NetGator: Malware Detection Using Program Interactive Challenges

Brian Schulte, Haris Andrianakis, Kun Sun, and Angelos Stavrou

The Center for Secure Information Systems,
George Mason University,
4400 University Drive, Fairfax 22030, USA
{bschulte,candrian,ksun3,astavrou}@gmu.edu
http://csis.gmu.edu

Abstract. Internet-borne threats have evolved from easy to detect denial of service attacks to zero-day exploits used for targeted exfiltration of data. Current intrusion detection systems cannot always keep-up with zero-day attacks and it is often the case that valuable data have already been communicated to an external party over an encrypted or plain text connection before the intrusion is detected.
In this paper, we present a scalable approach called *Network Interrogator (NetGator)* to detect network-based malware that attempts to exfiltrate data over open ports and protocols. NetGator operates as a transparent proxy using protocol analysis to first identify the declared client application using known network flow signatures.Then we craft packets that "challenge" the application by exercising functionality present in legitimate applications but too complex or intricate to be present in malware. When the application is unable to correctly solve and respond to the challenge, NetGator flags the flow as potential malware. Our approach is seamless and requires no interaction from the user and no changes on the commodity application software. NetGator introduces a minimal traffic latency (0.35 seconds on average) to normal network communication while it can expose a wide-range of existing malware threats.

1 Introduction

Targeted and sophisticated malware operates unhindered in the enterprise network. Indeed, the Anti-Phishing Working Group (APWG) [1] reported that the first six months of 2011, data-stealing malware and generic Trojans increased from 36% of malware detected in January, 2011 to more than 45% in April, 2011. Sophisticated malware utilizes obfuscation and polymorphic techniques that easily evade anti-virus and intrusion detection systems. A study by Cyveillance[11] showed that popular anti-virus solutions only detected on average less than 19% of zero day malware increasing only to 61.7% on the 30th day.

Once inside the host, malware can establish command and control channels with external points of control, often controlled by a single entity called a botmaster, and form drop points to exfiltrate data. To avoid detection, malware

utilizes legitimate and usually unfiltered ports and protocols, including popular protocols such as HTTP, to establish these communications. Due to the volume of network traffic, enterprises are unable to effectively monitor outbound HTTP 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. 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.

In this paper, we demonstrate the results of a novel approach called *Network Interrogator (NetGator)* to detect and mitigate network-based malware that (ab)uses legitimate ports and protocols to initiate outbound connections. NetGator operates as a transparent proxy situated in the middle of all network communications between the internal clients and external servers. Our approach consists of two phases that are real-time and completely transparent to the user. In the first phase, we employ a passive traffic classification module that performs protocol analysis to first identify the advertised client application type (i.e. browser, program update, etc.). We do so based on existing network signatures for legitimate applications including the use of ordering of traffic headers that sometimes characterizes the host application. The main purpose of this first order flow classification is to determine the claimed or declared identity of the end-point software that generated the traffic into two classes: potentially legitimate or unknown.

As a second phase, for flows that we can successfully classify as part of potentially legitimate applications inside the organization (e.g., approved browser using HTTP(S) on port 80 or 443), Netgator attempts to further probe the end-point application by inserting itself as part of the network communication.We do so by issuing a challenge back to the client that exercises existing functionality of the legitimate application. A challenge is a small, automatically generated piece of data in the form of an encapsulated puzzle that a legitimate application can execute and automatically respond without any human involvement. If the application is unable to correctly solve and respond to the challenge, NetGator will flag the source as potentially being malware and optionally sever the connection along with reporting the offending source. Therefore, rather than attempting to classify network traffic as either good or bad based on network (packet, flow, or content) inspection, the second phase of NetGator focuses on validating that the traffic stems from a legitimate application.

The proposed approach is an automated twist of the Human Interactive Proofs mechanism (e.g., CAPTCHAs), but focused on verifying program inter-

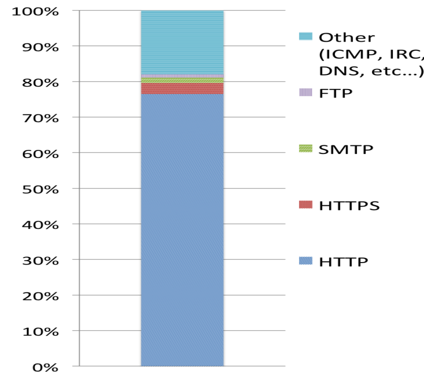


Fig. 1. Study of 1026 samples from popular classes of zero-day malware.

nal functionality rather than humans. We call this approach Program Interactive Challenges (PICs). We define a PIC as a challenge comprised of a request and expected response pair which tests for existing functionality of legitimate applications. A PIC can be generated when there is an end-program state that has a deterministic programmable network response and that state can be triggered by communication with the server. The complexity of the PIC depends on the complexity required to implement that state on the client side. The intuition is that if the challenges are diverse enough and exercise complex functionality of the legitimate applications, malware will have to either implement said functionality making it large or attempt to create application hooks or use the legitimate application to "solve" the puzzle. In the former case, the malware code will increase dramatically since malware has to now implement a lot of unnecessary and complex functionality, for instance, a JavaScript parser. In the latter case, the malware will have to farm out the traffic to the corresponding legitimate local end-point application to solve the puzzle. In addition, the malware will have to insert itself after the puzzle exchange while suppressing traffic from the legitimate application. Our approach raises the bar because it forces malware to perform additional invasive operations that would not be required without our system. It is not enough for the malware to just link to Browser libraries that implement communications, the malware has to also take over the HTML, Javascript, and Flash rendering engines. Therefore, NetGator increases the attack complexity for the adversary without requiring any human involvement.

We tested our system on 1026 zero-day malware samples in Windows virtual machines. Our experiments show that the majority of malware uses popular, unfiltered network ports to connect to remote servers for various purposes. Figure 1 shows that nearly 80% of the malware samples used HTTP/S for outbound communications. Therefore, we focus on developing PICs for browsers to show the effectiveness of our method. We can leverage HTML, Flash, Javascript, and other common browser components to form challenges for browsers. However, our approach is more general and can be extended to generate and use PICs for other applications (e.g., VoIP, OS updates) through analyzing the functionality sup-

ported by the application software agents. Second, most malware only includes minimal functionality to reduce its size and avoid being detected, so it cannot correctly respond to the challenges. For example, many malware scripts use the "*wget*" command to download malicious code from external servers without compromising the browsers in the OS. Such scripts do not know how to respond to the PICs for the browsers. If a large-size malware includes all the challenging functionality for a browser or compromises the browsers in the OS, it can defeat our solution; however, we increase the bar for attacks to succeed.

We implemented a prototype NetGator system that includes different PICs for browsers. For non-text/html data, NetGator issues challenges when it receives the request packets from the client, which we refer call *request challenges*; for text/html data, it issues challenges when it receives the response packets from the external servers, which we call *response challenges*. Compared to request challenges, response challenges can reduce the overhead that might be introduced when enacting the request challenge on each HTTP request and prevent a malicious agent from downloading an executable that is disguised as an HTML file. However, it may lower the security by allowing the malicious request to complete even if the software agent is detected as malicious later. The experimental results demonstrate the effectiveness of PICs in identifying malware that attempts to imitate the network connection of popular browsers. It introduces an average of 353 milliseconds end-to-end latency overhead using request challenges and 24 milliseconds using response challenges.

In summary, we make the following contributions:

- We designed a malware detection system that utilizes a two-pronged approach to identify malicious traffic. We first classify traffic using passive inspection. For the flows that correspond to potentially legitimate applications, we "challenge" the host application by automatically crafting Program Interactive Challenges (PICs) that exercise complex functionality present in the legitimate application.
- We demonstrate the feasibility of our approach using HTML, Flash, Javascript, and other common browser components to form challenges for browsers. PIC was able to expose a wide-range of malware threats operating inside Enterprise networks.
- Netgator can be used in practice: it does not incur significant communication overhead and it does not require any user interaction or changes on the commodity application software.

2 Background

2.1 HTTP Headers & MIME Types

HTTP request and response packets contain various header elements which encase pertinent information about the transmission. Requests are prefaced by various headers notifying the server of what the client expects to receive. Responses

are accompanied by headers as well informing the client of what is being transmitted. Each browser uses a distinct header ordering. The same browsers also have slight differences depending on which operating system they are running. Our passive inspection module uses these unique orderings to create signatures, which can be used by the active challenge module to identify browers and pick appropriate PICs to challenge the browser. Multi-purpose Internet Mail Extension(MIME) types describe the content type of the message being transmitted. The main general MIME types are application, audio, image, text, and video. Since the majority of most web pages have the MIME type text/HTML that can be challenged on the response instead of the request, we can reduce the overall network delay caused by the active challenges.

Internet Content Adaptation Protocol (ICAP) allows for the modification and adaptation of HTTP requests and responses. All the various elements of HTTP messages can be edited. Typical uses of the ICAP protocol include actions such virus scanning or content filtering. The protocol relies on an ICAP client that forwards traffic to an ICAP server, which is in charge of the adaptation of requests/responses. In our implementation, we use the open source proxy Squid [5] as our ICAP client and use the open source option Greasyspoon [2] as our ICAP server. Greasyspoon allows scripts written in various languages (Java in our system) to act on all incoming requests and responses.

3 Threat Model & System Architecture

3.1 Threat Model and Assumptions

We assume that a client machine in an enterprise network may be infected with malicious code, and malicious code needs to "call home" and establish a back channel communication with remote server(s). The malware does not form connections immediately upon execution, but waits for an indeterminate amount of time or a user event before initiating network connection. Moreover, we assert that a certain subset of browser components and capabilities are necessary to navigate the Internet and confine our challenges to these.

The sophistication of most current malware has not yet reached the level of implementing or imitating the entire HTML, Javascript, or Flash engines and software stacks within themselves. The code size of a sophisticated malware will increase dramatically in order to include the functionalities for responding all the known active challenges, and make it prone to being detected. We assume malware that has infected a host does not wish to access the full software stack of the legitimate application software (e.g. browsers) that natively reside on the system in order to remain stealthy and launch attacks quickly.

NetGator utilizes a network-level transparent proxy to identify and filter out unknown traffic. For the traffic that matches programs that have been approved for the organization, we automatically craft active challenges to probe and verify end-point applications. Figure 2 depicts NetGator's system architecture. In the network, all traffic to be inspected is routed to this proxy. If the application

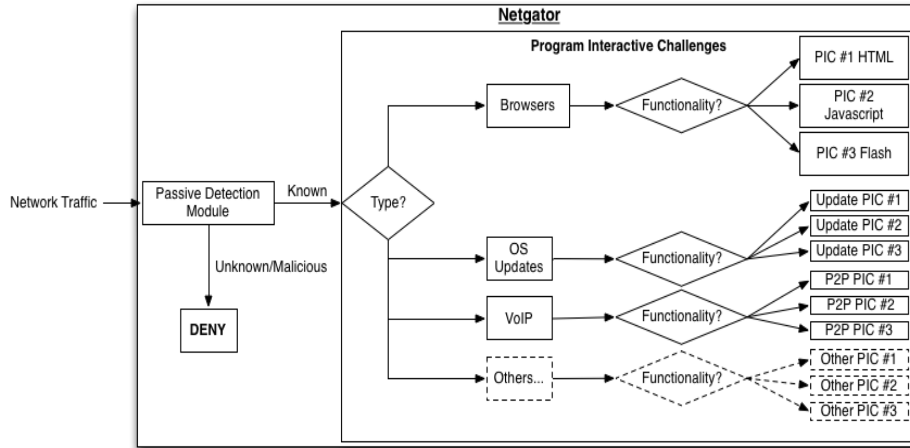


Fig. 2. NetGator System Architecture and Traffic Control Flow

passes both the passive inspection and active challenge, the proxy permits the outbound connection by forwarding the traffic to the default gateway. Connections that are either unknown or fail to pass the active test are dropped (or a human operator can be informed depending on the site's policy).

The network proxy consists of two major components: passive, signature-based flow inspection and active challenges. The passive inspection module acts as an initial filter, recognizing the known (and legitimate) category of the end-point software applications. We do not need the exact version of the end-point application but rathe its broader type (Browser, updater, etc.). Traffic from unknown applications can be treated preferencially allowing in the insertion of policies blocking, alerting, or even logging the flows that stem from such unmapped applications. This enables our system to adapt to new applications and network environments since new applications can be immediately recognized and mapped thus becoming "known". Traffic flows for which we already have a signature are issued active challenge(s) to further verify the legitimacy of the end-point software that generated the network traffic. The Program Interactive Challenges (PICs) are automatically generated by the proxy in advance and can chose from a wide variety of potential functionality based on the complexity of the end-point software. For instance, for browsers we show more than three different types of PICs that can be used.

To bootstrap in an enterprise network, traffic should be gathered for a period of time to decide what applications are operating on the network. Once armed with this information, we can utilize the passive inspection module to accomplish two tasks: filtering out malicious/unknown traffic and classifying known applications. First, by knowing which applications are expected to be traversing the network, we can drop any unknown, potentially malicious traffic as well as known malicious traffic. Second, the ability to classify the known applications

allows us to send the corresponding active challenges to specific types of applications. We can derive passive signatures from packet header information collected from packet sniffers such as Wireshark [7]. These signatures are representative of the distinct HTTP header content and HTTP header ordering that each browser possesses. However, the passive inspection module cannot provide timely filtering or blocking in zero-day attack situations. Moreover, since network requests can be easily altered, a request may be generated by malware masquerading as legitimate software. NetGator provides an active challenge mechanism to effectively detect and mitigate these attacks.

For a known application, the NetGator proxy can obtain the type and version of the supporting software from the passive signatures collected by the passive inspection. The proxy maintains a table that records the corresponding program interactive challenges (PICs) supported by each application software. Therefore, the proxy can send one or more PICs to the application that initiates the communication. For legitimate applications, they should be able to correctly respond to the challenges with their embedded functionalities. For example, we can leverage HTML, Flash, Javascript, and other common browser components to form challenges for browsers. If the application is unable to correctly solve and respond to the challenge, NetGator will flag the source as malicious.

Depending upon the type of requested data in the packet, the proxy can challenge either the request or the response. When challenging the network request the proxy receives a request from an application and blocks it, while returning a challenge as the response to the application's request. Only if the application can successfully solve the challenge and respond to the proxy, the proxy will forward the original request to the default network gateway. For instance, any HTTP request for application audio, or video data should be challenged and blocked until the challenge is solved. Challenging the network response allows the application's request to pass through but inserting our challenge into the original response from external servers and then sending it to the application. Legitimate applications can solve the challenges and notify the proxy. If the proxy cannot receive a correct answer from an application after sending the response challenge for a pre-defined time, it marks the software agent as malicious. For text/HTML requests, we can insert challenges into the response's text/HTML data. Challenging the request provides stronger security than challenging the response, since the application cannot obtain any useful data information from the external servers if it cannot pass the challenges. However, challenging the response can offer a smaller network latency, since the application only needs to solve the challenges after the response data has been received.

One of our primary design principle is to keep the system transparent to the end-user and adaptive to new software. Our approach does not require the user to prove or input anything, but shifts the onus of proof to the requesting application. Moreover, since our solution only utilizes the existing functionalities in commodity application software, we do not need to change their source code.

4 Design & Implementation

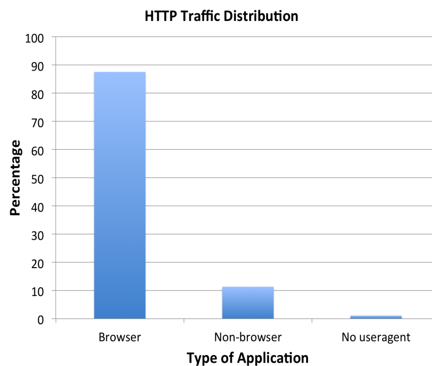


Fig. 3. Distribution of applications on port 80 of a large university network

We design and implement both the passive inspection module and the active challenge module in NetGator, which supports both challenging at the request and challenging at the response. Since browsers represent a vast majority of traffic on a typical network and HTTP protocol is widely exploited by malicious code, we focus on developing PICs for validating browsers. However, our methods can easily be adopted to challenge other agents by identifying unique functionality that they possess including ones that also utilize HTTP/S.

4.1 Passive Inspection

The passive inspection module consists of two parts: signature generation and signature matching. First, it employs protocol analysis to first identify the advertised client application type based on its network communication signature. This signature is derived from a distinct ordering of traffic headers found in each client. Since different versions may support different sets of functions, it generates the signatures for different software agents with different version numbers and saves the signature in a data set. Second, it inspects the real time traffic, dynamically derives the signature of the user agent, and compares it with the signature seen to identify the user agent. The signature set is used to determine the claimed identity of the end-point software that generated the packets as known or unknown. Our passive module will drop the traffic from unknown client programs. Known traffic (e.g., http(s) through port 80 or 443), may pass through without being blocked or receive further inspection.

The signatures of user agents are generated automatically by observing traffic on the network and extracting HTTP header orderings. First a packet capture is performed on the network to gather information about which applications are running on the network. The pcap file is then exported into an XML file from

Wireshark [7]. Once the data is in XML format, it is processed by a Python script. This script then extracts the message header and assigns a number to each HTTP header forming the signature. With unique signatures for each application that exists on the network we are able to issue the proper PIC(s) for each agent.

To perform real time detection, it sniffs the packets that traverse the network scanning for HTTP header instances that match our signature set of various user agents. A modified version of tcpflow [6], which we call *protoflow*, scans the network traffic and pass the information to our identifier program. The change made is to write the data acquired by the capture to a space in memory which is shared by our second piece of software called *inspector*. Inspector takes the strings from the shared memory and compares the data against a collection of regular expressions that distinguish various user agents. These regular expressions are made up of the specific HTTP header ordering that are unique to each different browser.

We develop a string matching algorithm to check whether the "user-agent" string in the HTTP headers contains the name of browsers, including "Firefox", "Chrome", "Safari", "Opera", "MSIE 8.0", "MSIE 7.0", "MSIE 6.0". If yes, we label the packet as "browser", otherwise label it as "non-browser". For packets that do not provide any user-agent information, we label them as "non-labeled". We run the algorithm on the traffic of a large university for two hours, and label 8,825,177 packets as "browser", 1,143,040 packets as "non-browser", and 110,763 packets as "non-labeled". Figure 3 shows that traffic on port 80 (HTTP) is mostly comprised of browsers, but other applications can communicate via HTTP as well. The most common non-browser user agents include web-crawlers, application updates, bots, or spiders. For example, we observe 15,506 occurrences of web-crawlers. For those "non-browser" applications that utilize HTTP/S, we need to develop separate PICs for each type of user agents.

4.2 Active Challenges

The active challenge module consists of a transparent proxy and an ICAP server. The proxy redirects all network request and response traffic on ports 80 and 443 to the ICAP server which then generates and verifies PICs to the client software. Based on the Multipurpose Internet Mail Extensions (MIME) type of the requested data, the ICAP server either rewrites the response completely, or simply inserts additional code into the response code. If the response is any type of non-text/html data, the client 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 original response to reduce the network delay.

We use Squid 3.1.8 as the proxy for rerouting HTTP and HTTPS traffic and the ICAP server, Greasyspoon, for handling scripts on requests and responses. To handle HTTPS traffic, Squid's ssl-bump feature is harnessed. HTTPS traffic leaving the host is encrypted with a single key that the proxy possesses. Once the traffic reaches the proxy, it is decrypted and forwarded to Greasyspoon for generating and verifying challenges. Next, the proxy re-encrypts the request with

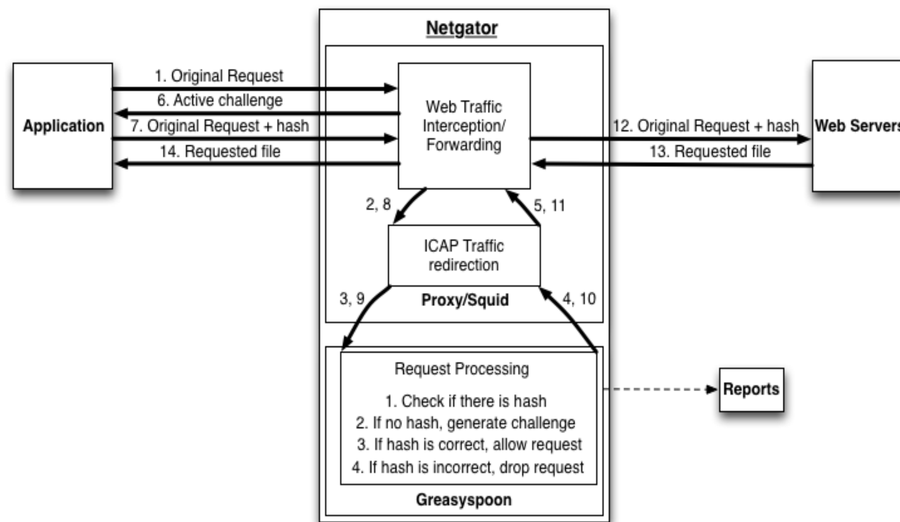


Fig. 4. Active Request Challenge Flow

the key established with the targeted external web server. This raises the concern that the client will never receive the external server's certificate, thus leaving it vulnerable to phishing sites. This can be handled by leveraging the trust of the Netgator proxy. That server can determine if a certificate that returns from an external site is legitimate or not. This method enables us to intercept HTTPS requests initiated by malware where other solutions would typically fail.

To keep track of the various connections, we leverage Greasyspoon's cache which contains a hashmap. For each IP address and user agent pair, this hashmap contain an entry that records whether the client passed a challenge, how many times it has been challenged, as well as how many times it passes the challenge. Thus, even if the malware correctly forms a user-agent string that is operational on the infected machine, the hashmap still can reflect that an entity on the system did not pass the challenge. The hashmap is periodically written to a log file available for inspection.

To reduce the number of challenges, the proxy automatically passes a network request for a page if the requesting client has passed a request for that page's domain. For example, if a client requests 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 the burden from websites that trigger many GET requests for items such as images or flash objects. The list of "passed" domains is periodically cleared in order to catch malware that might use one of these sites for control communication. It is also possible to operate Netgator without keeping these records, and thereby issues challenges to each request, albeit with an increase in overhead.

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

(a) Javascript Challenge Code

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

Fig. 5. Response with various active challenges.

Challenging Network Requests When the proxy observes a request for non-text/html data, it issues a challenge to the client. The challenge can take various forms based on the functionality of the browser. 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, which is a Flash object, not actual lines of code. The nature of the Flash challenge actually allows it to be more difficult for an attacker to bypass since the Flash object is embedded in the HTML page with Javascript, essentially combining two of our challenges into one. To correctly pass the requested URL and hash to the Flash object, Javascript functions are embedded and 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 can match the new request with an appended hash to the originally requested URL. If the hash is correct, the request is allowed to pass; if the hash is incorrect, the request is dropped and an error message is sent to the client. Figure 4 shows the implementation for the request challenges.

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, Greasyspoon executes a Javascript, which generates the hash and then crafts the new HTML code that will be returned to the client. A tuple of four factors is used to generate

the hash: 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 Javascript calculates a hash value from this tuple using SHA1. Next, this Javascript replaces the header and the body of the request in order to customize the response with the hash value for the client. The header must be replaced with a properly formed HTTP response header to signal Greasyspoon 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 HTML code and insert a fragment of Javascript code that executes a redirect operation. The sample response codes using Javascript, HTMl, and Flash are shown in Figure 5.

The codes for the Flash and HTML challenges both contain 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 a particular IP and software agent combination has passed the challenge. The size of the request scripts each average around 280 lines of code.

Since the hash is sent back in plain text it is conceivable that an attacker could simply parse the response for the hash and initiate the correctly formed request if they have knowledge of our system. We can prevent this attack by encrypting the hash with an AES Javascript implementation [3]. The new Javascript provides the code to decrypt the hash and requires the malware to include functions for AES decryption.

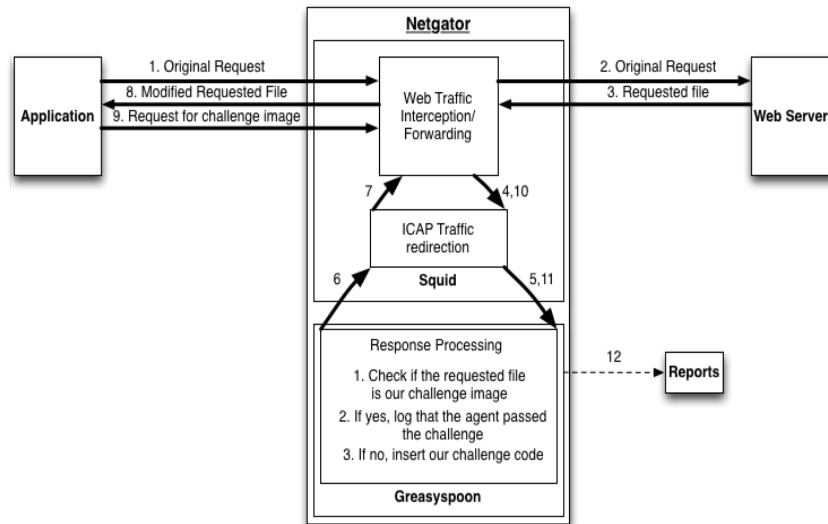


Fig. 6. Active Response Challenge Flow

```
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
```

Fig. 7. Example Logfile Entry.

Challenging Network Responses If the data requested is of the type text/html, the proxy allows the request pass through. When the response comes back for that request, a challenge code is inserted in the response. For instance, an image that resides on the proxy can be embedded in a Javascript challenge code. A Javascript *write* statement tells the browser to include the image via HTML "img" tags. The proxy then looks for requested for this specific image and once it sees one, it knows that the challenge has been completed successfully. Figure 6 shows the control flow for active response challenges.

The processing script has two parts: a request script and a response script. Combined, there is about 300 lines of code. The request processing script first determines if the client is expecting a text/html response or if the request is for our specific challenge image. If the client is expecting a text/html response, an entry of the user-agent string combined with the client IP is written into the hashmap. The original request then goes out to the intended server. If the request is for the challenge image, Greasyspoon searches for a corresponding entry in the hashmap and updates it reflecting that the client has passed a challenge. Once a response for the connection is received, the response processing 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 injects the HTML code with the embedded challenge image inside the original response and sends the response back to the client. The entry is revised to show that a challenge has been sent 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 7. The operating system, application name, and application version are all extracted from the user-agent string. It can help administrators analyze the logs and diagnose a problem should one arise.

The reason for adapting response challenges is two-fold. First, it allows us to reduce the overhead that might be introduced when enacting the request challenge on each HTTP request. Moreover, blocking every data request would impair our system's scalability and usability. It is conceivable that on a smaller, more confined network the system could be set up to challenge every request, but on a larger infrastructure this would most likely be impractical. Second, it can prevent a malicious agent from downloading an executable that is disguised as an HTML file.

5 Experimental Evaluation

We implemented a prototype NetGator system consisting of a laptop for the client and a Dell server for the proxy. The laptop is a Dell Latitude E6410 with an Intel Core i7 M620 CPU at 2.67 GHz, 8GB of RAM and a Gigabit network interface. The server is a Dell PowerEdge 1950 with two Xeon processors, 16GB of RAM and a Gigabit network interface.

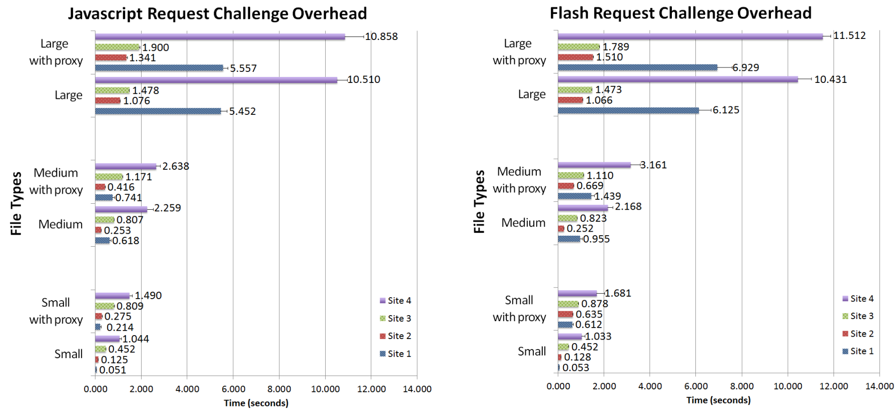
For performance testing, Firefox 3.6.17 is used as the client's browser throughout. To measure how efficiently the server could process the scripts that will be executed on the client's request, a script loops through 10,000 iterations of the request script with the iterations per second being returned. For all the figures in this section, we run this script 30 times and use the average value to determine the capability of our server. The error bars show confidence interval at 95% confidence. For all testing, Squid and Firefox's caching mechanisms are completely disabled.

5.1 Performance Analysis

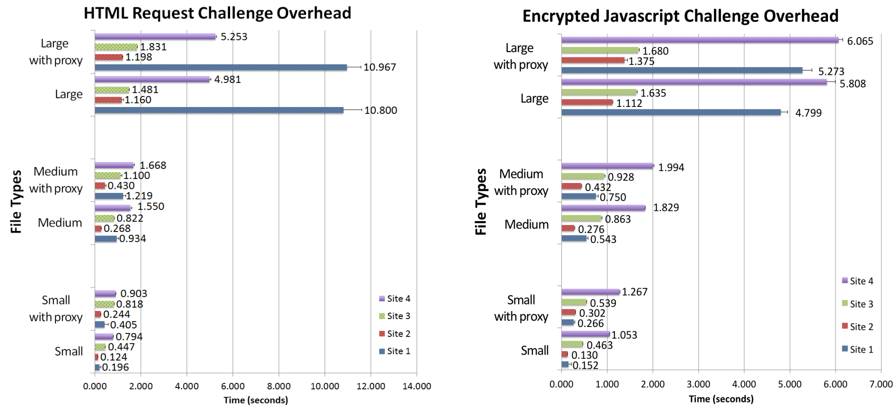
To evaluate the end-to-end latency of the request challenge, we analyze various types of challenges we create in HTTP download scenarios utilizing PlanetLab [4] combined with our passive inspection. Different PlanetLab nodes are used from throughout the world using all virtualized hardware. Four nodes (one from the East Coast, one from the West Coast, and two nodes from other continents) are utilized to perform the benchmarking. Executable files of large size (1000KB), medium size (100KB) and small size (10KB) are hosted on each node on an Apache web server. The client then downloads each file thirty times from each of the nodes, both with and without the NetGator proxy. The values of end-to-end latency are determined by the difference in the time-stamp of the packet that starts the initial request before the challenge and the last packet that closes the connection after downloading the file. Figure 8 shows the end-to-end latencies with and without using Javascript, Flash, and HTML request challenges for requesting different sizes of files.

Our experiments show that the end-to-end latency using request challenges is almost negligible to the client. When using Javascript as the challenges, it increases the end-to-end latency by 274 milliseconds in average for all size of file types from four sites. The Flash request challenge has an average latency overhead of 580 milliseconds, while the HTML request challenge introduces 206 milliseconds of latency overhead. Figure 8(d) shows that the encrypted Javascript request challenge has only 174 milliseconds of latency overhead, which is less than the normal Javascript challenge and HTML challenge. We see that enhancing security by encrypting the hash may not increase the end-to-end latency.

To measure the overhead of the response challenges, we perform experiments using a wide-range of websites throughout the country. Baselines are established for each website by performing a simple loading of each of them without the proxy involved. Once these baselines are established, the gateway of the client laptop was changed to be our proxy. Each website is loaded 30 times both with



(a) Javascript Challenge End-to-End Latency. (b) Flash Challenge End-to-End Latency.



(c) HTML Challenge End-to-End Latency. (d) Encrypted Javascript Challenge End-to-End Latency.

Fig. 8. Request Challenge Overhead.

and without the response challenge. In order to establish time, the difference between the time-stamp of the first and last packet in the stream is taken. The results of these experiments are shown in Figure 9. We can see that the response challenge only introduces very small latency overhead.

The request challenge results demonstrate that the overhead remains constant 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 24 milliseconds. With the minimal amount of overhead that our system introduces, it is not perceivable to the user. All of the challenging happens without any interaction from the user, allowing a seamless experience while maintaining the security of

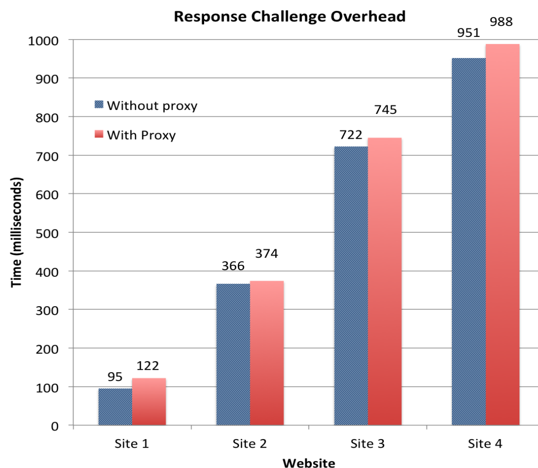


Fig. 9. Response Challenge End-to-End Latency.

a network. Moreover, our proxy is exceedingly efficient in processing the scripts, being able to handle on average approximately 1,200 request scripts per second.

For the malware testing we obtained a set of malware through a custom crafted retrieval mechanism. This mechanism is provided malicious URLs from Malware Domain List and Google. The samples used during testing were solely Windows executables. During our testing to establish what percentage of malware calls out utilizing HTTP/S, we find that none of the malware which used either protocol could overcome our challenge architecture. That sample size equates to 817 malware samples challenged. The typical behavior of the infected systems simply try to re-request the file it has originally sought after only to repeatedly be returned our challenge. We do not observe any false positives in our experiments. The level of false positives will be directly related to how many browsers in a network utilize HTTP/S but do not contain HTML, Javascript, or Flash engines.

6 Discussion and Limitations

We assume that malicious agents do not typically access the full software stack of applications that reside on the infected host. We also assume that the malware does not include its own Javascript engine. If malware is forced to implement a full browser agent complete with Javascript/Flash functionality, this would greatly increase the presence of the malware thus increasing its vulnerable to detection. Based on this assumption, the malware will not be able to decode the encrypted hash of the challenge even when the key is passed to the host. Our testing of recent malware samples shows us that the current level of sophistication of malware does not include their own Javascript engines nor the ability to access the full software stack of web browsers present on the system. By issuing

each piece of malware our Javascript challenge, we are able to determine that none of the malware tested encompasses the Javascript functionality to overcome our challenge infrastructure.

Our system can challenge either the initial request or the response when considering the trade-offs between security and performance. The request challenge allows NetGator to sever the malware's connection immediately to negate any damage before it happens. This also comes with a cost of slight latency. On the other hand, the response challenge allows the response to return to the requesting agent before it issues the challenge. This dramatically lowers the latency the user experiences but also allows the original request to complete even if the software agent is detected as malicious later. This method relies on the monitoring of the logs to identify compromised hosts. We could improve its security by utilizing the information about agents that fail the challenge in the response PIC and create a signature to block future outbound connection attempts. Intercepting SSL traffic causes a possible issue in the transparency to users. Our approach to processing HTTPS traffic essentially acts in the same way a man in the middle attack works. Browsers typically identify this behavior and report the suspicious activity to the user. This can be mitigated by the hosts having their organizations certificates installed on the end hosts. Without these certificates the transparency to the users is affected.

If an attacker is aware of the presence of our system, they would likely attempt to craft their communications in a way to evade our detection. An approach that they might take would be to label their communication as a simple non-browser agent. If the agent is not one of the approved applications to transmit across ports 80 or 443, then the connection would be severed. If it is an approved application, it would be challenged in the same way that browsers are. In an enterprise network, there would ideally be a full repertoire of challenges to issue to the various applications that communicate across ports 80 and 443. However, there may be a necessary application for which a challenge can not be crafted. In order for us to create a PIC for a particular application, we require that is has specific functionality that will have an expected response to some request. For agents that do not meet the requirements to create a PIC, a whitelist of servers for these application to communicate with can be constructed and any connections from these applications to other servers would be raised as suspicious. Malware might also utilize a full application (a legitimate web browser) in their communications to correctly pass our challenges. While possible, this increases the likelyhood of being detected due to not being able to rely on their own covert communications. Encompassing a browser's full capabilities would evade our detection and is a limitation of our approach.

7 Related Work

Our work is partly inspired by various automatic protocol analysis systems [20, 29], which utilize injection of messages to various applications in order to automatically determine how a particular protocol is organized. Instead of injecting

messages to test a protocol, we examine a particular application software to prove its identity.

Recent research by Gu et al. [16] is the most simliar to our approach. They aim to detect botnet communication over IRC through a combination of user interaction and probing for expected responses. There are two main differences with our approach. First, we do not expect a human to be behind the communications, nor rely on one at any time to be able to solve our challenges. Second, their paper focuses on detection of malicious botnets, while ours is concerned with verifying the identity of legitimate end-host applications. Our approach is beneficial in the sense that the signatures (expected responses) of botnets will continue to grow consistently while Netgator only has to establish signatures (challenges) for legitimate applications which will not likely change their functionality over time.

Our work can be compared to techniques used by OS and application fingerprinting programs such as Nmap [21]. The most popular form of real-time browser challenges is to utilize server-side techniques that read browser configuration files [23, 26] (Javascript, ASP, etc.), cookie information [22], or search for platform specific components like Flash blockers or Silverlight [12]. Another approach is to search traffic flows for known, specific identifiers like connections to Firefox update servers [30]. Conversely, techniques like the well-known CAPTCHA puzzles attempt to prove the existence of a human user. However, all those methods are disruptive and not transparent to the user.

Traditionally, botnet detection and mitigation systems like BotSniffer [17] have focused on zombies that contact Internet Relay Chat(IRC) C&C servers or utilize IRC-style communication [9]. Unfortunately, botnets have grown in sophistication to use Peer-to-Peer (P2P) and unstructured communication [10, 19]. In addition to the traditional techniques such as blacklisting, both signature and anomaly-based detection, and DNS traffic analysis, BotHunter [15] proposes using infection models to find bots, while BotMiner [14] analyzes aggregated network traffic. Our work is a another complementary study utilizing an active challenge technique to distinguish certain types of bots from benign applications.

Data transmission over HTTP/S is very common and consumes the bulk of un-filtered traffic in most organizations. For analyzing packets that contain a payload, deep packet inspection techniques are favored. Signature or anomaly based detection is applied to these packets [10, 13, 25, 27, 8, 28]. To foil this mechanism, malware may use the same secure protocols that users employ to protect themselves from malicious agents [24, 18]. Our approach only analyzes the data content to determine the protocol being transmitted. Once the protocol type is established, we are are concerned solely with the client behind the communications.

8 Conclusions

We study an active and in-line malware detection system, called NetGator. Our goal is to be able to detect outbound malware flows for when malware attempts

to establish a network connection to a back-end server. Our approach is two-pronged: the passive classification module analyzes network flows to determine the claimed identity of the end-point software that generated the packets. It relies on existing network program signatures to classify end-point applications and drop the unknown requests. Next, NetGator verifies the legitimacy of the end-point software by generating easy to generate and computed in-line Program Interactive Challenges (PICs) based on the functional capabilities supported by the end-point software.

Although our approach can be potentially circumvented by sophisticated targeted malware, we believe that NetGator significantly increases the complexity of the attack forcing the adversary to perform additional invasive tasks before it can successfully communicate data. On the other hand, NetGator is fully transparent to the user: our experiments demonstrate that it can examine real-time traffic. In addition, the detection results demonstrate the effectiveness of PICs in identifying malware that attempts to imitate the network connection of popular browsers. NetGator introduces an average of 353 milliseconds end-to-end latency overhead using network request challenges and 24 milliseconds using network response challenges.

References

1. Anti-Phishing Working Group, ADWG 2011 Trends Report. `http://apwg.org/reports/apwg_trends_report_h1_2011.pdf`.
2. Greasyspoon. `http://greasyspoon.sourceforge.net/`.
3. Javascript encryption. `http://javascript.about.com/library/blencrypt.htm`.
4. Planetlab. `http://planet-lab.org/`.
5. Squid. `http://www.squid-cache.org/`.
6. Tcpflow. `http://afflib.org/software/tcpflow`.
7. Wireshark. `http://www.wireshark.org/`.
8. G. Androulidakis, V. Chatzigiannakis, and S. Papavassiliou. Network anomaly detection and classification via opportunistic sampling. *Network, IEEE*, 23(1):6 –12, january-february 2009.
9. 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.
10. 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.
11. Cyveillance. Malware detection rates for leading av solutions. `http://www.cyveillance.com/web/docs/WP_MalwareDetectionRates.pdf`, 2010.
12. P. Eckersley. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.
13. P. Garcia-Teodoro, J. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1-2):18 – 28, 2009.

14. 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.
15. 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.
16. G. Gu, V. Yegneswaran, P. Porras, J. Stoll, and W. Lee. Active botnet probing to identify obscure command and control channels. In *In Computer Security Applications Conference (ACSAC)*, 2009.
17. 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.
18. 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.
19. 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.
20. 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.
21. G. Lyon. Nmap security scanner, 2010.
22. K. McKinley. Cleaning Up After Cookies, 2008.
23. M. D. Network. How to: Detect browser types and browser capabilities in asp.net web pages, 2010.
24. 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.
25. M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2:257–274, 2007. 10.1007/s11416-006-0031-z.
26. W. Schools. Javascript browser detection. `http://www.w3schools.com/js/js_browser.asp`, 2010.
27. T. Shon and J. Moon. A hybrid machine learning approach to network anomaly detection. *Information Sciences*, 177(18):3799 – 3821, 2007.
28. S. Thorat, A. Khandelwal, B. Bruhadeshwar, and K. Kishore. Payload content based network anomaly detection. In *Applications of Digital Information and Web Technologies, 2008. ICADIWT 2008. First International Conference on the*, pages 127 –132, aug. 2008.
29. 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.
30. T.-F. Yen, X. Huang, F. Monrose, and M. Reiter. Browser fingerprinting from coarse traffic summaries: Techniques and implications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5587 of *Lecture Notes in Computer Science*, pages 157–175. 2009.