Host based anomaly detection for webservers

RP1.4: Project Report (Host based IDS)

Sudesh Jethoe
12-11-2012

Summary
Byte Internet is a dutch hosting company, which runs a shared hosting platform. Securing this platform is an ongoing challenge. Partly, this is because the hosting company is dependant on the website owners to keep their sites maintained and secure.
In this research a solution was created and tested to detect a few types of malware often found on the servers of Byte Internet, namely webshells and uploaders.
A detection method was devised and implemented. Although it was able to detect samples, we were unable to find any malware during the live tests.

0. Index

1. Introduction

generic

Web applications have grown to play a big part in the internet of today. Web applications enable us to communicate through online fora, share our photo's and videos, do online shopping and even write reports and other documents. The majority of these web applications is written in php and runs on the apache webserver[1][2]. A site which uses a platform running apache and php is therefore an attractive target for attackers.

problem

Attacks on webservers these days often only have a few purposes. Mostly a hacked site is misused to send spam or a so called "webshell" is installed, which can be misused in a variety of ways.

Spamruns affect the reputation of the site owner if the domain is used for sending spam messages.

IP ranges can get blacklisted, which can block transmission of regular e-mail of only the site owner, but also the hosting provider and other customers of the hosting provider.

Webshells are scripts which provide a webinterface to upload and download files from the server and execute commands accessible on the server. Webshells not only enable potential hackers access to customer data, they also give attackers the possibility to gather information about the server and exploit the systems even further. For example by uploading malicious scripts or downloading any accessible files. Scripts which in turn are used to send spam e-mail or participate in ddos attacks.

cause

Webservers can get hacked in a variety of ways. There might be problems with the firewall configuration, bugs in the webserver software (Apache, IIS) or even bugs in the used coding language. However, the most common causes of hacks are:

- Outdated versions of web application frameworks (Joomla, Wordpress)
- Insecure plugins for these frameworks.
- Captured passwords for plaintext logins (ftp).

The reason for this is that exploits in software which is part of the hosting platform is actively maintained by the hosting company. Website builders build a site for their customers and put it online. However, not all live sites are updated and patched regularly, which makes them vulnerable for attacks.

possible solutions

Several types of detection systems have been developed to counter attacks against web services.

Network Intrusion Detection System (Network IDS)

There are detection systems which try to detect and block malicious traffic by inspecting traffic on the network. These are called network intrusion detection systems (NIDS), an example is snort.

Web Application Firewalls (WAF)

Web application firewalls run directly on the server or as an appliance and intercept traffic at higher layers of the networking stack. An example is mod_security, a module for the Apache webserver.

In heterogeneous server environments, a variety of attack vectors is available for attackers. All software contains bugs and the more different software is run, the harder it eventually will be to keep up with all security patches. Although network scanners and web application firewalls are able to detect uploads of malicious code, not all can be detected and some will eventually get through.

Host Intrusion Detection System (Host IDS)

It is equally important therefore to have a host based detection mechanism in place, which is able to classify suspected behaviour on servers and detect this kind of behaviour when it occurs.
Host based intrusion detection systems are systems which run locally on the server and process logfiles, check files for changes and detect other anomalies on the host, like rootkits.

Byte Internet

Byte Internet is a dutch hosting company which hosts around ten thousand websites. Part of these sites run in a shared hosting environment. In a shared hosting environment, sites of different customers run on the same physical server. In a shared environment it is very important to keep customer data separated. Not only to prevent customers accessing each other's data, but also to prevent potential hackers from accessing other customers data or interfering with the reachability of their websites.

In a shared hosting environment part of the responsibility of securing the system lies with the maintainers of these websites. This can make it difficult for a hosting company to secure such a system. Sites using outdated versions of frameworks or plugins for these frameworks can pose a threat to the environment.

As we already mentioned a lot can be done to prevent attackers from overtaking the systems. Detection can be done on the network level and on the servers themselves. However, some attacks will always slip through. When they do they are often only detected after hackers have achieved their goals and misused the systems for whatever purpose.

We think it should be possible to detect and prevent malware by looking at its behaviour when it is being executed. In this research we will try to develop a method to detect the execution

malware. In this research we will be looking into the detection of malware which enables attackers to access, send and retrieve data to and from the webserver. Examples are webshells and uploaders. This type of malware is selected, since it is encountered frequently within Byte Internet. We will refer to this type of malware as “interactive malware” or malware.

research questions

The following questions shall be considered when conducting the research:

- Can we develop a method which detects interactive malware running on servers?
    - What are the characteristics of this kind of malware?
    - How can the characteristics be used to detect this malware?
    - Is there a framework which is suitable for use in a shared webhosting environment?
    - Can we integrate the detection method in a suitable framework?

structure of this research

To answer these research questions, first we will give a general overview of research done and methods used in malware analysis. Then we are going to discuss the hosting setup used at Byte Internet. Once we have established a ground in malware analysis, we can decide on an approach for our research. We will motivate our approach and then define the characteristics of the malware which we want to detect. Based on these characteristics a solution will be selected in which we will integrate a detection method. After, the prerequisites for successful deployment will be evaluated. Once we know if our solution has a chance to succeed, an implementation will be created.

Finally we will test this implementation, review our achieved results, conclude our work and give our recommendations for the future.

2. Background (literature/related work)

Malware analysis

Malware can be analysed in three ways, namely static, dynamic and post-mortem [5].

In static analysis, the malicious program is reviewed without actually executing it. This is done by looking at the program with a decompiler, disassembler or just by looking at the plain source with a text editor. Static analysis has the advantage that it can show how a program will behave in unusual conditions. However it can be time consuming and complex with larger programs.

In dynamic analysis, the malicious program is studied while it is being executed. This is done by inspecting the execution of a program using debuggers, function call tracers and network sniffers. Dynamic analysis has the advantage of being fast and accurate. However dynamic analysis might not execute all functions contained within the program. Requiring the program to be executed, it could also harm live systems.

In post-mortem analysis, not the program itself, but the effects of a program after its execution are studied. This is done by looking at events stored in logfiles, files which have been modified or left over data in memory. Advantages of post-mortem analysis is that it is always available. Programs always interact with the environment they are running in, therefore they will always leave traces. These traces however, can become harder to detect as time passes. Logfiles are rotated and memory is overwritten with new data.

Previous studies on malware analysis

In [3] an approach has been researched, which tries to detect malicious programs by mapping high level (malicious) behaviours to low level triggers. Examples of these low level triggers are:

- the opening of send / receive sockets (network or file)
- ip addresses on which data is sent / received
- data
- combinations of keystrokes

The triggers were then measured for different kinds of software and malware and a comparison was made to identify differences between benign and malicious programs.

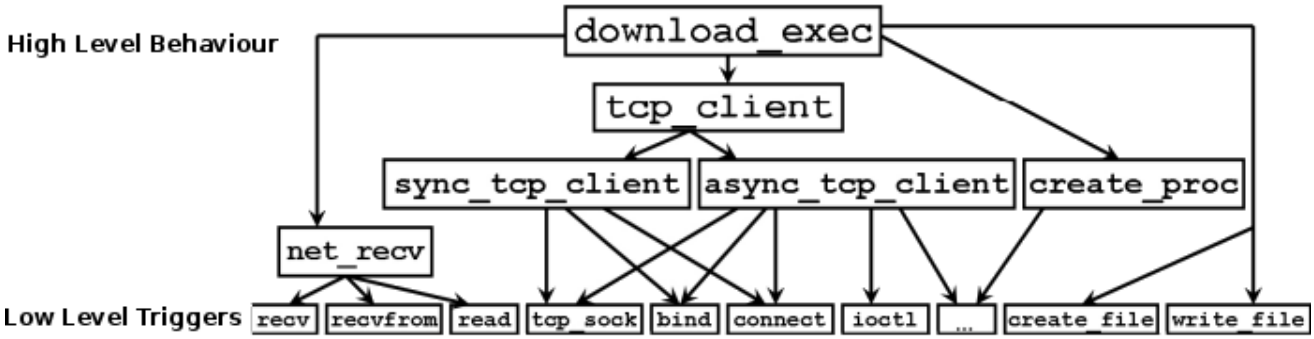

Figure 1: Overview of a mapping of low level triggers, which combined form the action "download executable" [3]

Some of the high level behaviours which were measured, were:

- File creation and execution
- Downloading files from the network
- Sending email
- Sending of UDP packets to uncommon ports and addresses

The method showed clear distinctions between benign and malicious programs for certain high level behaviours. However, the reported slowdown over native execution was in the order of 18 to 34 times the normal execution speeds.

In [4] a machine learning based approach was tested. The behaviour of known malware was recorded in a sandbox environment. The recorded patterns were then clustered and classified. To test for malware, all software was executed in the sandbox environment.

Interactive Malware

In this research we focus on "interactive malware". By interactive malware, we mean malware /malicious- code/programs, which enables an attacker to interact with the program and send instructions. Instructions to upload, download or modify files and scripts, or even send commands to be executed on the server.

Why do we focus on this kind of malware?
When attackers acquire unauthorized access to a server, the first thing they often do is upload a script, which enables them easy access to the server (see figure 2). Although the exploit may be fixed and a password may be changed, a script on a server can remain unnoticed for a long time. Sometimes the malicious code can even be incorporated into regular files, which makes it even harder to detect.
Once attackers have put their access scripts on the server, it is a lot easier for them to start using the server for their own purposes.
Attackers for example use interactive scripts to:
Upload their own programs, like spam- or ddos-scripts.

Gather information on the system, like which system files are accessible for the attacker or grab (hashed)passwords from configuration files
Download valuable customer data.
Redirect pages to spamsites.

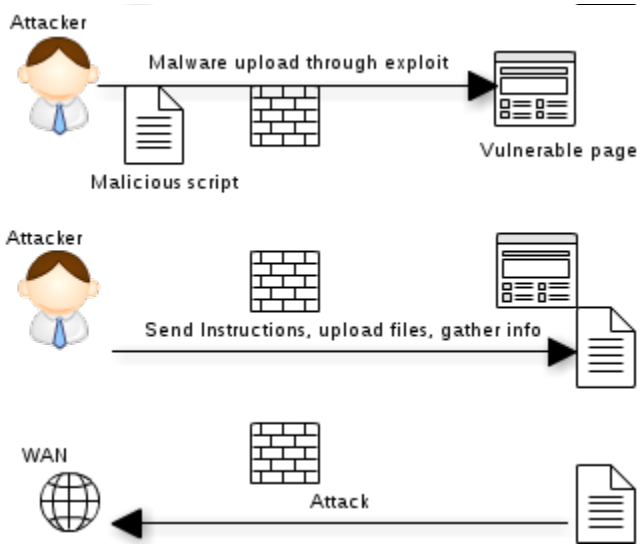

Figure 2: Attack schema

Hosting at Byte Internet

generic

Byte Internet offers shared and dedicated hosting for hosting generic PHP applications. Besides, they specialize in hosting of the PHP E-commerce platform Magento.
In this research we focus on the shared hosting platform for generic PHP applications. This, because the shared hosting platform is attacked successfully more often than the dedicated hosting and magento hosting platform. The reason for this is that web applications which run in a specialized environment are often maintained by professional webdevelopers and -designers. These applications are therefore more often updated and are less likely to run software containing exploits. Besides it could be argued that E-commerce software has a stronger focus on security by default, since it is involved in doing online transactions.

shared hosting overview

Byte Internet uses a multi tiered architecture for their shared hosting environment. This works roughly as follows (figure 2):

1. A website visitor comes in from the WAN.
2. The visitor passes the loadbalancer and gets redirect to one of many webservers.
3. On this webserver the Apache service runs and handles the HTTP-request.
4. Possible files requested are retrieved from the local cache or from the NFS-mounted storage server if it is not in the cache.
5. Possible files sent are stored on the NFS-mounted storage server.

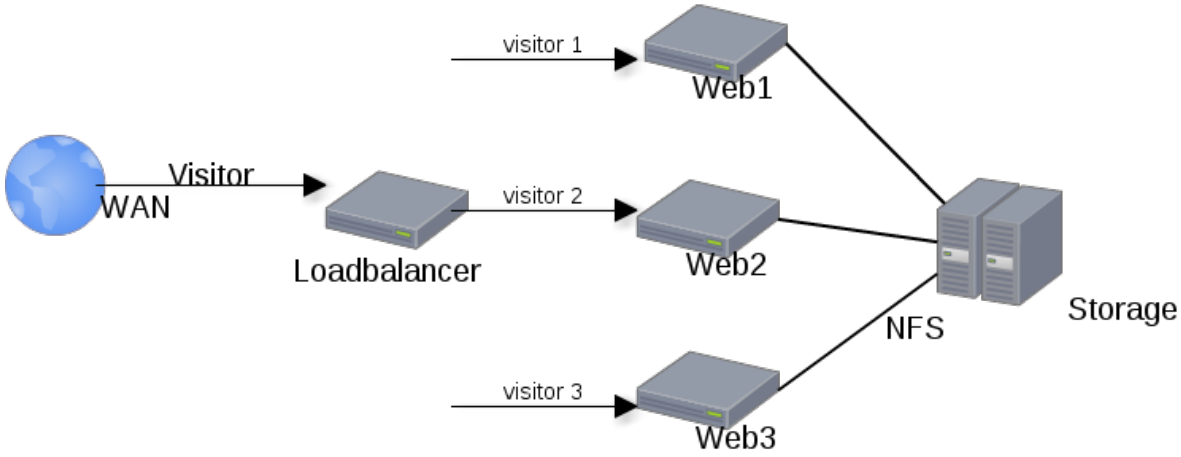

Figure 2: Rough schematic of the Byte Internet shared hosting platform

3. Method

Approach

To answer the research questions, first we will determine the characteristics of interactive malware.
Then we try to find a framework which can detect these kind of characteristics and is suitable for running in a webhosting environment.
After, we will collect some samples of malicious code found at the hosting company (Byte Internet).
Finally we will create a prototype and test it against our samples.

Motivation

The previously stated approach is chosen for the following reasons:

To be able to detect the malware we need to know how it behaves, therefore the first step is to determine its behaviour.

Once we have decided what behaviour we want to detect, we can find a framework which can detect it. Besides, we need to make sure the selected framework is suitable for running in a shared hosting environment.

After having selected our detection method and -framework we need to create a prototype to see if this behaviour can actually be used to detect these programs, to what extent and specificity.

Defining interactive malware

To be able to devise a method which can detect a variety of malicious code we need to determine how we can identify malicious code. We also need to determine how we can make a distinction between malicious code and regular code.

Identifying (malicious) code

Since these scripts run on a webserver there are only a few options to send information to them. All of these options require HTTP-request methods. Why? Because these are the only requests handled by the webserver. Relevant parts of the RFC on HTTP-request methods in use are shown below[6]:

```
OPTIONS
       The OPTIONS method represents a request for information about the
          communication options available on the request/response chain
          identified by the Request-URI. This method allows the client to
          determine the options and/or requirements associated with a resource,
          or the capabilities of a server, without implying a resource action
          or initiating a resource retrieval.

GET
```

```
      The GET method means retrieve whatever information (in the form of an
         entity) is identified by the Request-URI. If the Request-URI refers
         to a data-producing process, it is the produced data which shall be
         returned as the entity in the response and not the source text of the
         process, unless that text happens to be the output of the process.

HEAD
      The HEAD method is identical to GET except that the server MUST NOT
         return a message-body in the response. The metainformation contained
         in the HTTP headers in response to a HEAD request SHOULD be identical
         to the information sent in response to a GET request. This method can
         be used for obtaining metainformation about the entity implied by the
         request without transferring the entity-body itself. This method is
         often used for testing hypertext links for validity, accessibility,
         and recent modification.

POST
        The POST method is used to request that the origin server accept the
         entity enclosed in the request as a new subordinate of the resource
         identified by the Request-URI in the Request-Line. POST is designed
         to allow a uniform method to cover the following functions:

            - Annotation of existing resources;

            - Posting a message to a bulletin board, newsgroup, mailing list,
              or similar group of articles;

            - Providing a block of data, such as the result of submitting a
              form, to a data-handling process;

            - Extending a database through an append operation.
```

Of these request methods, only POST and GET are used frequently to send information to a server. However there is a difference between the two.
With GET requests, the information is embedded in the URI (uniform resource identifier). A full URI for HTTP-requests is specified by [8]:

*http(s)://domain:port/path?**query_string**#fragment_id*

When using a GET-method, the *query_string* part in this URI is passed as a parameter to the script parsing the HTTP-request. Often this *query_string* contains key value pairs, encoded as *"?key1=value1&key2=value2&key3=value3#"*. These pairs can often be called directly in serverside scripts. Although the RFC specifies that parameters passed in this way, with a GET-method, should only be used to return the information which is specified by the request. However, since information is sent to the server, a GET-method could also be used to send (malicious) code to a server. However, although the RFC[6] does not specify a maximum length, the amount of data which can be sent in this way is limited. Web servers and -browsers have differing maximum URL-lengths which they can process. For Apache the maximum length for a request line is limited to 8190 bytes by default, but can be set to other values [10].

With POST-methods the information to be sent is not included in the request line, but in the request body. According to the RFC, this is the method which should be used for sending data [6]. By default the Apache webserver does not impose a limit on the maximum length of a request body [11], however it can be set to a limit of up to two GiB.

Knowing the amount of information which can be sent using a GET-method is limited, we can assume that most malicious scripts will somehow make use of a POST-method. This gives us a way of identifying malicious scripts. However, POST-methods can also be used in normal scripts.

Discriminating between benign and malicious scripts

In normal websites the scripts to which data can be sent are known in advance by the website designer. Therefore if data is sent to another file than specified by the website designer, this is already an indication of a malware infection of the website. However malicious code not always resides in a separate file, it might also be integrated within regular files.

Detecting malware

To be able to detect malware, the detection framework should be able to:

- Detect POST requests
- Track which files are allowed to accept POST request
- Track the integrity of files which are allowed to accept POST requests, since attackers might hide malicious code inside these files.

Selecting a detection framework

The detection frameworks which can be used to detect malicious programs were already mentioned shortly in the introduction. Detection can be done by scanning the network, run as part of the web server service (Apache), or running as a separate agent on the webserver as a host IDS. For each of these we shall review the pros and cons and select a suitable solution. Besides looking at its possibility to detect malware, we will also need to determine whether the selected solution is suitable for running within a shared hosting environment.

Network Intrusion Detection System (Network IDS)

A network IDS tries to detect malicious activity by analyzing network traffic. A typical setup is show in figure 3. All traffic coming the Internet passes the IDS first before being sent to the local network. Network IDS's can often also be configured to block malicious traffic on detection. In this way the local network can be protected. An advantage of network IDS's can be placed on a central spot in the network, effectively protecting all machines behind it from known attacks. However, this also means that the machines which run the IDS, should be able to handle large volumes of traffic. Besides, encrypted traffic passing the IDS can not be scanned. Last, they depend on signatures for their workings, meaning they will only detect known malware.

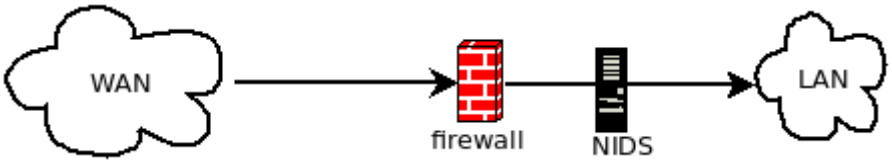

Figure 3: Typical setup of network IDS

For this research a network IDS is not a suitable solution because, although it can be configured to track POST requests, we expect it will not be capable of handling the large volumes of traffic, without requiring a heavily modified setup. Besides, since the scanner runs on the network, it is not possible for a network IDS to track the integrity of files on servers.

Web Application Firewall (WAF)

According to the ModSecurity Handbook[12] A web application firewall is defined as "*An intermediary element that enhances the security of web applications*". This intermediary element can be implemented as either a software add-on, process or a network device. There are many different solutions which offer this type of functionality. For this research only ModSecurity was investigated, since it integrates in the Apache webserver and is open source.

ModSecurity is an Apache module, which can be used to intercept, analyze and store HTTP traffic. The handling of HTTP traffic can be done based on specific rules, which can be set manually. Rules can be based on the contents of any of the HTTP-request and -response headers and body.

For this research ModSecurity is not suitable, although it can work with SSL-sites, it is not possible to use ModSecurity to track the integrity of files on the system.

Host Based Intrusion Detection System (Host IDS)

Host IDS's try to detect intrusion by monitoring files on the (local) mounted filesystem. Although several solutions exist, only OSSEC[13] provides a way to track POST requests and do file integrity checking. Unfortunately OSSEC could not be configured to track specific files. This makes OSSEC unsuitable for this environment. Customer sites can contain thousands of files. Only a small cluster with up to hundreds of customers will already have at least a few hundred thousand (100.000) files on it. Tracking all these files is unnecessary and will almost certainly lead to failures. Set aside, when doing this on a NFS-mounted share.

Final approach

Since we were unable to find a suitable existing framework, we decided to write our own IDS-script.

Testing the prerequisites

To determine whether it would be feasible to create a custom solution we need to find out the following:

1. Is the POST-method really used in the malware we want to detect?
2. What is the number files which need to be tracked?
3. Can this number of files be tracked over NFS?

Is the POST-method really used in the malware we want to detect?

To test whether this method is used, we collected some malware samples, which had been detected previously by the hosting company. These samples have been included separately with this paper. In the malware scripts we looked for the following code:

$_*POST[*

This piece of code indicates a predefined variable which is used to collect values from data sent using the POST-method[14]. Below the result:

malware sample:	**number of POST collectors found**
php.cmdshell.Err0R.226	34
php.cmdshell.indx	5
php.cmdshell.infectslab	3
php.cmdshell.unclassed.344	183
php.KroozUploader	1
php.uploader.max.541	1

Table 1: Count of *$_POST[* occurences in malware samples

As can be seen in table 1, all selected malware samples use the POST-method at least once in their scripts.

What is the number files which need to be tracked?

To find out how many files need to be tracked, we saved the URLs of successful POST requests in one week for the top seven sites. Then we tested the number of real files which we could relate these URLs to.

Site	URLs POSTed to	real files found	remarks
1	451	13	
2	37	0	
3	198	12	
4	0	0	site only uses GET
5	410	0	
6	344	1	
7	130	2	
Total	**1570**	**28**	

Table 2: Overview of files found for top 7 sites hosted at Byte Internet, sites are obfuscated to protect the owners privacy.

As can be seen in table 2, the real number of files which URLs point to is very small. This is probably caused by the fact that most sites are using rewrite rules[15]. The Apache mod_rewrite module uses a rule based rewriting engine to map a URL to a filesystem path. It however can also be used to redirect one URL to another URL or to invoke an internal proxy fetch.

Although probably not all URL's were translated correctly, this gives us at least an indication of the number of files which need to be tracked. The method used for relating URLs to real files is roughly the same as used in the final script.

Can this number of files be tracked over NFS?

Since there are only 28 real files found for the total of the top seven sites, we expect all files can easily be tracked over NFS.

Implementation

The main program is written in Perl, since it is the default language used for programming at Byte Internet. See appendix I for the full program. The program works as follows:

- A lookup table is generated to relate URLs found in an apache log to real files on the mounted filesystem
- A whitelist of files for which it is allowed to send POST requests to is read and for each of these files, the following are stored in the program:
    - A SHA-1 hash
    - Timestamp of last modification
    - Filesize
- Now the program start tailing the Apache access log and looks for POST requests
- If a POST request is found and it returned statuscode 200 (OK), the following is checked:
    - Can this request be related to an existing file and is it in the stored whitelist?
        - If not, alert
        - If so, check if the file has been altered since the last time it has been seen
            - If so, alert
            - If not, return OK

The program does not take any action based on the alert, since this is a prototype it would be too intrusive to delete files or block IP's. The requested files name, visitor IP's and visited URLs are however stored by the program and could be used for blocking or informing an administrator or website owner.

4. Results

We ran the created script against each of our collected samples and tested whether it would detect:

- execution of a remote command
- uploading of a file to the webserver
- editing of a file on the webserver

detected on: malware sample:	command execution	uploading	file editing
php.cmdshell.Err0R.226	yes	yes	yes
php.cmdshell.indx	no	yes	Na
php.cmdshell.infectslab	yes	yes	yes
php.cmdshell.unclassed.344	yes	yes	yes
php.KroozUploader	Na	yes	Na
php.uploader.max.541	Na	yes	Na

Table 3: Detection of specified actions from different malicious scripts. Na means, this functionality was not available in this script.

In table 3 the results of these tests are shown. For all examples the uploading of a file was detected and if possible the editing of a file. This is probably because “editing a file” in a browser, merely means downloading the file to the client and uploading back the new file. As far as applicable, it was only for one script impossible to detect the commands. This script used a GET-method for sending its commands. However it only allowed for the execution of very specific commands and not just any. In this specific case it concerned a ddos staging script using `ab` (Apache Benchmark)[16].

The script was also run on a live server for two full days. In these two days the script was able to identify 1704 scripts which was POSTed to. These scripts were in total 2055 detected. This means 1704 scripts new scripts were found and they were in total 351 times modified. These 1704 scripts belonged to 762 domains. The total number of domains accessible on this webserver however, was 2033. This could mean any of the following:

- There are 1271 domains which do not use POST requests.
- During the time of measurement no POST requests were found on to these 1271 domains
- POST requests were made, but we were unable to relate them to actual files

To find out whether there whether or not there were really no POST requests on these servers, we parsed the access logs, using the following command:

```
zcat access.log.{2,3}.gz | perl -alne'$h{$F[0]}++ if ($_ =~ m/POST/
 );END{$h{$_}&& print "total posted domains: ". scalar keys%h}
```

This resulted in a total of 603 domains, which were accessed by a POST request. Therefore it is quite possible that the other domains do not make any use of POST request or at least did not use them at the moment of measurement.
As we already mentioned, the Apache mod_rewrite module redirects virtual URLs to a real file on the filesystem. Although it is probably possible to relate more HTTP-requests to real files, when enabling logging of this module, it would also slow down the entire server. Besides we suspect the chances of having both the .htaccess file (where rewrite rules can be stored) point to a webshell as the upload of a webshell very slim. Therefore we focus our solution more on having a high performance, than being able to trace all HTTP-requests.

5. Conclusion

It is possible for a host based IDS to detect webshells and other malicious scripts, when they have a component of user interaction. It works and is able to run, not only for one site, but thousands. This is quite an advantage when comparing this method to, for example, mod_security.
A system like the one we wrote, provides a hosting provider and its customers with a tool to find suspicious behaviour in its early stages. This makes it possible to take action early and for example prevent other sites from being unreachable or ip-addresses from getting blacklisted.

Unfortunately our program could only be run in a small part of the hosting environment, but to be effective it should be run globally.
Although no real malware was found during execution, the program was able to trace POST requests to files. It was also able to keep track of the modification status of these files. To make a more significant analysis, it would be better to run it for a longer time on multiple servers. In that way the program has the ability to view more “live” malware instead of manually collected samples.

Research Questions

- Can we develop a method which detects interactive malware running on servers?

Yes the method we developed can detect interactive malware, when being executed.

  - What are the characteristics of this kind of malware?

The characterics of this malware is that a POST-method is used to send data and sometimes also commands to this malware. Besides, a distinction can be made between benign and malicious scripts.
- Benign scripts accepting POST requests are known beforehand.
- Malicious scripts are not known beforehand.
- Benign scripts can be modified by attackers to execute malicious code.

  - How can the characteristics be used to detect this malware?

The characteristics can be used to detect this malware by:
- Monitoring the Apache access logs for POST requests and determine to which files these requests correspond.
- Keep a whitelist of files which are allowed to be accessed by a POST request.
- Monitor the files on this whitelist for unauthorized changes

  - Is there a framework which is suitable for use in a shared webhosting environment?

No, although the reviewed frameworks could be used for one of the aspects of the research, they were not able to cover everything needed to succeed.

  - Can we integrate the detection method in a suitable framework?

Since no suitable framework was found, a custom program was built. Therefore, no.

6. Future work

Our program can be extended in a variety of ways.

- not only parse “POST” HTTP request method logs
- check the integrity of other files which can influence a site’s behaviour (.htaccess)
- ability to review modifications (if a file has been modified, what is modified)
- read rewrites to find all files (mod_rewrite)
- enable whitelist access and modification for customers
- extend alerting for administrators and customers, by for example sending an e-mail
- deny access to or delete suspicious files
- automatically ban IP’s requesting suspicious files

7. References

[1]http://news.netcraft.com/archives/category/web-server-survey/
[2]http://w3techs.com/technologies/overview/programming_language/all
[3] Martignoni, L. and Stinson, E. and Fredrikson, M. and Jha, S. and Mitchell, J. (2008), *A layered architecture for detecting malicious behaviors*
[4] Rieck, K., Trinius, P., Willems, C. and Holz, T. (2011), *Automatic analysis of malware behavior using machine learning*
[5] Farmer, D. and Venema, W. (2005), *Forensic Discovery*, chapter 6: Malware Analysis Basics.
[6] http://www.ietf.org/rfc/rfc2616.txt
[7] http://kris.blog.usf.edu/2011/06/22/wso-web-shell-2-5/
[8] http://en.wikipedia.org/wiki/Uniform_resource_locator
[9] http://support.microsoft.com/kb/208427
[10] http://httpd.apache.org/docs/2.2/mod/core.html#limitrequestline
[11] http://httpd.apache.org/docs/2.2/mod/core.html#limitrequestbody
[12] Ristić, I. (2012), *ModSecurity Handbook*, chapter 1: Introduction.
[13] http://www.ossec.net/
[14] http://www.w3schools.com/php/php_post.asp
[15] http://httpd.apache.org/docs/current/mod/mod_rewrite.html
[16] http://httpd.apache.org/docs/2.2/programs/ab.html

Appendix I Main program

```
#!/usr/bin/perl
=head1 NAME

byte-security-POST-IDS.pl

=head1 SYNOPSIS

count hits on POST files and raise an alarm/block if necessary

=head1 AUTHOR

S. Jethoe <sudesh@byte.nl>

=cut

use 5.010;
use strict;
use warnings;
use Data::Dumper;
use JSON;
use Digest::SHA qw/sha256_hex/;
use File::Tail;       # requires: 'libfile-tail-perl'
use File::Find::Rule; # requires: 'libfile-find-rule-perl'
use File::Slurp;      # requires: 'libfile-slurp-perl'

$|++;

######################## GLOBALS ###############################
my $FILE_HASH;       # hash of whitelisted files and their checksum
my $SHORT_LOOKUP;    # hash to lookup domain shortnames by base_url
my $HOST    = `/bin/hostname --fqdn`;
chomp $HOST;
my $CLUSTER = (split('\.',$HOST))[1];
say "$HOST\t$CLUSTER";
my $log_file = '/var/log/apache2/access.log';
my $path = '/home/users';
my $progpath = "$path/randrftp";
my $whitelist_file = "$progpath/whitelist.txt";
my $sha = Digest::SHA->new(1);
#Returns a new Digest::SHA object.  Allowed values for $alg are 1, 224, 256,
384, or 512.  It's also
#possible to use common string representations of the algorithm
(e.g. "sha256", "SHA-384").  If the
#argument is missing, SHA-1 will be used by default.

######################## FUNCTIONS #############################
```

```
sub generate_shortname_hash { # {{{ Generate hash to lookup shortnames by
base_url
    my $hash = {};
    # read existing hash if file exists
    if (-r "${progpath}/${CLUSTER}\_short_lookup.json"){
        $hash = decode_json (read_file("${progpath}/${CLUSTER}
\_short_lookup.json"));
    }
    else {
        my $rule = File::Find::Rule->new;
        $rule->mindepth (2);
        $rule->maxdepth (2);
        $rule->any( File::Find::Rule->symlink,
            File::Find::Rule->directory);
        $rule->start( '/home/users');

        while ( defined ( my $path = $rule->match ) ){
            my ($shortname,$base_url) = (split("/",$path))[-2,-1];
            $hash->{$base_url} = $shortname;
        }
    }
    return $hash;
} # }}}
sub read_whitelist($){ # {{{ read json hash or whitelist from file
    my $file = shift;
    my $whitelist;
    if ( -e $file ){
        open FILE, "<", $file or warn "Can't open file: $file";
        while (<FILE>){
            chomp($_);
            push (@{$whitelist},$_);
        }
    }
    my $hash = {};
    for (@{$whitelist}){
        $sha->addfile($_);
        $hash->{$_}->{digest} = $sha->hexdigest;
        my @stats = stat $_;
        $hash->{$_}->{size} = $stats[8];
        $hash->{$_}->{mtime}= $stats[10];
    }
    return $hash;
} # }}}
sub read_dirs { # {{{ read multiple logfiles from a dir
    my @files;
    my $log_dir;
    my $file_dir;
    if (not $ARGV[0]){
        print "No file(s) as argument running globals\n";
```

```
        print "Log_dir: $log_dir\n";
        print "File_dir: $file_dir\n";
        @files = <$log_dir/*>
    }
    else { @files = @ARGV; }
    return @files;
} # }}}
sub is_modified ($){ # {{{ test whether given file (fullpath!) is modified
    my $file = shift;
    my @stats = stat $file;
    my $hash = $FILE_HASH->{$file};
    if (not $hash){
        # "file $file does not exist in whitelist, update hash and alert!";
        $FILE_HASH->{$file}->{size}   = $stats[8];
        $FILE_HASH->{$file}->{mtime}  = $stats[10];
        $sha->addfile($file);
        $FILE_HASH->{$file}->{digest} = $sha->hexdigest;
        return "Unknown file found!";
    }
    if (($hash->{size} != $stats[8]) or ($hash->{mtime} != $stats[10])){
        # "$file size or mtime modified, will check hash!";
        $sha->addfile($file);
        my $digest = $sha->hexdigest;
        if ($hash->{digest} ne $digest){
            # !file modified
            # update hash to refer to new values:
            $hash->{size}  = $stats[8];
            $hash->{mtime} = $stats[10];
            $hash->{digest}= $digest;
            return "Known file was modified!";
        }
    }
    return 0;
} # }}}
sub generate_filenames { # {{{ generate array of filenames to check
    my $base_url = shift;
    my $post_url = shift;
    my $homedir = '';
    # split off trailing characters after index.php,
    $post_url =~ s/\.php.*/\.php/;
    my @post_urls = $post_url;
    my @base_urls = $base_url;
    if ( $post_url =~ m/\/$/ ){
        # check if post_url contains trailing /
        # /$ refers to -> index.{php,html}
        # remove $filename (it contains a trailing /)
        push @post_urls,($post_url."index.php",
            $post_url."index.html");
    }
```

```
    if ($base_url =~ m/.*\.(.*\..*)/){ # {{{
        # if base_url like www.mysite.nl
        # append stripped base_url to base_urls to check
        push @base_urls, $1;
        $base_url = $1;
    }
    if ( $SHORT_LOOKUP->{$base_url} ){
        # append shortname to homedir path
        # if corresponding entry in SHORT_LOOKUP hash is found
        $homedir = "${path}/$SHORT_LOOKUP->{$base_url}";
    }
    else { return 0; } # }}} return Null if homedir not found in hash
    my $filenames= [];
    for my $base (@base_urls){
        for my $post (@post_urls){
            push @${filenames}, "${homedir}/${base}${post}";
        }
    }
return $filenames;
} # }}}
sub tail_logfile ($){ # {{{
    my $log_file = shift;
    say "opening log:\t $log_file";
    if ($log_file =~ /\.gz$/) {
        open( FILE, "gunzip -c $log_file |") || die "can't open pipe to
$log_file";
    }
    else {
        my $ref=tie *FILE,"File::Tail",(name=>$log_file);
#       else {  open(FILE, $log_file) || die "can't open $log_file"; }
    }
    my $rx = qr/^(?<base_url>\S*)\s*(?<ip>\S*).*POST\s*(?<post_url>\S*)
\s*\S*\s*(?<status>\S*)/;
    while (<FILE>){
        if ($_ =~ $rx and $+{status} == 200 and $+{base_url} eq
'www.randomstream.nl' ){
            my $filenames_to_check = generate_filenames
($+{base_url},$+{post_url});
            next unless ($filenames_to_check); #only process if filenames
could be found
            for my $file (@${filenames_to_check}){
                if (-f $file){
                    my $file_status = is_modified $file;
                    next unless $file_status;
                    # only execute following if file status changed
                    #print $_;
                    say "ALERT: $file_status";
                    say "Source IP   :\t$+{ip}";
                    say "Base  URL   :\t$+{base_url}";
```

```
                    say "POST  URL   :\t${post_url}";
                    say "Suspect file:\t$file";
                    say '';
                }
            }
        }
    }
} # }}}
$SIG{'INT'} = sub { # {{{ write all found files to new whitelist on exit
    say "\nCaught INT, will write whitelist and exit.";
    # write new whitelist to file
    my $new_whitelist = "${progpath}/${HOST}\_whitelist.txt";
    open WHITELIST_FH, ">", $new_whitelist or
    die "Can't write to file:\t $new_whitelist";
    for ( keys %$FILE_HASH ) { say WHITELIST_FH $_ }
    close WHITELIST_FH or warn "Can't close file: $!";
    # write short_lookups to file
    write_file ("${progpath}/${CLUSTER}\_short_lookup.json",
        { atomic => 1 },
        encode_json($SHORT_LOOKUP));
    exit 0;
}; # }}}

######################## MAIN      ##############################
$SHORT_LOOKUP = generate_shortname_hash;
$FILE_HASH    = read_whitelist $whitelist_file;
# create initial hashes, including hash(digest), mtime and filesize
tail_logfile $log_file;
```