# A Generic Toolkit for Converting Web Applications Into High-Interaction Honeypots

Michael Müter[1], Felix Freiling[1], Thorsten Holz[1], and Jeanna Matthews[2]

[1] Laboratory for Dependable Distributed Systems
University of Mannheim, Germany
`Michael.Mueter@rwth-aachen.de`, `{freiling|holz}@informatik.uni-mannheim.de`
[2] Department of Computer Science
Clarkson University
`jnm@clarkson.edu`

**Abstract** In this paper, we present the design and implementation of a generic toolkit for turning arbitrary PHP web applications into high interaction honeypots in an automated fashion. We demonstrate the wide applicability of this approach by applying it to four popular existing web applications. Moreover, we show the effectiveness of our system by using it to analyze 70 actual attacks including 9 complete malware tools that were captured within a period of ten weeks. In addition, we present a method of driving traffic to a web-based honeypot by advertising it through *transparent links*. We conclude that our toolkit allows for simple and effective deployment of web-based honeypots based on existing PHP web applications.

**Keywords**: Honeypots, Web Applications, Intrusion Detection Systems

## 1 Introduction

Web application technologies like PHP, CGI, Javascript, and Ajax have made it much easier for people to construct and deploy services on the Internet. Unfortunately, this has opened a wide avenue for new attacks since it is as easy to unintentionally introduce new vulnerabilities into web applications as it is to intentionally introduce new functionality. Consequently, web applications have increasingly been the focus of attackers.

*Honeypots* are popular and effective tools for studying new attack patterns. A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource [16]. Honeypots are electronic decoys, that pretend to be normal system but are really waiting to be attacked and compromised for the purpose of tracking attackers. Honeypots are equipped with special monitoring software which makes it easier to study successful attacks in detail. *Honeynets* are networks of honeypots have proven to be a very effective tool in learning more about Internet crime like credit card fraud [11] or botnets [4] and as sensors in intrusion detection systems [3,9]:

The simplest form of a honeypot is a real vulnerable system that has been modified to include surveillance methods. Such a system is called a *high-interaction honeypots* because the attacker is able to fully interact with the honeypot just like a real system. This offers the best potential for analyzing all aspects of an attack, but also introduces risk that the attacker will use the capabilities of the system to attack others. A high-interaction honeypot must disguise itself as a real machine, hiding its surveillance methods to all users even if they have root privileges. This is usually done using very risky and resource intensive techniques like full system emulators [24] or rootkit-type software as in the *GenIII honeynet* [1]. To monitor automated attacks as for example those performed by autonomously spreading malware, such effort is not always required. So called *low-interaction honeypots* offer limited services to the attacker, for example by emulating only those parts of a service which are vulnerable. Low-interaction honeypots can typically be deployed with fewer resources because they are not fully offering the expected services and they also incur less risk. However, it is more likely that the attacker will cut short the attack before useful information can be learned either because the system does not support functionality needed for the attack or because the attacker suspects the system is a honeypot. A popular example of this kind of honeypots is *honeyd* [17], which is very easy to deploy (at least in comparison to a high-interaction honeypot).

Since web applications are so popular targets, the following question has been raised: How can we adapt honeypot technology to web applications? On the one hand, it is of course always possible to run a webserver together with a vulnerable web application inside a high-interaction honeypot. But this entails all resources and risks as described above. On the other hand, it is not entirely clear how to adapt the concept of a low-interaction honeypot to web applications. This is because we ideally want to (1) provide the functionality of a full web application inside the honeypot and (2) trace the entire user input throughout the attack and not stop at a certain point in monitoring. Therefore, web application honeypots fall somewhat outside of the classic low/high-interaction taxonomy. The corresponding research question here is: How can we construct web application honeypots so that we have the flexibility of high-interaction honeypots combined with the resource-consumption of low-interaction honeypots?

*Contributions.* In this paper, we explore the automated creation of high-interaction honeypots based on web applications. We present a framework to construct honeypots for web application based on the popular scripting language PHP. We introduce two tool: *Honeypot-Creator* and *HIHAT*. The first tool is a program that transforms an arbitrary web application into a high-interaction, web-based honeypot. Therefore, these honeypots offer the flexibility of high-interaction honeypots (they retain the full behavior of the original application) while using only "application level" resources. The second tool is HIHAT, the *High Interaction Honeypot Analysis Tool*. It allows us to efficiently and semi-automatically analyze the data collected via web application honeypots.

We also address the problem of *advertising* web application honeypots in this paper. We investigate different techniques based on *transparent links* on web pages, i.e., links which a human user cannot see but which an automatic search engine like Google will see and put into its search index. Attackers employ search engines like Google to find their targets. This technique, known as *Google hacking* [7], makes it necessary to enter the URL of the honeypot into the index of search engines, potentially opening it for accesses from normal Internet users. We experiment with different types of transparent links and show that this kind of links can be used to lure malicious traffic efficiently to our honeypot.

We evaluate honeypots constructed using our framework in several ways: Firstly, we analyze the performance overhead of the logging mechanism, showing that the overhead is almost negligible (below 10% overhead in execution time) and therefore cannot be used to reveal the existence of the honeypot code within the web application. Secondly, we evaluate the data collected by four popular web applications (PHPMyAdmin [14], PHPNuke [15], phpBB [13], and PHPShell [5]) which were turned into honeypots using Honeypot-Creator. This data gives insight into the techniques used by attackers to identify vulnerable applications using search engines as well as the vulnerabilities most often exploited by attackers to take control of vulnerable machines. Within ten weeks, we were able to observe 70 attacks against four honeypots, ranging from SQL injection attempts over file inclusion attacks up to automated attacks of web-based worms.

*Related Work.* Threats to web applications have been previously investigated using intrusion detection systems [10] and classical honeypots [20] and a lot of information about attack techniques has been gained. Honeypots generated with our approach aim at collecting more such information using less resources.

Our approach is not the first tool in the field of web application honeypots. *GHH* (the Google Hack Honeypot) [6] is a project within the Honeynet project [16] which creates a vulnerable web application from scratch. The main focus of GHH (and of its descendent PHP.Hop [12]) is to collect data about the search patterns attackers use to identify vulnerable applications. In this sense (and in contrast to our approach), GHH is a low-interaction honeypot and does not provide real web application functionality.

Another related area is that of so-called *hitlist worms* [23,22,25]. A hitlist worm can use (among other information sources) search engines to collect large lists of vulnerable machines before spreading. Especially worms that target web applications are very dangerous and must be investigated before they spread [18]. Typically, these *search worms* can only be observed by search engine operators or victims. Honeypots created by our framework can be used in this endeavor: Essentially, we *become part of the hitlist* and are thus able to learn more about this kind of attacks.

*Roadmap.* This paper is structured as follows: Section 2 presents a set of requirements a web-based honeypot has to fulfill. In Section 3, we describe a generic toolkit for turning an arbitrary web application into a high-interaction honeypot, and Section 4 focusses on the problem of driving traffic to the honeypots. Section 5 presents the results of our work and we conclude the paper in Section 6.

## 2 Requirements For Web-based Honeypots

Before describing the development of our generic toolkit for converting web applications into high-interaction honeypots, we first identify a set of requirements that the ideal system of this type would possess:

– *Functionality*: The system functionality should be the same as the identical web application would offer as a non-honeypot system.
– *Performance*: The system performance should be only slightly lower compared to the performance of the non-honeypot web application.
– *Input Information Coverage*: The data capturing capabilities of the honeypot should be comprehensive and cover *all* information sent by the attacker to the honeypot.
– *External Logging*: Captured data should be stored externally on a separate and secured system to ensure integrity of the logged data.
– *Security*: The system should prevent an attacker from using the honeypot as a stepping stone to cause harm to other non-honeypot systems.
– *Universality*: A universal honeypot system is required which does not only cover a single web application.

These six requirements are based on the core functionality a honeypot system should have. In addition, the following two requirements refer to the basic concept necessary for data analysis:

– *Analysis Support*: The system should provide different means to support the process of analyzing the collected data.
– *Extensibility*: The honeypot and its analysis capabilities should be extensible in such a way that it can easily adapt to new attack patterns and exploits.

A system that complies to all of these requirements would be an optimal system to create. Nevertheless, this causes problems as the realization of all requirements is a difficult task. Some requirements stand in contradiction to each other. For example, it is not possible to allow full functionality in any case and simultaneously ensure that the security never gets violated: if we create a high-interaction *PHPShell* (a web application with which an administrator can execute arbitrary commands on a machine), an attacker can immediately use the shell to start further attacks against other non-honeypot systems. Furthermore, ensuring comprehensive logging, no matter what an attacker transmits to an arbitrary web application, is a non-trivial task. Depending on the amount of transmitted log data, we may also perceive effects in performance.

## 3 A Generic Toolkit For Creating High-Interaction Web-based Honeypots

### 3.1 *Honeypot-Creator*: Converting Web Application to Honeypots

In this section, we describe the design and implementation of the *Honeypot-Creator*, a toolkit to transform an arbitrary web application into a *web-based* *honeypot*. Our basic approach is to start with an existing web application and convert it into a honeypot in an automated and generic way. This involves adding capabilities for logging important data about an attack and containing the existing application within a honeynet to protect others.

Most of the prevalent web applications are written in PHP. Thus, we chose to focus on PHP based web applications in this version of Honeypot-Creator. Nevertheless, the ideas presented in this paper could also be applied to web applications written in other programming languages used on the web.

To automatically identify the data we will log, we begin by observing that all traffic coming to a web-based honeypot will use HTTP. This protocol provides two basic transmission methods:

– The *GET method* means that form data is encoded into the uniform resource locator (URL) by the web-browser. This method is commonly used when the form processing is idempotent in such a way that no status changes will apply by performing the request. The maximum amount of data that can be transferred with a single GET request is limited and depends on the maximum size of a URL. For example, Microsoft Internet Explorer has a maximum URL length of 2,048 characters, minus the number of characters in the actual path.
– The *POST method* describes a procedure to transmit data that is meant to be used for non-idempotent queries. Every request that results in a status change is non-idempotent. In contrast to GET requests, form data appears within the body of a message when using POST requests. Typical examples for such non-idempotent requests using POST are file uploads or sending e-mails. The amount of data transferable via POST is larger and theoretically unlimited. In practice, however, the maximum length depends on the settings of the web server.

In order to log the information an attacker enters into a web application, we need to track these two transmission methods. This can be achieved by monitoring four crucial arrays, which are provided by PHP within a global scope:

1. `$_SERVER`: This array contains the main server information such as headers, paths, and script locations. The entries within this array are created by the webserver. There is no guarantee that every webserver will provide all of them, servers may omit some, or provide others. However, a large number of these variables are part of the CGI 1.1 specification [8], thus it is reasonable to expect those.
2. `$_GET`: This array contains all data transferred to the server via HTTP GET requests. This type of requests typically includes data like session IDs or path information, referring to clicks of the user inside the web application.
3. `$_POST`: This array contains all data transferred to the server via HTTP POST request. These requests are used for similar purposes as the GET request. The differences between both types of requests were outlined above.
4. `$_COOKIE`: This array contains all data transferred to the server via HTTP cookies. Web applications typically use cookies in order to store data like

configuration settings and session information, but also login information like username and password can often be found here.

These arrays contain all information that is needed to track every step of an attack against an arbitrary web application. If we thus monitor all these arrays and correlated the collected data, we are able to monitor the exact attack traffic for any PHP-based web application.

Based on the chosen design approach, the Honeypot-Creator transforms an arbitrary web application into a high-interaction honeypot. This is achieved by automatically inserting special *logging code* into each relevant file of the web application. This approach has the advantage that it allows us to monitor all kinds of direct file access an attacker might perform. Even files that the web application only uses in an indirect manner could be accessed immediately by an attacker. The logging code ensures that all his actions are monitored and all information is logged.

The logging code itself needs to comply to two major requirements:

1. The code is not allowed to change the behavior of the web application or interfere with the rest of its source code in any way.
2. Moreover, the attacker should not be able to detect the logging code when he performs the attack.

Ensuring non-interference of the logging code with the rest of the web application is a non-trivial task. This is due to the fact that the support for arbitrary web applications may also result in arbitrary lines of code to appear in the source files. Nevertheless, implementing the following measures achieve a good level of separation from the application and ensure a high level of independence of the logging code:

– Each PHP file opens a separate connection to the logserver and transfers its data to the database. This avoids interferences which may occur when data would be collected first.
– Every PHP file closes the connection to the logserver as soon as the transfer is completed. Although this results in a higher overhead and a loss of performance, it ensures comprehensive logging even if an attacker accesses files that are not meant to be accessed directly. Furthermore, it decreases the chances of detection by an attacker due to pending connections.
– The logging code is not allowed to include other files in order to perform logging. This of course causes redundancy, but preserves the independence of the logging code. It helps to ensure that every access to the honeypot will still be monitored, even if an attacker would manage to delete some files of the web application on the honeypot.
– All variable names used in the logging code are obfuscated in such a way that they do not interfere with the web application's source code.

The use of PHP ensures the code gets executed at the server side and cannot be downloaded by an attacker. This decreases chances of detection. The

Honeypot-Creator inserts the logging code in the following way: first, it recursively crawls the directory of the web application and creates a list of all PHP and HTML files. Image files and other filetypes can be neglected. The code cannot be inserted at the end of each file because the exit point during the execution is not known, so there would always be a chance the code does not get executed at all. Hence, the Honeypot-Creator performs all insertions at the beginning of each file. The logging code itself needs to read out the content of the four crucial arrays identified above. Afterwards, it serializes the data and stores it in an external SQL database located at the logging server.

An optional *tag name*, like *PHPMyAdmin* or *PHPNuke*, can be specified before the logging code is merged with the web application in order to identify the corresponding source module which was accessed. This tag name becomes part of the logging code and is supplemented when an entry is stored in the database. This feature supports extensibility and facilitates the analysis.

## 3.2 *HIHAT*: Analyzing the Collected Attack Traffic

In this section, we describe *High Interaction Honeypot Analysis Tool* (*HIHAT*), our tool to support the analysis process for the data collected by a high-interaction, web-based honeypot. Different automated means are provided which facilitate a user who is monitoring and analyzing the acquired data. The combination between automatic data preparation by the tool and manual monitoring by a person ensures the highest detection rate for attacks and interesting incidents and the lowest rate of false positives. Furthermore, only a reasonably low amount of effort for the user is required for monitoring and analysis due to the high level of automatization.

In terms of data analysis, the chosen design approach, which allows arbitrary web applications to be transformed, now results in several challenges. Thus, different problems and issues have to be considered in the design of HIHAT:

- The tool should display each access that was made to the honeypot web application. For example, this includes every single click within the application and every entry in a form. Thus the logging results in a vast amount of data. Nevertheless, all the information needs to be presented by the tool in such a way that the person who is monitoring the honeypot can quickly overview the information and extract the important data.
  Furthermore, the importance and impact of data depends on the web application it originates from. In one application a variable called `username` may be very important and at high risk of being attacked, whereas in another application, the same variable could be negligible.
- As arbitrary web applications are involved, it is not possible to focus on some set of variables. All kind of variables with different names, contents, and lengths have to be taken into account.
- The tool should identify known attacks automatically as good as possible. An example would be the automatic identification of all SQL injection attacks. But again, even for known attacks, this is not trivial since arbitrary web

applications can be involved: whereas in one case an application may use parts of SQL statements to perform normal operations, SQL commands may refer to an attempted SQL injection attack in another case with a different variable or application. While it may be possible to automatically identify all SQL statements in the data, the decision if a specific statement actually denotes an attack or is part of the application's normal behavior cannot simply be made automatically.

– The tool should support the detection of new attacks as good as possible. Certainly not all new attacks can be detected automatically. Nevertheless the system could try to identify patterns, strings, or names that are more likely to represent an attack. This helps in identifying *zero day attacks*, i.e., attack vectors which are unknown at the time of attack.

As described before, the tool supports the person who is monitoring the honeypot and analyzing the collected data. The combination of automatic filtering by the system and human inspection is most likely to yield the highest accuracy in terms of information collection about attacks as well as the lowest rate of false positives. All issues identified above led to the following design decisions for HIHAT. First, two main views are supported: On the one hand, there is the *overview mode*, which allows the user to get a quick general view about all the activity that was captured. On the other hand, the tool allows to switch into a *detailed viewing mode*, where all information about a single event is provided. The detailed view provides an overview of all data transfered to the web application via the four different arrays provided by PHP.

The tool automatically filters for attack patterns that can be derived from known attacks like SQL injection or file inclusion attacks. This is achieved via regular expression which we derived from an analysis of known attacks against web applications. For example, we search for patterns like `INTO OUTFILE`, `script`, or `include` which indicate possible attacks. Furthermore, we include generic attack patterns that identify common commands executed by an attacker after a compromise, e.g., the commands *id* or *uname*. The tool provides high extendibility because it supports the easy supplement of new patterns.

The tool includes an *automatic download function*: the idea is to automatically detect all additional tools an attacker may try to download and use, e.g., via the commands *wget*, *curl*, or similar tools. In addition, generic URLs are identified via a regular expression. HIHAT automatically retrieve a copy of all identified resources and stores them in a specially secured place. This way, they cannot cause any harm by the attacker, but are available for a detailed examination and research at a later point of time. Moreover, HIHAT has support for *location mapping* in order to visualize the origin of the attacking IP address. Lastly, the tool generates various statistics in order to give an overview about the traffic of the honeypots. This includes for example the total number of hits, a ranking of the most often accessed files, a list of search engines and patterns which were used in the HTTP-referrer, and a pie chart visualizing the hits per honeypot module.

The overview mode helps the user to recognize attacks quickly and gain an impression about the traffic and the activities of a honeypot at a glance. When the user has spotted an interesting entry in the overview, he can access further information by switching to the detailed viewing mode. HIHAT is also equipped with a search function in order to allow the user to quickly find the desired information and to facilitate the handling of large amounts of data. It can for example search for IP addresses, specific attacks, or date and time.

One of the development goals was to design a toolkit which provides a high level of extensibility. In order to achieve this goal, HIHAT has a modular structure. Each of the modules can describe individual filtering patterns for white- and blacklisting which are applied in the overview. A honeypot running *PHP-Myadmin* for instance may need different variables to be filtered than a system running *PHPNuke*. As described in Section 3.1, the Honeypot-Creator can be supplemented with an optional *tag name* which labels every access to the honeypot. This tag is used by the analysis tool in order to provide additional filtering patterns. These modules are loaded automatically and can be easily created with the help of a provided template.

## 4 Driving Traffic to Honeypots with Transparent Links

Search engines provide the possibility for attackers to look for exactly the type and version of a vulnerable application in which they are interested. Instead of performing random scans, they can precisely focus their efforts on targets which match their criteria. Therefore attackers are able to conduct the attacks in a much more efficient way and they can create a *hitlist* of targets likely to be vulnerable to a specific attack [25]. An example of such a query might be "*'1.7' inurl:phpshell filetype:php*", which lists vulnerable versions of the program *PHPShell* [5].

In order to attract attackers to our honeypot, we need to catch their attention and interest. Since attacks on web-based applications commonly use search engines in order to find their victims, we want our honeypot to be listed by the indices of popular search engines. Once the honeypot is indexed, attackers that use search engines are drawn to the system, which results in more traffic being driven to the honeypot. Basically the trick is to *become part of the hitlist*.

The important question is how to add the honeypot to the index of a search engine. Nowadays, the search index is commonly constructed automatically with the help of so called *web spiders*. Web spiders are programs which crawl the World Wide Web in a methodical and automated manner with the intent of creating an index about the crawled contents. As search engines do not provide a method for directly modifying search results for such a research purpose, we need to use the behavior of the web spiders themselves in order to complement the search index with information about our honeypot. Specifically, we add links to our honepot in existing, regularly crawled web pages.

There are two problems with this approach. First, the details about the exact behavior of the web spiders are usually kept secret, in order to avoid abuse or distortion. Neither the exact construction criteria for the ranking of the index are public, nor information about if and how the content of a web page gets rated. Some documents describe the basic principle of the algorithms [2], but the exact details are commonly not known. Secondly, we cannot place arbitrary links to our honeypot on a website. This is due to the fact that not only web spiders or attackers may follow the link, but probably also many benign users who are just visiting the webpage. By following the link, these users would cause false positives in our logfiles and incidentally also increase the chance that an attacker reveals the true purpose of this link for our honeypot.

In order to tackle these problems, we choose the following solution: a specially crafted link is required, which satisfies two requirements. First, it needs to be *invisible* to a benign Internet user surfing the web page. Second, it is still recognized by web spiders crawling the page. A link of this type is named *transparent link* since only web spiders can see it.

We have to keep in mind that transparent links represent an issue where web-based honeypots strongly differ from traditional honeypots where every access is considered to be an illicit use of that resource. Instead, web-based honeypots need to be indexed by web-spiders in order to catch a reasonable amount of interest and to work properly as we explained above. Hence, in this point, web-based honeypots pursue a different concept than other honeypots: we need to *advertise* the presence of the honeypot. Nevertheless, the main value for both types of honeypots lies in the unauthorized or illicit use of that resource.

Our experience shows that a high position in the ranking is not required in order to attract automated tools or even manual attacks. However, if a web spider does not follow a specific link at all, of course it is useless for this purpose. In the following, we explore possible ways for transparent linking and show suitable classes of such links which meet our two criteria.

We experimented with a number of different possible types of links that are invisible or nearly invisible to a human viewer. Some types of links are not noticeable at all when the page is displayed in a web browser, whereas others may be recognized at the screen up to some degree, for instance as a single pixel or little dot at the website. Of course, the visibility may also depend on the web browser a user choses to look at the page. However, at all times, the link can be identified in the source code of a website and this denotes the point which ensures that web spiders have the chance to detect and crawl the link. In the appendix, we list twelve different types of transparent links, and here we give just one example of a hyperlink in an image map of size zero:

```
<img src="crawler.gif" border="0" usemap="#ImageMap">
<map name="ImageMap">
<area shape="rect" coords="0,0,10,0" href="HONEYPOT_URL">
</map>
```

In order to test which of our link types are followed by web spiders, we established the following test setup: A homepage starting with *index.html* was created which served as an entry portal to the test area. From there, two more

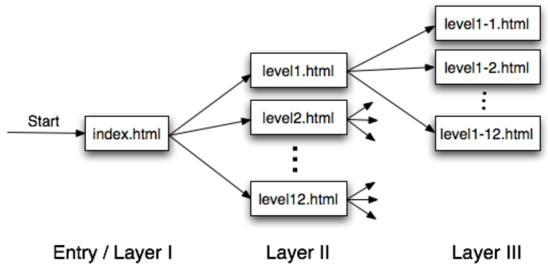

**Figure 1.** Layout of test setup for transparent links

layers of links were referring to HTML documents. The document contents were generated randomly, in order to emulate a regular web page. The *index.html* and the first layer of referenced files contained further links, one for each type of transparent link. This results in a total number of 157 files included in the test. Figure 1 shows the layout of the test area. Every access to one of these files was monitored and logged.

The given setup allows to determine which types of transparent links are recognized and followed by web spiders. Furthermore it does not only show if spiders quickly check a link, but also if they continue crawling the entire content of a page and subpages.

*Test results.* We focus now on the test results we obtained for the following three search engines, which are currently the most important ones:

1. *Googlebot* by Google (`http://www.google.com/bot.html`)
2. *Yahoo! Slurp* by Yahoo! (`http://help.yahoo.com/help/us/ysearch/slurp`)
3. *msnbot* from MSN Search (`http://search.msn.com/msnbot.htm`)

Interestingly, the results were very similar for all three web spiders. Every spider visited the test area in irregular periods and made only a few requests every visit, probably in order to avoid to overload the site and to demand too many resources. The only changing aspect was the time the different pages were accessed. For Yahoo! Slurp, the entire process of crawling took about 39 hours, comprising the total time between the first access and the last visit necessary to crawl the complete site. Msnbot needed about 26 hours and Googlebot was performing less intense inquiries, taking about 124 hours. All spiders switched between different IP addresses at some point during the crawling process.

The behavior regarding the recognition and crawling of the links was identical for all spiders: of of our transparent link types were followed by spiders except one (hyperlink hidden in a comment). We therefore conclude that transparent links are an effective method of drawing attackers to our page while avoiding accidental access by legitimate web surfers.

## 5 Results

In order to evaluate our ability to turn arbitrary PHP web applications into high-interaction honeypots with our tools, we tested our approach on the following four web applications:

- *PHPMyAdmin* [14]: a tool written to perform management and configuration of MySQL databases via a web interface. Its functionality comprises the creation, deletion, and modification of database tables and contents as well as the management of users and user privileges.
- *PHPNuke* [15]: a news publication and content management system based on PHP and MySQL. *PHPNuke* allows users to publish their own news and items online. In the background, it provides support for different themes, languages and additional modules.
- *phpBB* [13]: The name phpBB is an abbreviation for *PHP Bulletin-Board*, implementing a web based Internet forum. It supports different themes, template styles, and languages.
- *PHPShell* [5]: *PHPShell* is a shell wrapped in a PHP script, which allows to execute arbitrary shell commands. It is a very powerful tool and can for instance be used to perform administration and maintenance operations via a web interface.

These applications have been selected because they are known to represent very popular targets for web-based attacks. *PHPNuke* for instance has a long history of security vulnerabilities and has been attracting attackers for a long time. *PHPMyAdmin* is a popular and widely-used tool for database administration at the Internet and thus represents an attractive target for an attacker. The mixture of these applications has a high chance of allowing us to observe some interesting attacks and a reasonable amount of traffic on our honeypot.

As a test platform, we used an Intel Pentium M 1.5GHz with 1.25GB of RAM. In order to transform the web applications into web-based honeypots, we used Honeypot-Creator (see Section 3.1). We used an individual tag name for each web applications in order to facilitate the analysis process later on. The transformation took less than 30 seconds for each application. Since this process has to be executed only once when the honeypot is set up, the needed time seems to be reasonable and the overhead can be neglected. Each of the transformed web applications was uploaded onto a webserver in order to complete the setup of the web applications. In addition, we added 18 transparent links on other web pages that pointed to the honeypot. These web pages were selected ad-hoc, e.g., the web page of the lab or several private weblogs. We used these transparent links to have several locations that could be picked up by search engines so that our honeypot was included in the search index.

### 5.1 Performance

We tested the performance of every transformed application to measure the overhead introduced by the logging code. Table 1 provides an overview of the

| | normal PHPNuke | PHPNuke with logging code |
|---|---|---|
| login | 2,32s | 2,39s (+ 3%) |
| load starting page | 1,59s | 1,77s (+ 11%) |
| access top 10 | 1,60 | 1,63 (+ 2%) |

| | normal PHPMyAdmin | PHPMyAdmin with logging code |
|---|---|---|
| login | 4,15s | 4,29s (+ 4%) |
| load starting page | 1,16s | 1,25s (+ 8%) |
| select database | 2,11s | 2,33s (+ 10%) |

**Table 1.** Performance of web applications with and without our logging code

measurements for two of the four applications. The results for the other two applications are very similar. The overhead for the honeypot logging code is rather small, adding on average less then ten percent compared to the original web application. Such a small overhead is likely not noticeable by an attacker who interacts with the web-honeypot via the Internet.

The small overhead is mainly caused by a flat database structure, i.e., the contents of the four crucial arrays are stored in one single table. This ensure that the performance requirement is met and the chance of detection for the honeypot is kept to a minimum. As a side effect, the performance of the analysis tool is improved since only a single table has to be queried.

### 5.2 Observed Attack Traffic

All activity at the honeynet was monitored for a period of about ten weeks between 15th of December 2006 and 28th of February 2007. We start to describe the observations concerning the overall traffic. Afterwards, we depict the results we observed which are related to specific attacks.

The total number of hits we received within the given time period is 8177, caused by 765 distinct source IPs. With a total of 6005 (73.44%), most of these hits were caused by web spiders. We recognized an amount of 27 different web spiders accessing our honeypot. Obviously the transparent links we have been setting up were working nicely. The remaining 2172 hits (26.56%) were caused by different kinds of attacks.

When analyzing the attack distribution by the four different web-based honeypots, *phpBB* is the application which attracted most hits with 44.96%, followed by *PHPNuke* with 35.25%. Although *PHPMyAdmin* is a very popular application at the Internet, the amount of traffic we received is comparatively low with 14.38%. An additional 4.50% of traffic targeted *PHPshell*. These percentages include all traffic recognized at the honeynet, also hits which are caused by web spiders. For testing purposes, a few other, small modules were installed which served as tools to check and validate the proper functionality of the system. Table 2 summarizes the hits per module. *PHPNuke* and *phpBB* allow a user to view large parts of the application without the need to login, whereas *PHPMyAdmin*

| Web-Honeypot Module | Number of Hits | Percentage |
|---|---|---|
| phpBB | 3676 | 44.96% |
| phpNuke | 2882 | 35.25% |
| phpMyAdmin | 1176 | 14.38% |
| phpShell | 368 | 4.50% |
| Others | 75 | 0.92% |

**Table 2.** Hits per web-honeypot module

| Source | Number of Hits | Percentage |
|---|---|---|
| No parameter | 1203 | 14.72% |
| HTTP GET | 6874 | 84.07% |
| HTTP POST | 99 | 1.21% |
| HTTP COOKIE | 826 | 10.10% |

**Table 3.** Source of transferred data targeting the web-honeypot

requires an immediate login to access the application. This might be a reason for the comparatively low level of traffic recorded for *PHPMyAdmin*.

In 14.72% of the hits, just the webpage itself was requested without transmitting further parameters to the application. When parameters were transmitted, the HTTP GET method was used in 84.07% of the cases. Although only few requests were using the HTTP POST method, these hits contained a comparatively high number of attacks that were detected. Almost all these POST requests were malicious in nature. This can be explained by looking at the functionality of the module *PHPShell*, which exclusively uses HTTP POST to send the command inserted into the online form to the web application.

Orthogonal to this, the request can also use cookie parameters in addition to GET or POST requests. This additional cookie data was transferred in about 10% of the requests. Interestingly, we were not able to observe any request which was trying to use HTTP cookies to perform an attack. Of course, this may change with a longer observation time or a different set of applications that is deployed. The sources for the transferred data are summarized in Table 3.

The HTTP_REFERRER was set in 22.20% of the requests (1815 hits). This field indicates the web page from the current request is coming from. We can use the HTTP referrer to discover which search engines the attacker used in oder to find our honeypot. For 240 referrer, we could determine the search engine, i.e., at least that many attacks were caused by hitlists generated via search engines. The remaining referrer are mainly caused by subsequent requests to a web application following after the first request. An analysis of the search engine referrer information showed that in about 97% of all queries (233 hits in total) the attackers used Google to find our honeynet. In the remaining 3% of the cases (7 hits), the referrer used either Yahoo! (6 hits) or MSN (1 hit). This shows that

| **Content of HTTP REFERRER** | **# Hits (Percentage)** |
|---|---|
| http://www.google.com/search?q=inurl:phpmyadmin[..] | 8 (3.33%) |
| http://www.google.it/search?q=allinurl:.org/phpmyadmin/[..] | 4 (1.67%) |
| http://www.google.com/search?q=phpmyadmin import [..] | 3 (1.25%) |
| http://www.google.com/search?[..]q=ocrad gocr | 3 (1.25%) |
| http://search.yahoo.com/search?p= Show MySQL system variables phpMyAdmin [..] | 3 (1.25%) |
| http://www.google.no/search?[..]q= Powered by PHP-Nuke inurl:/admin.php | 2 (0.83%) |
| http://www.google.com/search?[..]q= remove All logos and trademarks phpnuke | 2 (0.83%) |
| http://www.google.lv/search?[..]q=php-nuke 7.0 modules[..] | 2 (0.83%) |
| http://www.google.co.za/search?q= "Character Sets and Collations" phpMyAdmin 2.5.[..] | 2 (0.83%) |
| http://www.google.com/search?q= PHPNuke - create super user right now [..] | 2 (0.83%) |

**Table 4.** Top 10 HTTP_REFERERs monitored via high-interaction web-honeynet

search worms mainly use Google in order to find vulnerable hosts, but other search engines can also be utilized by an attacker.

The top 10 of the HTTP referrer are presented in Table 4 in a shortened form. Additional options which attackers used to specify their requests to the search engines in more detail, like language and browser settings, character set definitions, and search range limitations, have been truncated in order to provide a better overview. In the table this is indicated by *[..]*.

In our observations, the search parameters `inurl` and `allinurl` were often used in order to specify the target application attackers were looking for. Sometimes also the version of the application was determined, which helps the attacker to find victims which run a vulnerable version of the software that can actually be exploited. For instance, we recognized searches for "`"Character Sets and Collations" phpMyAdmin 2.5.`" which lists older versions of *PHPMyAdmin* which are immediately accessible for an attacker without needing login data. Attackers presumably use such techniques to build accurate hitlist which only contain vulnerable hosts in order to maximize their success rate.

Other queries we observed were looking for the string "`PHPNuke - create super user right now`". This request aims at unexperienced users which have been installing *PHPNuke* without a proper configuration and thus allow the attacker to acquire administrator privileges immediately.

The number of hits and percentages for each of the top 10 referrers are comparatively low, which is due to the fact that many different versions of search requests were found in the parameter HTTP_REFERRER. This results in various numbers of different queries each of which comprises a different modification or variant in the search request.

| Web-Honeypot Module | Number of Attacks | Percentage |
|---|---|---|
| phpNuke | 45 | 64.29% |
| phpShell | 13 | 18.57% |
| phpBB | 7 | 10.00% |
| phpMyAdmin | 5 | 7.14% |

**Table 5.** Successful attacks per web-honeypot module

| Name of Attack | Number of Attacks | Percentage |
|---|---|---|
| SQL-Injection | 41 | 58.57% |
| File-Inclusion | 13 | 18.57% |
| Command-Injection | 8 | 11.43% |
| Directory-Traversal | 3 | 4.29% |
| Others | 5 | 7.15% |

**Table 6.** Distribution of types for successful attacks

After showing various results concerning the general traffic we were monitoring at our honeynet, we now present the observations which are related to specific attacks that could successfully be detected by HIHAT.

## 5.3 Overview of Observed, Successful Attacks

The total number of hits containing a successfully detected attack was 70. Table 5 shows the distribution of attacks per module. One interesting aspect is that *PHPNuke* was attracting most of these attacks without receiving the majority of hits. Therefore, *PHPNuke* has a higher ratio of observed attack patterns per hits, making it appear to be a more attractive and worthwhile target for attackers than the three other web-based honeypots. For *PHPShell*, this ratio is even higher, probably due to the various and powerful possibilities the application provides for an attacker, which make it an attractive target.

Table 6 displays the distribution of the different attack types. SQL injections build the vast majority of the attacks, together with remote file inclusions they represent nearly 80% of all attack types we observed on our honeypots. The main reason for the high percentage of SQL injections probably lies in the selection of modules we deployed. In several versions of *PHPNuke*, a high number of SQL injection vulnerabilities are known. Hence, deploying the module *PHPNuke* strongly increases the chances to monitor this type of attack.

When looking at where exactly the attack vectors are exploiting the different applications, we found out that a single file of an application often contains distinct variables which are attacked. Attackers use different attack patterns to target these variables. Table 7 displays the top 10 variables which were used to insert attacks into the applications and also points out the vulnerable source file of the corresponding web application.

| **Source File** | **Name of Variable** | **Number of Attacks** | **Percentage** |
|---|---|---|---|
| /phpnuke/modules.php | query | 15 | 27.14% |
| /phpshell/phpshell.php | command | 12 | 17.14% |
| /phpnuke/modules.php | cid | 6 | 8.57% |
| /phpnuke/modules.php | name | 6 | 8.57% |
| /phpBB/includes/functions.php | phpbb_root_path | 4 | 5.71% |
| /phpnuke/modules.php | forwhat | 3 | 4.29% |
| /phpnuke/modules.php | instory | 3 | 4.29% |
| /phpmyadmin/main.php | lang | 2 | 2.86% |
| /phpnuke/modules.php | lid | 2 | 2.86% |
| /phpBB/posting.php | message | 2 | 2.86% |

**Table 7.** Top 10 variables carrying attack patterns and their vulnerable host files

For instance, the variable "`query`" in the file *modules.php* of *PHPNuke* comprises a vulnerability which is used by attackers very often to carry out an SQL injection and is counting the highest number of hits. This vulnerability is very well known and was reported in the security advisory SA17543 by Secunia [21].

A total of nine unique files were captured via the automatic download function of HIHAT. The download function is triggered each time HIHAT detects either a URL in the request or an attack pattern which indicates the usage of a tool to download additional content to the honeypot. The captured files were typically additional PHP scripts which the attackers tried to include via a vulnerability. The analysis of these files provided us with more information about typical web-based attacks.

Of course, all these results cannot be seen independently from the selection of applications that are used as decoys. Different applications contain different vulnerabilities. Once a certain vulnerability becomes known for a specific web application, more traffic can usually be observed targeting this weakness. The more popular and widely-used a web application at the Internet is, the more attractive it becomes for an attacker. In general, a different selection of modules therefore can usually result in a different distribution of traffic, attacks, and attack types.

## 5.4 Sample Attacks

This section describes some of the attacks we have been monitoring on our honeypots with the use of the Honeypot-Creator and HIHAT in more detail. If the attacker attempted to download malicious files, the contents are examined and tested within the sandbox of a virtual machine.

*Command Injection Example.* We start with a brief example of a command injection which attempts to exploit *PHPNuke*. The following HTTP GET request was monitored when the attacker was accessing the file *modules.php*:

```
name=Forums
highlight=%2527.$poster=%60id%60.%2527
```

The attack aims at a known remote command execution vulnerability in *PHPNuke*, which is based on the fact that the variable `highlight` is not filtered correctly. In the second line of the request we see that the attacker tries to inject and execute the command "*id*" into that variable. The command identifies the current user and denotes a typical test done by attackers in order to check a system for a suitable command injection vulnerability.

*File Inclusion Example* The next example deals with a typical remote file inclusion and explains the tool attackers tried to download and use on our honeypots. The attack itself consisted of just a single (sanitized) request:

```
/phpBB/includes/functions.php?phpbb_root_path=http://XXX/c99.txt
```

The attacker attempts to use a vulnerability in the `phpbb_root_path` variable in order to download the file *c99.txt* from a remote server and include it in the web application. A copy of the file was automatically captured by HIHAT and stored in the database which allowed us to analyse its content. *c99.txt* is actually a PHP script which allows an attacker a web-based backdoor to a compromised machine. Via this script, the attacker can for example create files, execute arbitrary commands, or list files and directories.

*Self-Propagation Example.* As a third example, we show a more complex attack, involving different tools and file downloads. The attack started with a single HTTP GET query using the following (sanitized) request:

```
/phpBB/includes/functions.php?phpbb\_root\_path=\%20\
%22powered\%20byhttp://XXX/j0.gif?\&add=bot
```

This denotes an attack against the PHP bulletin board: an attempted file inclusion attack targeting the variable `phpbb_root_path` in the file *functions.php*. The attack tried to include the file *j0.gif* from a remote location. Again, we retrieved a copy of this file automatically via HIHAT. This file turned out to be a PHP-based shell utility. The utility supports all basic operations like file listing, changing of permissions, command execution, and file upload. Moreover, it includes a mechanism for *self-propagation*. This mechanism is activated once the shell is executed the first time. At this time, it tries to download and execute a second file, named *spread.txt*. The second file has only one purpose: it attempts to fetch and execute a copy of a third file called *fast.txt*. It uses fifteen different commands, download locations, and options to get a copy of *fast.txt*. Once the third file has been downloaded successfully, it is executed.

This file contains a so called *IRC bot*. An IRC bot is a program that connects to an Internet Relay Chat (IRC) server and typically allows to automate some of the IRC functions. Bots usually denote programs which allow an attacker to remotely control and utilize vulnerable machines once they have been infected successfully. Often different bots are combined and connected to powerful networks, so called *botnets*. More information about IRC bots and botnets can be found in the literature [4,19].

The IRC bot we found provides various typical IRC functions. Apart from this, it also includes the option to access the system it is running on and execute arbitrary shell-commands at the host. It also has the ability to perform Denial-of-Service attacks against other machines. Moreover, it can scan for other vulnerable machines with the help of a query to a search engine, building a hitlist of new targets. This allows the bot to be used for different attacks and various targets. The results of the search query are checked for validity. On success, an exploit is performed and a payload is transmitted to the new victim. In general, the script supports different payloads. In our example it was using a copy of itself for transmission.

## 6 Conclusion and Future Work

In conclusion, we have presented the design and implementation of a generic toolkit for turning arbitrary PHP web applications into high interaction honeypots. We have demonstrated its wide applicability by applying it to four popular existing applications, PHPMyAdmin, PHPNuke, phphBB and PHPShell. We have described a method for drawing attackers to our honeypot with transparent links. Finally we have demonstrated the effectiveness of our system by using it analyze 70 actual attacks which included 9 complete malware tools that were captured when downloaded by attackers. This is a powerful result that indicates that any PHP application can be turned into an effective high-interaction honeypot in a simple and automated fashion.

In our case, we used this tool to deploy PHP applications with known vulnerabilities. This allowed us to study how often in what ways already known vulnerabilities are being exploited. We could instead have deployed the newest and most patched versions of these applications shifting the emphasis to monitoring for new exploits discovered in the latest software. Applying our logging code is so simple and un-intrusive to apply that original application developers could consider use of this tool as a phase in testing their software.

Although the data analyzed in this paper is December 15 2006 through February 28 2007, we continue to run the system - collecting more attack data and more downloaded malware tools. The emphasis in this paper was on the design and validation of this approach, but we are also interested in the results obtained with the tool - patterns of attacks over time, characterization of the type of attack tools downloaded,etc.

One keys area of future work is to extend support to other programming languages popular for web development such as Javascript and Perl. Another is to make the limits on outgoing web traffic more dynamic. In order to protect other systems, we currently place relatively tight limits on the amount of outgoing web traffic an attacker can generate from our honeypot. However this can cut off the process of the attack before sufficient data is collected to completely analyze and understand it.

We would like the administrator of the honeypot to be able to write triggers to match attack patterns they have seen before and for which they want to allow incrementally more access in order to learn about the next stage of the attack vector.

## References

1. Edward Balas and Camilo Viecco. Towards a Third Generation Data Capture Architecture for Honeynets. In *Proceeedings of the 6th IEEE Information Assurance Workshop*, West Point, 2005. IEEE.
2. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 107–117, 1998.
3. David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian B. Grizzard, John G. Levine, and Henry L. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of 7th International Symposium (RAID'04)*, pages 39–58, 2004.
4. Felix C. Freiling, Thorsten Holz, and Georg Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of 10th European Symposium on Research in Computer Security (ESORICS'05)*, pages 319–335, 2005.
5. Martin Geisler. PHPShell webpage. `http://mgeisler.net/php-shell/`. Last checked: 04/2007.
6. "Google Hack Honeypot" project. `http://ghh.sourceforge.net/`. Last checked: 04/2007.
7. Google Hacking Database. `http://johnny.ihackstuff.com/ghdb.php`. Last checked: 04/2007.
8. NCSA Software Development Group. Common Gateway Interface (CGI) specification of the environment variables. `http://hoohoo.ncsa.uiuc.edu/cgi/env.html`.
9. Thorsten Holz, Jan Göbel, and Jens Hektor. Advanced honeypot-based intrusion detection. *;login:*, 31(6), 2007.
10. Christopher Kruegel, Giovanni Vigna, and William Robertson. A multi-model approach to the detection of web-based attacks. *Comput. Networks*, 48(5):717–738, 2005.
11. Bill McCarty. Automated Identity Theft. *IEEE Security & Privacy*, 1(5):89–92, 2003.
12. Laurent Oudot. PHP.Hop – PHP Honeypot Project. `http://www.rstack.org/phphop/`, February 2006.
13. phpBB - creating communities. `http://www.phpbb.com/`. Last checked: 04/2007.
14. The PHPMyadmin project - effective MySQL management. `http://www.phpmyadmin.net/`. Last checked: 04/2007.
15. Content management system PHPNuke. `http://www.phpnuke.org/`. Last checked: 04/2007.
16. The Honeynet Project. *Know Your Enemy: Learning About Security Threats*. Addison-Wesley Longman, 2nd edition, May 2004.
17. Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

18. Niels Provos, Joe McClain, and Ke Wang. Search worms. In *WORM '06: Proceedings of the 4th ACM Workshop on Recurring Malcode*, pages 1–8, New York, NY, USA, 2006. ACM Press.
19. Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 41–52, 2006.
20. Jamie Riden, Ryan McGeehan, Brian Engert, and Michael Mueter. The Honeynet Project: Know Your Enemy - Web Application Threats, March 2007. `http://www.honeynet.org/papers/webapp/`.
21. Secunia. Advisory SA17543: PHP-Nuke query SQL Injection Vulnerability. `http://secunia.com/advisories/17543/`, November 2005.
22. Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 33–42, New York, NY, USA, 2004. ACM Press.
23. Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
24. Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP '05: Proceedings of the 20th ACM symposium on Operating systems principles*, pages 148–162, New York, NY, USA, 2005. ACM Press.
25. Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 11–18, New York, NY, USA, 2003. ACM Press.

## A Transparent Link Types

This section describes twelve transparent links, all of which comply to at least one of the evaluated requirements: each of these links is invisible to the user up to some degree. Some types of links are not noticeable at all when the page is displayed in a web browser, whereas others may be recognized at the screen up to some degree, for instance as a single pixel or little dot at the website. Of course the visibility may also depend on the web browser a user choses to look at the page. However, at all times the link can be identified in the source code of a website and this denotes the point which ensures that web spiders always have the chance to detect and crawl the link.

The following transparent links have been considered:

1. A link with image size one:

```
<a href="HONEYPOT_URL">
<img src="link.gif" height="1" width="1" border="0">
</a>
```

2. An exterior div element with offscreen-absolute-positions including a hyperlink:

```
<div style="position: absolute; top: -500px; left: -400px;">
<a href="HONEYPOT_URL">crawl here</a>
</div>
```

3. Display:none in an exterior div element containing a hyperlink:
   ```
   <div style="display: none;">
   <a href="HONEYPOT_URL">link here</a>
   </div>
   ```
4. An anchor containing style setting display-none:
   ```
   <a href="HONEYPOT_URL" style="display:none;">crawl here</a>
   ```
5. A hyperlink only containing a comment:
   ```
   <a href="HONEYPOT_URL"><!-- crawl here--></a>
   ```
6. A hyperlink hidden in a comment:
   ```
   <!-- <a href="HONEYPOT_URL">crawl here</a> -->
   ```
7. A blank hyperlink:
   ```
   <a href="HONEYPOT_URL"></a>
   ```
8. A hyperlink including text-formatting:
   ```
   <a href="HONEYPOT_URL">
   <font face="Verdana"></font>
   </a>
   ```
9. A hyperlink with style settings, comprising only a non-breaking space:
   ```
   <a style="cursor:text;text-decoration:none;" href="HONEYPOT_URL">
   &nbsp;
   </a>
   ```
10. A hyperlink with interior, non-displaying span element:
    ```
    <a href="HONEYPOT_URL">
    <span style="display: none;">crawl here</span>
    </a>
    ```
11. A hyperlink containing zero size div element:
    ```
    <a href="HONEYPOT_URL"><div style="height: 0px; width: 0px;">
    </div>
    </a>
    ```
12. An image map of link size zero:
    ```
    <img src="crawler.gif" border="0" usemap="#ImageMap">
    <map name="ImageMap">
    <area shape="rect" coords="0,0,10,0" href="HONEYPOT_URL">
    </map>
    ```

This list covers the main types of transparent links, but many small variations exist. An example would be a link with an image where the width and height is set to zero, as a variant for link number 1. Every little variant can have a different degree of transparency, but may also cause a different result in indexing and ranking through the web spiders.