature on home and work computers.

Flash has built-in APIs to use an attached webcam. To use the webcam, Flash must first ask for the user's permission. However, Flash can determine if a webcam exists without requiring user intervention. If a camera is connected, we launch the exploit.

[1]"in mature markets"—see `http://www.adobe.com/products/player_census/flashplayer/`

Connection Speed Large companies who would benefit from honeyclient testing e.g., google.com generally have high throughput Internet connections to deliver fast and reliable services. As such, their honeyclient operations are likely to access the web with a very fast download rate. We determine the connection speed of the client by disabling the browser's cache, downloading a one megabyte file, and recording the time it takes to complete the operation. If the connection rate is below 1 MB/sec, we launch the exploit.

Browser History Honeyclients process potentially malicious URLs in a sequence in distinct sessions. If two potentially malicious websites were viewed in a row and the machine was compromised, it is unclear which site was responsible for subverting the machine. Thus, most honeyclients process malicious URLs in isolation from standard browser usage. Due to this behavior, it is unlikely honeyclients will have extensive browser history.

Attack	**State-based**	**Behavior-based**	**Description**
Alert		✓	An alert pop-up is displayed when the page renders. The exploit is delivered only if the alert box is closed.
Camera	✓		We query the flash plugin to check for the presence of a web camera. If one exists, we deliver the exploit.
CAPTCHA		✓	We deliver the exploit after successful completion of a CAPTCHA.
Connection Speed	✓		We determine the connection speed of the connected client and deliver the exploit if its below a predefined threshold.
Browser History	✓		We view the number of sites the client has visited from a set of popular websites. If the number of visited sites is above a certain threshold, we deliver the exploit.
Date	✓		We compare the day of the week of the client to the current day of the week in the United States (EST). If the days of the week match, we deliver the exploit
Immediate History	✓		We check the back button history for previous sites. If above a certain threshold exist, we deliver the exploit. This differs from the previous history attack in that it looks at an individual page or tab history, not the browser's history in its entirety.
Message Box		✓	We ask the client a randomly generated question with two possible answers using a dialog box. Based on the question, if the "correct" answer is chosen, we deliver the exploit.
Microphone	✓		Same as Camera, but checks for the presence of a microphone instead.
onMouseMove		✓	If the mouse is moved within the browser before a preset timeout, we deliver the exploit.
onBeforeUnload		✓	An offensive image is displayed. If the window is closed within a preset timeout, we deliver the exploit.
Plugins	✓		We query the browser for the existence of common browser plugins. If the number of usable plugins is above a predefined threshold, we deliver the exploit.
Referrer	✓		If the current client has a non-empty referrer string, we deliver the exploit.

Table 1: Attack Overview

Using a well-known method of accessing the browser's history[4], we check the client's history against a list of commonly visited URLs. If over 10% of the websites have been visited by the client, we launch the exploit.

Date Honeyclients that use virtualization will likely spawn fresh VM instances for each potentially malicious site view. If the VM is always spawned from the same snapshot, its current date will be when the snapshot was created and will not accurately reflect the true current date. We compare the current date (current year and day of the year) of the client with the current date in the eastern timezone of the United States. If the client is within 3 days of the actual EST US date, we launch the exploit. We obtain the correct date by querying a remote time server from Javascript.

Immediate History A human user is likely to stumble upon links to drive-by downloads by way of other websites. A honeyclient, however, generally has a list of URLs of potentially malicious websites that it visits directly. In the human case, we will see a clear series of visited websites, while in the honeyclient no such series will exist. This differs from Browser History as Immediate History only looks the back button history of the browser, while Browser History considers all stored history information.

In the event the client has three or more websites in its back button history, that is the client viewed the attack page through a path of direct hyperlinks containing three or more websites, we launch the exploit.

Microphone This attack is identical to Camera, except we query for the presence of a microphone. If one is found, we launch the exploit.

Plugins Browsers often extend functionality by way of plugins, like Flash. Plugins are commonly used for user digestible material, such as movies and online games. A honeyclient is unlikely to have multiple plugins installed. We check for the existence of the following plugins:

- RealNetworks RealPlayer
- Adobe Director
- Apple QuickTime
- Windows Media
- Adobe Flash

If the client has made querying specific plugins impossible, we consider the client to be a honeyclient and do not launch the exploit. If the client has at least three of the five plugins installed, we launch the exploit.

Referrer As discussed in Immediate History, a human user is more likely to visit a malicious page through direct hyperlinks rather than through a compiled list of URLs. We check for the existence of a referring website. If one exists, we launch the exploit.

4.2 Behavior-based Attacks

Alert A Javascript `alert()` window is shown to the user. A completely passive honeyclient will not interact with the loaded page, never launching the exploit. A curious user, however, will close the alert, triggering the exploit.

CAPTCHA As we discussed earlier, CAPTCHAs are commonly used on the Internet today as a way of differentiating a human from an automated bot. We assume the honeyclient lacks the capability to solve a complicated CAPTCHA and only launch the exploit in the event the CAPTCHA is correctly solved. It is interesting to note that at the time of the HoneyMonkey paper's writing[14], CAPTCHAs were uncommon on non-malicious sites and would likely raise suspicion in human users. The web today, however, makes liberal use of CAPTCHAs, nullifying the earlier assumption.

onMouseMove The Javascript event `onMouseMove` is bound to the page's document prototype in the DOM. This will capture any mouse movement that occurs within the context of the browser's window that displays the current page. If any mouse movement is detected, we flag the client as a human and launch the exploit. More sophisticated mouse movement tracking could be implemented to differentiate between smooth, human-assisted mouse paths and discrete, computer-assisted mouse paths. We discuss future enhancements in Section 7.

onBeforeUnload The Javascript event handler `onBeforeUnload` fires immediately before a browser window is closed. This attack has two key components:

- A negative stimulus to the client browser. In the current implementation, this is an offensive image.
- A function bound to the `onBeforeUnload` event handler to deliver the exploit.

The intuition behind this attack is a human user, when presented with an offensive image, will quickly close the window in a short period of time. A honeyclient, however, will not process and understand the offensive nature of the image, and will simply wait. We bind the event handler to a function responsible for loading attack code and present the user with the perenially popular Goatse image[2]. If the window is closed within 15 seconds, the exploit is launched. Otherwise, the event handler is unloaded to prevent accidentally exploiting a honeyclient.

Message Box A Javascript `confirm()` message box is presented to the client with a randomly generated question and two possible answers. One button launches the attack, while the other executes only benign code. To a human, the answer is readily apparent based on the random question. For example, the question "Do you want to do this awesome thing?" is answered "correctly" with the user responding "Yes". This reduces the likelihood of infecting a honeyclient to a coinflip but, in general, guarantees human infection.

5 Evaluation

Our evaluation is divided into two parts: attack obtrusiveness, and evasion effectiveness. In attack obtrusiveness, we discuss individual attacks that introduces a deviation from expected user experience. For example, an attack that uses a CAPTCHA is more likely to raise suspicion with the user than an attack which executes its test invisibly. In evasion effectiveness, we discuss how well our attack discerns a honeyclient from a desired human user. We address some limitations in our evaluation and ways to improve the analysis of our approach.

5.1 Attack Obtrusiveness

Understanding how obtrusive our attacks will be on a human client's browsing experience is an important metric to consider when choosing which attacks to use. We rate each attack based on two factors: does the attack require human interaction, as in a human interactive proof (HIP), and does the attack have potential to rouse suspicion. We present a summary of our evaluation in Table 5.1.

Attack	HIP	Suspicious
Alert	✓	✓
Camera		
CAPTCHA	✓	
Connection Speed		✓
Browser History		
Date		
Immediate History		
Message Box	✓	
Microphone		
onMouseMove	✓	
onBeforeUnload	✓	✓
Plugins		
Referrer		

Table 2: Summary of Obtrusiveness Evaluation

Transparent Attacks Some of our attacks operate transparently to the user, using only state information that can be gathered from client-side languages. These attacks include: camera, browser history, date, immediate history, microphone, plugins, and referrer. Camera and microphone both use an information leak that allows a malicious Flash executable to query the presence of devices without explicit user permission. Browser history uses an information leak in the bridge between Javascript and CSS that allows browser history to be inferred. Some attacks, like immediate history, plugins, date, and referrer use only standard DOM object accesses that are seen in many common web applications to mount successful honeyclient detection. The remaining attacks all introduce some kind of HIP or may make human viewers suspicious of the webpage's trustworthiness.

[2]`http://en.wikipedia.org/wiki/Goatse.cx`

Connection Speed The connection speed attack requires no HIP, but does introduce some overhead to download an additional large image. This overhead may be substantial on very low bandwidth systems, making the attack noticeable in low throughput situations.

The following attacks all use some kind of human interaction, whether direct or indirect, to determine the type of client.

Alert The alert attack requires direct user intervention to close a popup alert window. Javascript popups are considered irritating, but the propensity of humans to simply close the popup window will likely overcome any user suspicion.

CAPTCHA The CAPTCHA requires direct user intervention to interpret an image as text and submit a correct solution. CAPTCHAs have become very common on the web, and most users would not think twice before submitting the answer to a CAPTCHA to access web content.

onMouseMove onMouseMove requires indirect user intervention to determine if the attack should be deployed or not. We observe the standard user behavior of mouse movement to identify a client as human. The current honeyclient implementations do not simulate user input of any kind.

onBeforeUnload onBeforeUnload requires direct user interaction of closing the browser window to launch the attack. The user is shown an offensive image hoping she will quickly close the browser window in disgust. Since current (and future) honeyclients are unlikely to include complex image processing to determine the perceived offensiveness of an image, we use this to differentiate them from humans.

Message Box The message box attack requires direct user intervention to answer a question correctly to launch the exploit. As described before, the question will be simple enough for a human to solve, but difficult for a computer to without sufficient language processing capabilities. If the honeyclient resorts to chance, we still increase evasion to 50% from 0%.

5.2 Evasion Effectiveness

To test our attacks, we set up the Honeynet Project's honeyclient, HoneyC [13]. HoneyC consists of a number of Ruby scripts that access web pages and use predefined signatures to indicate successful attacks. Thus, it is a low-interaction honeyclient, as it does not attempt to simulate a full OS.

We ran HoneyC in the default configuration, using the ListQueuer module to visit our test pages. We then used the default Visitor module to visit each page, and passed the results to a Snort-based analysis module. To determine if the attack had been successfully launched, we configured the AJAX server to reply with a unique phrase ("wibble") when the attack code had been sent. In order to use HoneyC to detect successful attacks, then, we simply set up a Snort rule that searched for "wibble" in the incoming traffic.

As we expected, HoneyC was unable to detect any of our test pages as malicious. This is likely because of its poor support for JavaScript – as HoneyC was unable to execute any scripts on the page, it was unable to download the attack code (which was delivered via AJAX).

We ran the attacks on our personal machines to determine reasonable initial threshold values and to confirm our assumptions implicit in each attack. All the attacks successfully ran on our machines.

6 Defenses

Many of the evasions proposed in this paper can be countered by more careful emulation of a human attacker on the client side. Many of the state-based evasions we describe can be countered by taking care to prepare the honeyclient virtual machine in a way that mimics a real user's system. We describe mitigations that can be used to counter many of the attacks we have presented below.

Camera/Microphone
Modify the virtual machine to report that a camera and microphone are attached.

Referrer Check
Either spoof a legitimate referrer, or simply have the honeyclient navigate to the attack page from another site.

History
Visit other pages before going to the attack site.

Plugins
Install a number of commonly used plugins in the honeyclient.

Date/Time
Before visiting the attack site, synchronize the date and time with a centralized time server.

Mouse Movement
Instrument the VM to move the mouse over the attack page.

Connection Speed
Throttle bandwidth to the honeyclient VM, or simply run the honeyclient from a slower connection.

Although the defenses to these attacks are relatively easy to implement, they may adversely affect the scalability and performance of a honeyclient system. For example, performing time syncrhonization, moving the mouse, visiting other pages, and using a low-bandwidth connection will increase the amount of time required to test each page, and installing additional plugins in the virtual machine increases the amount of space required.

The behavior-based attacks are more difficult to counter. In particular, the interactive attacks (i.e., requiring that the user confirm a message box or solve a CAPTCHA) are likely to be impossible to detect automatically, as they require a human to read and understand the message presented. On the other hand, these attacks are extremely obtrusive, and run the risk of alerting users that something amiss. Likewise, attacks that involve unpleasant stimuli will easily evade honeyclients, but will quickly be detected if they are used on a high-profile web site.

7 Discussion and Future Work

There are a number of attacks we considered that did not turn out to reliably distinguish between humans and malware, or that were simply technically infeasible to test for. For example, we considered using Flash Local Shared Objects (better known as Flash cookies), which provide similar functionality to that of HTTP cookies, but can be accessed across domains. By checking for the presence of Flash cookies from popular web sites, we might be able to detect honeyclients by their lack of cookies. However, we discovered that while Flash does permit cross-domain cookie access, permission to access the cookie must be granted explicitly by the site that set the cookie, making enumeration of stored Flash cookies impossible.

Similarly, we had hoped that by loading popular sites that require authentication (such as Gmail) inside an iframe, we could determine whether a user was already logged into one of these sites. Because persistent login sessions are common among human users and unlikely to be found in honeyclients, this would have provided a simple test to distinguish the two. Unfortunately, it appears that many sites, and Gmail in particular, can detect when they have been loaded in an iframe and instead redirect the browser to the full site (with no iframe), which causes the browser to navigate away from the attack page. Worse, it appears that Javascript is forbidden from accessing the contents of iframes in other domains, so any determinations must be made on the basis of side channel features such as timing. Although we hope to explore this attack vector in future work, we were unable to find a way to use this in our current effort.

Looking forward, we can envision other types of attacks that may be both unobtrusive and extremely difficult to counter. Rather than simply checking for mouse movement, client-side code could track and analyze the path of the user's mouse cursor on the page to determine whether the movement corresponds to a human. This approach, which is known as a Human Observational Proof, has been proposed to detect bots in online games [2], and it may serve equally well to detect honeyclients. We hope to explore this attack, as well as possible defenses, in future work.

Finally, in addition to specific defenses, it is worth considering whether there are general techniques that could be used to expose the malicious functionality of web pages. With traditional malware analysis, a similar problem appears: given a malware sample, how can an automated system ensure that it has exercised all of the malicious functionality present in the executable? By applying techniques such as multipath exploration [9], it may be possible to explore code paths within web page content, allowing an automated system to trigger the malicious payload. However, the highly dynamic nature of common web languages such as Javascript and Flash may make the number of paths to be explored too high to be feasible.

8 Conclusion

We have presented a new attack that aids in evading honeyclient/HoneyMonkey detection of web pages with malicious downloadable content. This will serve two purposes, to delay both drive-by download website takedowns and malicious binary analysis. By determining whether a client is human or not, we can selectively target only humans to prevent early detection and prevention of compromising websites. We described and implemented 13 different attacks to detect a honeyclient and show they launch attacks against human users. We will supplement additional experimental information in the near future.

A Undecidability of Malicious Web Page Detection

Proof. Suppose that detection of malicious web pages is possible. Then there exists a detector H that can distinguish between malicious and non-malicious pages (for some non-trivial definition of "malicious"). Now, we can design a new page P that does the following:

1. Simulate running H on P. This can be done using Javascript, as Javascript is Turing-complete, and so it can be used to emulate H. For example, supposing H is a VM-based Windows system, one could implement a Javascript-based x86 emulator

2. If H decides that P is malicious, do nothing.

3. If H decides that P is benign, launch the attack.

By construction, P is malicious if and only if H says it is not malicious. This contradicts the assumption that H can distinguish between malicious and non-malicious pages, and hence H cannot exist.

□

References

[1] F. Cohen. Computer viruses: theory and experiments. *Computer Security*, 6(1), 1987.

[2] S. Gianvecchio, Z. Wu, M. Xie, and H. Wang. Battle of botcraft: fighting bots in online games with human observational proofs. Jan 2009.

[3] Google, Inc. Google safe browsing API. `http://code.google.com/apis/safebrowsing/`.

[4] J. Grossman. CSS history hack. `http://ha.ckers.org/weird/CSS-history-hack.html`.

[5] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, 2005.

[6] X. Jiang and D. Xu. Collapsar: a VM-based architecture for network attack detention center. In *Proceedings of the USENIX Security Symposium*, 2004.

[7] N. Krawetz. Anti-honeypot technology. *IEEE Security and Privacy*, 2(1):76–79, 2004.

[8] MITRE Corp. Honeyclient project. `http://www.honeyclient.org/trac`.

[9] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2007.

[10] N. Provos. A virtual honeypot framework. In *Proceedings of the USENIX Security Symposium*, 2004.

[11] 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 Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

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

[13] The Honeynet Project. Honeyc. `https://projects.honeynet.org/honeyc/`.

[14] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2006.