Department of Computer Science
George Mason University   Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/   703-993-1530

NetGator: Malware Detection Through Program Interactive Proofs

Brian Schulte bschulte@gmu.edu
Rhandi Martin rmartinl@gmu.edu
Haris Andrianakis candrian@gmu.edu
Angelos Stavrou astavrou@gmu.edu

Technical Report GMU-CS-TR-2011-6

Abstract

Exfiltration of data using internet-borne attacks has become a credible threat for organization and enterprises. History has shown that crafted targeted attacks and zero-day malware are capable of penetrating even the most sophisticated defenses. To make matters worse, intrusion detection systems that perform analysis of network traffic are dependent on the timely information provided by blacklisting, signature schemes, or anomaly patterns. This is especially true for heavily used communication protocols where blocking decisions affect the everyday operations of the organization. Even when the intrusion is detected in a timely manner, valuable data might have already been stolen.

In this paper, we propose a new approach to distinguish legitimate browser software from malware that generates identical network traffic signatures. Our system consists of two parts: a signature-based passive detection module, and a module that issues Program Interactive Challenges (PICs) to client applications. Initially, we perform passive detection utilizing content inspection techniques that analyze network header element order. Of course this is not enough: we demonstrate that passive detection can be easily subverted. To amend that, we introduce an interactive detection module in the form of inline network proxy that challenges the detected browser forcing it to exercise known specific internal functionality. For browsers, we can leverage HTML, Flash, Javascript and other common browser components to form challenges that malware will not be able to respond to due to lack of functionality. Contrary to interractive challenges, our challenges are transparent to the user allowing for a seamless browsing experience. We demonstrate the effectiveness of active challenges on thousands of real world malware samples. Our results show that, depending on the deployment scenario, PICs incure no or minimal (average of 0.35 seconds) latency per inspected objected.

1 Introduction

Sophisticated malware currently operates unhindered in the enterprise. The Anti-Phishing Working Group (APWG) reported in 2010 that more 48% of all computers are infected with some form of malware. Furthermore, sophisticated malware utilize obfuscation and polymorphic techniques that easily evade anti-virus and intrusion detection systems. The result is that security teams are forced to a losing position, as they are not able to gain a complete and timely picture of the malware operating within their enterprise.

Once inside the host, malware establishes command and control channels with external botmasters and drop points to exfiltrate data. To avoid detection, malware utilizes legitimate and usually unfiltered ports and protocols, such as HTTP(S), to establish these communications. By utilizing well-known ports and protocols, malware comms blend into the sea of legitimate application traffic traversing the network boundary. Due to the volume of network traffic, enterprises are unable to effectively monitor outbound HTTP traffic let alone encrypted HTTPs traffic and discern malware from legitimate clients. Current botnet detection systems focus on identifying the botnet lifecycle by looking for specific observables associated with either known botnets or typical botnet behaviors. These approaches suffer from the fact that malware continues to evolve in sophistication improving their ability to blend into common network behaviors.

Additionally, current systems are unable to inspect encrypted communications such as HTTPS leaving a major hole that malware will increasingly capitalize on. Note that the use of encrypted traffic has been growing as web applications begin utilizing HTTPS for its privacy benefits. For example, Facebook recently announced HTTPS as an optional protocol for accessing its site. While an improvement for privacy, the use of HTTPS poses major technical hurdles for current network monitoring and malware detection. NetGator is short for Network Interrogator and offers an entirely different approach to network-based malware detection and mitigation. Rather than attempting to classify network traffic as either good or bad based on packet, flow, or content inspection, as most current systems do, NetGator focuses on identifying the application behind the source of the communication. Indeed, NetGator can effectively and unobtrusively verify that the initiator is in fact the application that it is advertising to be and not malware code that attempts to disguise itself as a benign application.

To that end, NetGator operates as a transparent proxy situated in the middle of all conversation between clients and servers. NetGator employs protocol analysis to first identify the advertised client application type based on its network communication fingerprint. Next, NetGator validates that the app is in fact who it is advertising to be by issuing a challenge back to the client that exercises existing functionality of the legitimate benign application. The challenge is a small, automatically generated piece of data in the form of an encapsulated puzzle that a legit app will be able to process and automatically respond without any human involvement. For instance, a JavaScript packed HTTP redirect request required a JavaScript enabled application to be unpacked and executed. If the app is unable to solve or respond to the challenge, NetGator will flag the source as potentially being malware and optionally severe the connection and report the offending source. The proposed approach is an automated twist of the Human Interactive Proofs mechanism (e.g., CAPTCHAs), but focused on verifying program internal functionality rather than humans. As such, Invincea Labs has coined this approach as Program Interactive Proof or PIP.

The major challenge in developing robust malware detection solutions is to develop fast, scalable, real-time systems that can detect misuse, protocol deviation and malicious downloads. The contemporary approach to content-based client identification is easily defeated by falsifying browser configuration items or by setting the User-Agent field of HTTP request packets. Most active methodologies are in two flavors: offline systems that analyze traffic and make retroactive decisions, and online systems that disrupt communication by challenging the user, or by server side queries of client configuration files or components.

To address the inadequacies of contemporary detection systems, we implement a malware detection system using two major components. For passive detection, we focus on high-speed packet analysis. For our active module, we focus on client-side queries that are real-time, transparent, and do not disrupt communication. To this extent, our work contributes the following advances:

- We implemented an algorithm that utilizes a two-pronged approach to identify malicious traffic. We screen traffic with our passive detection module, as well as challenging the client with our active module.

- We incorporate passive client identification that analyzes the order of header elements in packets generated by a particular client.

- We introduce a novel architecture in which our gateway device challenges the application to prove itself. By our design principle, our system does not require user interaction.

2 Threat Model

2.1 Assumptions

We assume that a client machine is possibly infected with malware. It is also assumed that malware that infects the client machine establishes back channel communication with remote server(s). We consider that malware may not form connections immediately upon execution, but may wait for an indeterminate amount of time before initiating connection. Also, we assert that a certain subset of browser components are necessary to navigate the Internet and confine our challenges to these. Lastly, we assume that the sophistication of most current malware has not yet reached the level of implementing entire HTML, Javascript, or Flash engines within themselves.

2.2 Limitations

We are aware that we have not tested every browser; it is highly improbable that we could enumerate them all. Instead, we choose test clients that are popular and cross-platform to create a signature set that encompasses the majority of available clients. In controlled environments, like government networks, on which every application is known and approved, this approach is functional and results in low false positives. On open networks, like University networks, this approach would result in higher positive rates of unknown application detection.

3 Motivation

There are numerous anti-virus and firewall solutions available currently, but the question is how effective are they? Also, it is important to show that our concentration on HTTP traffic is well founded. We answer both of these questions by running a plethora of malware samples from Google and Malware Domain List. 1170 zero-day samples were obtained and were run in Windows virtual machines under two basic scenarios: 1) while host-based Symantec AntiVirus and Windows Firewall were active to ascertain the effectiveness of AV signatures and rule-based security; 2) with no host-based security to find which malware generate HTTP/S connections when executed. In order to get a broader understanding, we waited over various time periods to see if malware attempted communications. The results can been seen in Figure 1.

Out of the sample we used, nearly half of the malware used HTTP/S for outbound communications. This shows that there is a great need for a system that can thwart malware's HTTP/S connections. Aside from the fact that malware used HTTP/S, 75% of those type of connections were unclassified by Norton AntiVirus and Windows Firewall which tells us that current technologies, while not completely ineffective, are not up to par.

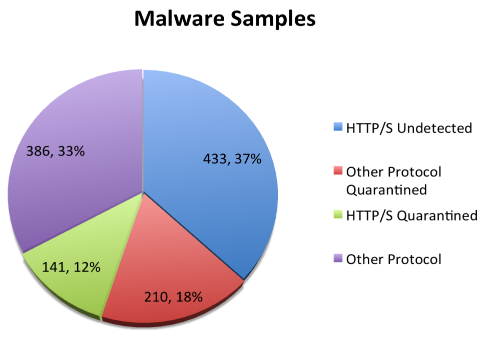


Figure 1: Study of Zero-day Malware

4 Background

In order to design an effective system to efficiently detect and prevent malicious traffic from leaving the confines of the infected system, we need to understand the various elements of HTTP transmissions and how they operate. The understanding of the HTTP protocol allows us to construct an architecture that can accurately make use of the protocol in order to best detect and thwart malicious agents' attempts at calling out.

4.1 HTTP Headers

HTTP request and response packets begin with various header elements contains pertinent information about the transmission. Requests are prefaced by various headers notifying the server of what the client expects to receive. Before the headers begin, the very beginning of the packet contains the request method such as GET or POST. Responses are accompanied by headers as well informing the client of what is being transmitted. Similar to request packets, the response code precedes the response packet's headers. This response code broadcasts the result of the request made by the client. Common codes include "*200 OK*" meaning the file requested was returned properly and "*404 Not Found*" indicating the file could not be found.

4.2 MIME Types

Multipurpose Internet Mail Extension(MIME) types describe the content type of the message being transmitted. This is especially important to us due to the fact that we want to confine our different challenges to particular messages based on their MIME types. There is a plethora of specific MIME types, but there is only a minimal amount of general types. For instance, the MIME type application/x-compressed represents a .tgz file but the application type is the general type which represents a large set of files. The main general MIME types are application, audio, image, text, and video. The main distinction we have to make in our system is between data which is labeled text/html and which is not. Text/html is the MIME type of most webpages and therefore allows us to better target our challenges to the appropriate HTTP transmissions.

5 Related Work

There exists a body of work in application fingerprinting and malware detection, primarily, for the purpose of distinguishing legitimate traffic from malware communication, and for policing file downloads. The most popular form of real-time browser challenges is to utilize server-side techniques that read browser configuration files[12][15] (Javascript, ASP, etc.), cookie information[11], or search for platform specific components like flash blockers or Silverlight[3]. Another approach is to search traffic flows for known, specific identifiers like connections to Firefox update servers[19]. Conversely, techniques like the well-known CAPTCHA puzzles prove the existence of a human user. However, such methods are disruptive and overt which is contrary to our design principle.

For our implementation, we prefer a combination of these approaches, a real-time traffic analysis of message requests. Our passive detection system is similar to both methods in that it is non-invasive and does not disrupt communication. Unlike previous systems, ours moves the detection mechanism closer to the source. We also add the focus of preventing data exfiltration. However, analyzing the existence and order of header elements is not a new concept.

Our work was partly inspired by various automatic protocol analysis systems[10][18]. While not similar to our work in desired outcome, the overall architecture of various automatic protocol analysis systems are very similar to ours. These systems utilize injection of messages to various applications in order to automatically determine how a particular protocol is organized. We are not injecting messages to test a protocol, but rather to examine a particular application if order to prove identity.

For malware detection, a great body of research is concerned with policing file downloads. Congruent with this reasearch is to identify what services or applications are active on a network, in order to distinguish malware from legitimate use. One approach is to search traffic flows for known, specific identifiers like connections to Firefox update servers[19]. Another is to analyze the existence and order of header elements.

While these passive and active detection mechanisms may be sufficient for the current state of malware, we contend that they raise the bar to a degree that is technically trivial to overcome. As such, our active modules challenge the necessary, and some optional, components and functions of browsers, in order to verify the legitimacy of requesting applications. In addition, we have the added benefit of disrupting botnet Command-and-Control(C&C) channels, and preventing spam and data exfiltration. We have not encountered any work that mirrors ours in concept, implementation or completeness.

5.1 Botnets

Traditionally, botnet detection and mitigation systems like BotSniffer[6] have focused on zombies that contact Internet Relay Chat(IRC) C&C servers or utilize IRC-style communication[1]. Unfortunately, botnets have grown in sophistication to use Peer-to-Peer (P2P) and unstructured communication[2, 8]. In addition to the traditional techniques such as blacklisting, both signature and anomaly based detection, and DNS traffic analysis, BotHunter[5] proposes using infection models to find bots, while BotMiner[4] analyzes aggregated network traffic. However, these approaches are inadequate for disrupting non-IRC botnets and C&C channels. When data is contained completely within the header elements of a packet, the greater the amount of data that is embeded, the more likely a malformed packet is created[17]. C&C control messages can be very small, and easily embedded in a request packet, therefore, the best method for disrupting C&C messages is to prevent malicious requests packets from arriving at the destination.

5.2 Data Exfiltration

In addition to HTTP request packets, data streams are more often used to convey data. For analyzing packets that contain payload, deep packet inspection techniques are favored. To these packets, signature- or anomaly- based detection is applied[2]. To foil this mechanism, malware may use the same secure protocols that users employ to protect themselves from malicious agents[13, 7]. Our approach does not analyze the data, but questions the client that is sending or requesting data. Due to this difference, we are able to thwart zero-day malware as well as not requires any machine learning. Furthermore, we implement systems that allow us to analyze even the secure communications. We consider that both methods may be used in an overarching system. We envision that our PIC module would pre-screen traffic for the deep inspection module.

5.3 SPAM

Spam is widely regarded as the most notorious Internet nuisance. The favored approaches for combating spam are through IP and behavioral blacklisting known-bad hosts, and through data stream analysis[14, 16, 9]. We handle spam in the same manner as we do for information leakage; we prevent the initial communication from completing by challenging the client.

6 Architecture

Our design principle is to keep our system online and transparent to the user. It does not require a human to prove themselves, but shifts the onus of proof to the requesting application. Since the User-Agent field in browser requests is easily altered, we cannot trust that a request is not generated by malware masquerading as a particular client. We also cannot depend on blacklists to provide timely information in zero-day attack situations. Therefore, our modules utilize network-level transparent proxies and non-disruptive queries to analyze network traffic and probe applications. While the bar is raised with our inspection of packet header ordering, it is a trivial task for an attacker to craft a perfectly formed GET or POST request. Therefore, the rest of the paper is concentrated on the active testing, rather than the passive.

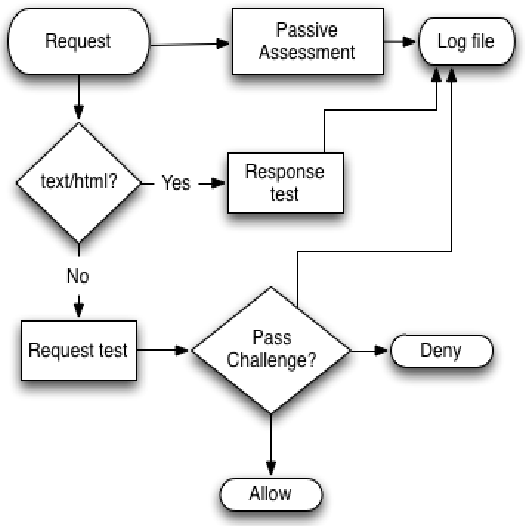


Figure 3: Flowchart of a Client's Request Through Our System

The infrastructure that we have architected consists of a single system acting as a transparent proxy which resides on the network. All traffic from machines on the network that are to be inspected is routed to this proxy. The proxy in turn performs any processing necessary and then, if the client has passed the challenge, forwards the traffic to the original default gateway of the network. The flow of packets through our architecture is depicted in the flowchart in Figure 3.

A transparent proxy and Internet content adaptation protocol (ICAP) server make up the software duo that resides on the proxy. The proxy software passes all port 80 and port 443 destined traffic to the ICAP server which then handles the processing of our active challenge. If the client needs to be challenged, the ICAP server handles inserting the proper code into the original response, either rewriting it completely, or simply inserting it into the existing code. This decision is based on the Multipurpose Internet Mail Extensions(MIME) type of the requested data; if the response is any type of non text/html data, the request is blocked and a completely new response containing only our challenge is sent back to the client. For text/html types, the challenge code is inserted inside of the orignal response.

7 Implementation

In our active module implementation, we wish to inject challenge code into a response the client receives that invokes a function or component of the client. The challenge is triggered by finding GET or POST in the message header, however, packets encrypted by unknown keys cannot be decoded and parsed. In order to receive cleartext headers for all web traffic, our analysis module consists of a proxy that can terminate HTTPS communications. In summary, our browser challenge system consists of:

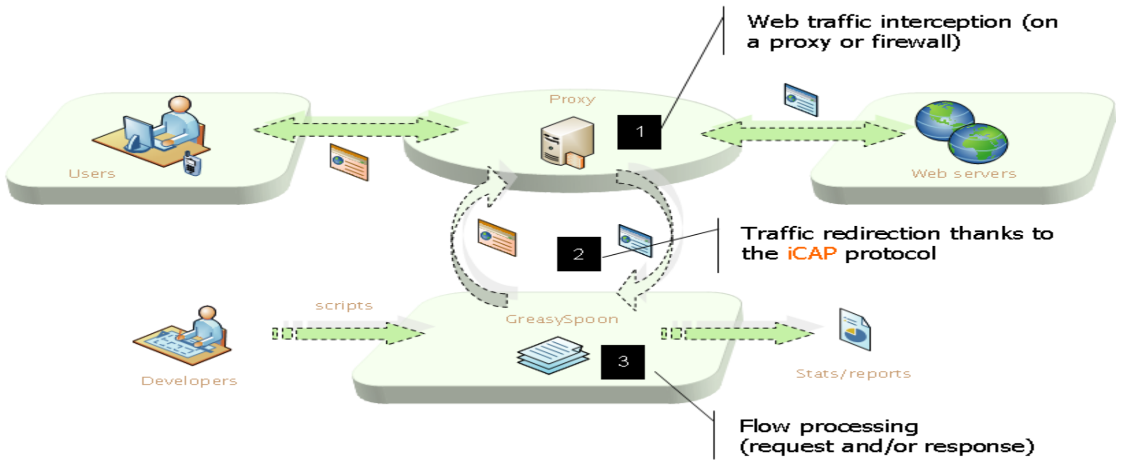

Web traffic interception (on a proxy or firewall)
Proxy
1
Users
Web servers
Traffic redirection thanks to the iCAP protocol
2
scripts
GreasySpoon
3
Developers
Stats/reports
Flow processing (request and/or response)

Figure 2: Proxy-ICAP Server relationship

1. Squid 3.1.8 - an open-source proxy for HTTP, HTTPS and FTP traffic

2. Greasyspoon - an open source, Java-based ICAP server with an API for running scripts on requests and responses

All traffic that has a destination of port 80 (HTTP) or port 443 (HTTPS) that is received by the proxy machine is rerouted to the appropriate ports for the Squid proxy. Once Squid receives the traffic, it is then passed to the ICAP server for processing. The overall architecture of this system is illustrated in Figure 3. In order to handle HTTPS traffic, it is encrypted with the proxy's key. Once the traffic reaches the proxy, it is decrypted for any processing necessary (ICAP scripts) and then is re-encrypted on it's way out to the intended target.

Depending on what type of data is being requested, the client is challenged either at the request or the response. For any HTTP response in which the data is not text/html, the client is challenged at the request, which blocks it until the challenge is completed. If the data requested is text/html, the challenge is inserted inside the response allowing it to pass through. In order to keep track of the various connections Greasyspoon's cache, which contains a hashmap, is levereged. This hashmap contain entries for each IP address + user agent pair that is seen with information about it such as whether it passed the challenge or not. It also stores how many times the particular client has been challenged as well as how many times it passed the challenge. With this mechanism in place, even if the malware correctly forms a user-agent string that is operational on the infected machine, the hashmap will still reflect that an entity on the system did not pass the challenge. The hashmap is periodically written to a log file available for inspection.

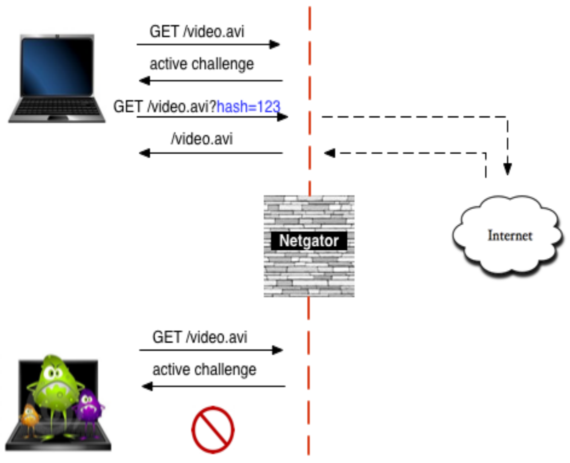
In order to avoid challenging the same client an unneccesary amount of times, a record of which top level domains a particular proven client has visited is kept. This way, when the proxy sees a request, it first checks if the client has already passed the challenge. If it has, it then lets the request pass if the top level domain has already been visited by the client. In other words, if a client tries to access *www.foo.com/bar* and has already proven itself while requesting *www.foo.com*, the proxy will let it pass automatically. This enables us to lessen any further burden from websites that trigger many GET requests for items such as images or flash objects. This way, we can have an even lower overall overhead for users.

7.1 Request Testing

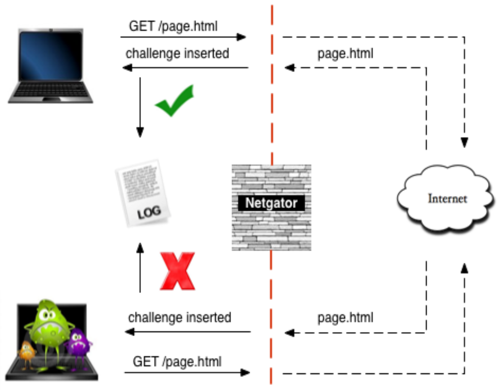
When the proxy sees a request for non-text/html data, it issues a completely new response to the client in order to challenge it. The challenge can take various forms based on what the network administrator deems appropriate. Whichever test is administered, the driving element behind each of them is a redirect to the original requested URL with a hash appended to it. The only challenge that needs to contain more than a simple redirect command is the Flash challenge. This is due to the fact that the challenge is a Flash object, not actual lines of code. In order to correctly pass the requested URL and hash to the Flash object, Javascript functions are returned in the HTML code to interact with the object and provide it with the redirect information needed. If the client is able to correctly execute the challenge, Greasyspoon will then see a new request for the originally requested URL with the hash appended as a parameter. If the hash is correct, the request is allowed to pass while sending back an error message to the client if the hash is incorrect. The implementation for the request challenging can be seen in Figure 4. Once the request has been seen with the correct hash, Greasyspoon updates the hashmap to reflect that that particular IP and user agent combination has passed the challenge.

7.1.1 Request Script

The responsibility of Greasyspoon is to intercept the connection when it observes a request and send back a custom crafted response to the client in order to initiate the challenge. In order to correctly form the response to be returned, the ICAP server executes a Javascript program. The two undertakings



(a) Request Challenge Flow



(b) Response Challenge Flow

Figure 4: Request and response challenge architectures

of the script are to generate the hash and then craft the new HTML code the client will see. First, to generate the hash, a combination of four factors is used: a static, secret key known only to the proxy, the requesting client's IP address, the URL being requested, and the current time's seconds value. The combination of these elements is then hashed using SHA1 and then consequently inserted into the HTML code to be sent back to the client.

The second task for the request script to perform is to replace the header and the body of the request in order to form the custom response that is to be sent back to the client. The header must be replaced with a properly formed HTTP response header to signal to the ICAP server that a response is required to be sent back to the client directly from the proxy. For our implementation we use a standard HTTP/1.1 200 OK response. The HTML code that is inserted is simply a fragment of Javascript code executing a redirect. Once the custom crafted HTML code is sent back, the hashmap entry is updated to reflect that a challenge has been sent to the current requesting client. The various versions of code that is returned from the proxy is shown in Figure 5.

```
<html>
<head>
<script type ="text/javascript">
window.location={URL requested}?{hash generated}
</script></head>
</html>
```

(a) Javascript Challenge Code

```
<html>
<head>
<meta http-equiv='Refresh' content='0;
     \url={URL Requested}?={hash generated}'>
</html>
```

(b) HTML Challenge Code

```
...code to set up flash embedding...
<script type="text/javascript">
var params = {allowScriptAccess: "always"};
function GetURL(){return {URL requested};}
function GetHash(){return {hash generated};}
swfobject.embedSWF(\
      "http://{Gateway IP}/flashtest.swf",\
      ...size parameters..., params);
</script>
```

(c) Flash Challenge Code

Figure 5: HTML returned by proxy for the various challenges

The code for the Flash and HTML challenges are essentially identical, each containing a redirect function to the originally requested URL with a hash concatenated to it. If the client is able to correctly execute the challenge code, the proxy will see a separate request with a hash appended to it. If the hash is correct, the new request is allowed to pass through and the hashmap of Greasyspoon is updated to reflect that the client passed the challenge. The size of the request scripts each average around 280 lines of code.

7.2 Response Testing

The reason for response testing is two-fold: it allows us to diminish the overhead that might be introduced from enacting the request challenge on each HTTP request and also for the case where a malicious agent attempts to download an executable that is disguised as a .html file. If we were to block at every data request, that would impact scalability and also the user's experience. It is conceivable that on a smaller, more confined network the system could be setup to challenge every request, but on a larger infrastructure this would most likely be impractical. If the data requested is of the type text/html, the proxy lets the request pass through. When the response comes back for that connection the challenge code is then placed in the response. For this type of testing, an image that resides on the proxy is embedded in Javascript code. The proxy then looks for requested for this specific image and once it sees one, it then knows that the challenge has been completed. This implementation can be seen in Figure 4.

```
Record number=1
IP=192.168.0.2
OS=Windows XP
App_Name=Firefox
App_Version=3.0.5
UA=Mozilla/5.0 (Windows; U; Windows NT 6.1; fr;
rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
Challenges_Issued=587
Challenge_Responses=587
```

Figure 6: Example entry from the logfile

7.2.1 Response Script

The response script is actually two part: a script that processes on the request and a script that is called when the response is seen with them combining to be about 300 lines of code. The initial step in the process is to determine if the client is expecting a text/html response or if the request is for our specific challenge image. If it is not, the response challenge is not activated. If the request is for our challenge image, Greasyspoon searches for an already established entry in the hashmap and updates it reflecting that the client has passed a challenge. If the client is simply expecting a text/html response, an entry of the user-agent string and client IP is written into the hashmap showing that it is in the state of a request being seen. The original request then goes out to the intended server. Once a response for the connection is seen, the response script is called. This script probes for an already present entry in the hashmap for the client the response is to be sent to. If an entry is located, it is revised to show that a challenge has been sent to the client. It then injects the HTML code inside the original response from the queried server to imbed our challenge image and sends the response back to the client. The response infrastructure is also responsible for the transformation of the hashmap into a logfile format. An example entry from the logfile is shown in Figure 6. The operating system, application name, and application version are all extracted from the user-agent string. This is done to enhance the engineer's ability to analyze the logs and diagnose a problem should one arise.

8 Experimental results

The hardware we used to perform the tests consisted of a laptop for the client and a Dell server for the proxy. The laptop was a Dell Latitude E6410 with an Intel Core i7 M620 CPU at 2.67 GHz, 8GB of RAM and a gigabit network interface. Firefox 3.6.17 was used as the client's browser throughout testing. The server was a Dell PowerEdge 1950 with two Xeon processors, 16GB or RAM and a gigabit network interface. In order to assess the impact of a user on a network employing our system, we measured the overhead that each of our tests introduced to HTTP connections. We also desired to establish how efficiently the server can process the scripts that are being executed on the client's request. In order to accomplish that a script was designed to loop through 10000 iterations of the request script with the iterations per second being returned. An average of this script being run 30 times was taken to

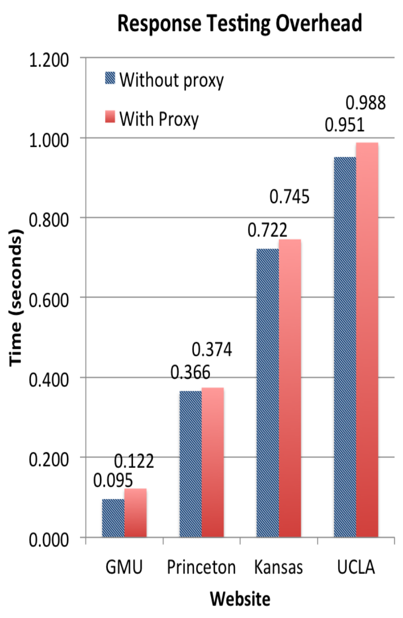


Figure 7: Overhead from response challenge

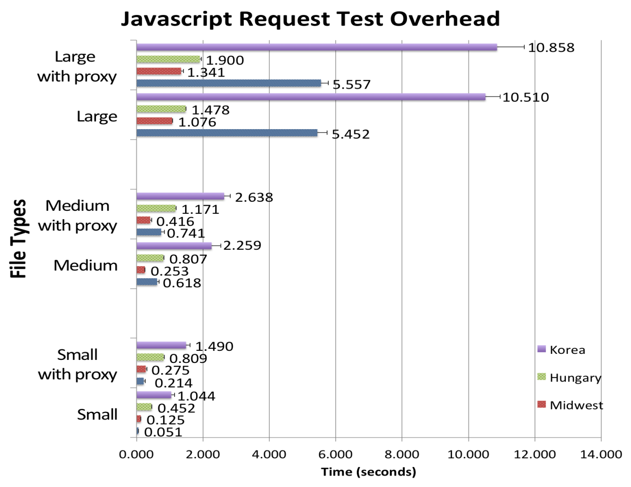
determine the capability of our server in processing scripts. For all testing, Squid's caching mechanism was disabled as well as all of Firefox's caching.

8.1 Response testings results

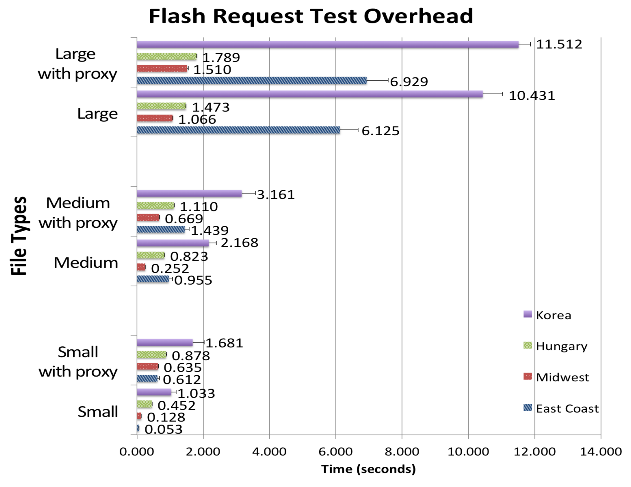
In order to test the overhead of our response challenges, we performed tests loading various Computer Science Department websites throughout the country. Baselines were established for each website by performing a simple loading of each of them without the proxy involved. Once these baselines were established, the gateway of the client laptop was changed to be our proxy. The websites were once again loaded which introduced our response test to the interaction. Each website was loaded thirty times both with and without the response testing. In order to establish time, the difference in between the time-stamp of the first and last packet in the stream was taken. The results of these tests are shown in Figure 7.

8.2 Request testings results

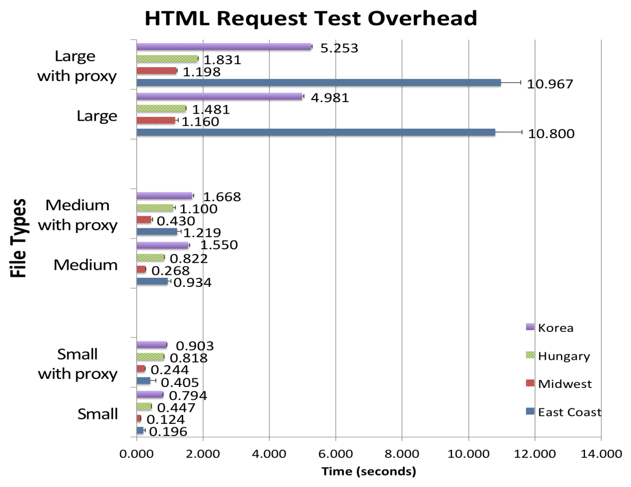
To evaluate our request testing architecture, we tested the various types of challenges we use in download scenarios utilizing PlanetLab. Various PlanetLab nodes were used from throughout the world using all virtualized hardware. Nodes residing in Virginia, Kansas, Hungary, and South Korea were utilized to perform the benchmarking. Executable files of various sizes (10KB, 100KB, and 1000KB) were hosted on each node on an Apache web server. The client then performed the downloading of each file thirty times from each of the nodes, both with and without the proxy. The values of time were determined by the difference in the time-stamp of the packet that started the initial request before the challenge and



(a) Javascript challenge



(b) Flash challenge



(c) HTML challenge

Figure 8: Request Challenge Overhead

the last packet that closed the connection after downloading the file. We performed experiments to determine the overhead of the Flash, Javascript, and HTML challenges shown in Figure 8.

8.3 Analysis

Our testing shows that the overhead introduced by any of our challenges is completely negligent to the user. For the more basic Javascript request test, the average overhead was only 0.274 seconds while the more challenging Flash request test had an average overhead of 0.580 seconds. The most basic test of an HTML redirect introduced only 0.206 seconds of overhead. The request challenge results show that the overhead remains the same across various file sizes; this means that in terms of percentage the overhead gets progressively smaller as the files downloaded become larger. The response challenge results are even smaller with an average overhead of only .024 seconds. With the minimal amount of overhead that our system introduces, it is unperceivable to the user. Also, we saw that our proxy was exceedingly efficient in processing the scripts, being able to handle on average around 1200 request scripts per second.

It is of worth to note that our infrastructure can actually have a positive impact on performance on the network of which it is deployed. This is due to Squid's caching mechanism. For our testing, we were forced to disable the caching of Squid to get a true measurement of the overhead introduced from the scripts being processed. If utilized on a powerful server of which Squid's caching can take full advantage the network could see performance increases for commonly visited websites or downloaded files.

During out testing to establish what percentage of malware calls out utilizing HTTP/S, we found that none of the malware which used either protocol could overcome our challenge architecture. That sample size equates to 574 malware samples challenged. The typical behavior of the infected systems would simply try to re-request the file it had originally sought after only to repeatedly be returned our challenge. The only false positive we observed in our testing was the use of Wget. However, our testing atmosphere was limited and comprised of a select group of user agents and did not mimic an enterprise system. It is possible that on an implementation of that scale that increased false positives may be observed due to various agents without full browsing capabilities requesting files via HTTP/S .

9 Conclusions

In this paper, we introduced a system for identifying malware downloads and malicious transmissions. We divide this system into two subsystems that are real-time and transparent to the user. Our first phase is a passive detection module that analyzes request packets, and relies on the consistency in header element order to apply our signatures. We analyze HTTP and HTTPS transmissions using regular expression pattern matching to determine the legitimacy of a client. Our passive system does not rely simply on the content of the User-Agent field, nor does it query any components of the browser. It is the job of our active challenge subsytem, PIC, to query a client on particular attempts to download or send information. It requires the application to call functions that should exist for legitimate conversation. Failing this, the communication is broken.

First, for all our test clients, we derived signatures that were unique where possible. Next, we demonstrated the worth of our approach by quantifying the effectiveness of other anti-malware mechanisms on zero-day malware. Then, we showed how feasible our active approach was for HTTP/S communications. Specifically, we broke the connections of any application that could not prove its identity. Our results confirm that PIC is a low-latency, lightweight solution. In our worst-case simple communication tests, PIC introduced approximately 1s overhead for our flash variant of the challenge while maintaining an average of 0.353s across all of the request tests and .024s on the response testing.

References

[1] B. AsSadhan, J. Moura, D. Lapsley, C. Jones, and W. Strayer. Detecting Botnets Using Command and Control Traffic. In *Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications-Volume 00*, pages 156–162. IEEE Computer Society, 2009.

[2] M. Bailey, E. Cooke, F. Jahanian, Y. Xu, and M. Karir. A survey of botnet technology and defenses. In *Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security-Volume 00*, pages 299–304. IEEE Computer Society, 2009.

[3] P. Eckersley. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[4] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *Proceedings of the 17th conference on Security symposium*, pages 139–154. USENIX Association, 2008.

[5] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. Bothunter: detecting malware infection through ids-driven dialog correlation. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[6] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS08)*. Citeseer, 2008.

[7] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao. Malware Behavior Analysis in Isolated Miniature Network for Revealing Malware's Network Activity. In *Communications, 2008. ICC'08. IEEE International Conference on*, pages 1715–1721. IEEE, 2008.

[8] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, page 7. USENIX Association, 2007.

[9] T. Kusumoto, E. Chen, and M. Itoh. Using call patterns to detect unwanted communication callers. In *Applications and the Internet, 2009. SAINT '09. Ninth Annual International Symposium on*, pages 64 –70, 20-24 2009.

[10] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through conectect-aware monitored execution. In *In 15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[11] K. McKinley. Cleaning Up After Cookies, 2008.

[12] M. D. Network. How to: Detect browser types and browser capabilities in asp.net web pages, 2010.

[13] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, page 26. USENIX Association, 2010.

[14] A. Ramachandran, N. Feamster, and S. Vempala. Filtering spam with behavioral blacklisting. In *Proceedings of the 14th ACM conference on computer and communications security*, pages 342–351. ACM, 2007.

[15] W. Schools. Javascript browser detection, 2010.

[16] R. C. University and R. Clayton. Stopping spam by extrusion detection. In *In First Conference on Email and Anti-Spam (CEAS), Mountain View, CA*, 2004.

[17] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection*, pages 203–222. Springer, 2004.

[18] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *In 15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[19] T.-F. Yen, X. Huang, F. Monrose, and M. Reiter. Browser fingerprinting from coarse traffic summaries: Techniques and implications. In U. Flegel and D. Bruschi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5587 of *Lecture Notes in Computer Science*, pages 157–175. Springer Berlin / Heidelberg, 2009.