Weems: An extensible HTTP honeypot

Deon Pearson*,Barry Irwin*,†, Alan Herbert
*Department of Computer Science, Rhodes University, P.O. Box 94, Grahamstown 6140, South Africa
[1]g13p1006@campus.ru.ac.za [2]b.irwin@ru.ac.za [3]a.herbert@ru.ac.za
†Council for Scientific and Industrial Research, Pretoria, South Africa.
[2]b.irwin@ru.ac.za

***Abstract*—Malicious entities are constantly trying their luck at exploiting known vulnerabilities in web services, in an attempt to gain access to resources unauthorized access to resources. For this reason security specialists deploy various network defenses with the goal preventing these threats; one such tool used are web based honeypots. Historically a honeypot will be deployed facing the Internet to masquerade as a live system with the intention of attracting attackers away from the valuable data. Researchers adapted these honeypots and turned them into a platform to allow for the studying and understanding of web attacks and threats on the Internet. Having the ability to develop a honeypot to replicate a specific service meant researchers can now study the behavior patterns of threats, thus giving a better understanding of how to defend against them. This paper discusses a high-level design and implementation of Weems, a low-interaction web based modular HTTP honeypot system. It also presents results obtained from various deployments over a period of time and what can be interpreted from these results.**

***Index Terms*—Internet security, Network security, HTTP Honeypots**

I. Introduction

Honeypots serve a valuable role in understanding and protecting against cyber-attacks. According to [1] a honeypot can be defined as "...a security resource whose value lies in being probed, attacked or compromised." Honeypots can serve several purposes, they can lure an attacker away from valuable network resources, warn network administrators of possible exploitation or provide valuable data for an in-depth examination of attack methods [2]. This philosophy of luring attackers to a web server designed to be hacked and attacked, has evolved into what we now refer to as the "honeypot" [3].

Honeypots can take on one of two architectural designs, low-interaction or high-interaction. A low-interaction honeypot is one which replicates a specific service on a server and nothing more. A high-interaction honeypot however, replicates the full server and all the processes on it. Weems is built as a low-interaction web based HTTP honeypot. Its goal is to provide a tool for the research into known and unknown threats from the Internet. To achieve such a system, four key features were addressed in the design:

1) Weems must be capable of responding to all HTTP GET and POST requests from any web client.
2) Each request received by the system must be logged and recorded with as much information on the request as possible.
3) Weems must be designed to be deployable on multiple IP address, in multiple locations at the same time, independent of each other.
4) Weems must be designed as a modular system to provide extensibility during its deployment. This extensibility will allow for new modules to be developed and added to each instance of Weems. Allowing for furture tailoring to specific data collection needs.Furthermore developing a honeypot with the default ability to respond to all requests without giving any error messages

Weems aims to provide a variation to the traditional honeypots discussed in Section II. By developing a modular system which is easily extensible, Weems provides the ability to adapt to new threats or new requirmments without the need for a complete redesigning from the ground up. Furthemore, having a fully adaptable, customizable and open source system allows each indivdual to taylor Weems to their specific needs.

The remainder of this paper is laid out as follows.

- Section II discusses existing client and server side honeypot systems used in the research and analysis field.
- Section III presents a short explanation a server-side honeypot and how it is designed.
- Section IV presents the design of Weems along with the various components and sub-components which were developed into the system.
- Section V summarizes the live testing environment used for the duration of the testing phase.
- Section VI discusses the analysis and conclusions which can be drawn form the log entries received during the testing and deployment phase.
- Section VII concludes the paper bring the work to a close with conclusions and future work.

II. Related Work

Honeypots are not a new concept; previous research and development of honeypots have produced many types and varieties of honeypots. The honeypots discussed in this section are unique in the way they operate and the threats they target. These honeypots are significant to this project because they all are web based HTTP honeypots focusing on the malicious client or server entity on the Internet.

A. HoneyC

HoneyC is designed to emulate a web client issuing a request to a web server [4]. The goal of HoneyC is catch web server that serve malicious content back to the client. To achieve this HoneyC maintains a list of URLs which it plans to investigate. For each URL the system will send a request and wait for the response before analyzing it to determine if there is any malicious code being sent back.

B. Glastopf

Glastopf was developed to emulate a set of known vulnerabilities currently being exploited on the web [5]. Glastopf employs a distributed architecture with honeypots located in various parts of the world. All the data recorded by these honeypots such as the log files are stored in a central MySQL database. Glastopf was designed to initially have a small set of predefined attack templates. These templates can be monitored to build pattern matching software to match each URL against. Over time it will compile a list of vulnerabilities by monitoring which URL is being requested by an attacker and adding it to their search index.

C. MWCollectD

The main purpose of MWCollectD is to emulate known vulnerable protocols [6]. The purpose of this emulation is to catch malicious worms attempting to propagate themselves to other systems. These worms will propagate themselves through the execution of shell code on a target system. The goal of MWCollectD is to entice malicious worms to execute code on the system, allowing the shell code to be captured and analyzed. As part of the analysis MWCollectD attempts to execute the code in a controlled environment using a tool called *Libemu* [7], which allows the developers to gain further understanding of how the infection works, while preventing the worm from gaining control of the system.

D. Honeyware

Honeyware attempts to mimic a client browser interacting with a web server. The goal is to determine if the server has any malicious code running on it [8]. Honeyware was developed to overcome two main challenges identified in current low-interaction honeypots, IP tracking and Geo-location. IP tracking is when a malicious server will only send malicious code back to a client after the client makes a number of requests to the server. Geo-location dependent servers implement a feature which presents malicious code to clients browsing from a specific country or continent. To bypass the IP tracking feature on malicious web servers, Honeyware will issue the same request multiple times in an attempt to activate the malicious code. To overcome the Geo-location feature the designers of Honeyware had to first determine the top locations for malicious attacks and deploy the server in these location to bypass Geo-location dependent servers.

III. Honeypots

Honeypots can be desinged to opperate as either client or server side applications. The characteristics of a client-side honeypot according to [9] are:

- Their purpose is to detect malicious servers posing a threat to web clients visiting the web server.
- They provide information client-side attacks present on the Internet.
- They are active honeypots, they go out searching for malicious web servers.

The characteristics of a server-side honeypot according to [9] are:

- It replicates a vulnerable web server. The purpose is to attract malicious clients to interact with the honeypot.
- Server-side honeypots are passive honeypots; they sit on the Internet waiting for an attacker or worm to interact with it.
- The aim behind is to log the client interactions in the hope capturing attempts at exploiting the web application.
- It is developed with specific vulnerabilities designed to lure the attacker to exploit the honeypot. The goal is to convince the attacker they are interacting with a legitimate web server.

When a web client and server communicate with each other they pass HTTP messages between each other. The ability to correctly accept and respond to these messages enable a web server to be turned into a honeypot. An HTTP message contains the request and the payload. The request line is of most importance in a honeypot. This is due to the fact that it gives al the information on what action the attacker is trying to perform and on what resource. The payload proves useful when the attacker is uploading data such as web shells or scripts. The payload can be dumped to a file for furture use.

IV. Design and Implementation

A key design aspect of Weems is its ability to be rapidly deployed in a minimal hardware environment for a period of time. To achieve this, Weems needed to be light-weight and easy to use. For this reason Weems was developed in a Linux environment using Python 2.7.12. To provide the web server capabilities, Flask[1] was chosen for the web framework and Jinja[2] to was chosen for the templating engine.

Weems must be capable of responding to any HTTP request issued by any web client. However, simply responding to a request does little in the way of mimicking a particular service. Therefore, Weems is designed to respond to a request in a specific way based on the type of request being received.

The default response to any request is an `HTTP/200` response. This type of response tells the client that the resource they are requesting does exist on the server. Often the requesting client is sending the request from a none web browser platform probing for a resource. Therefore, a simple response such as `HTTP/200` is enough to signal to a client that the request was successful. The client is not concerned about the actual content of the message, only that the resource exists.

When a malicious client receives an `HTTP/200` response, they are enticed to issue further exploitation requests. Subsequent attempts will all be met with an `HTTP/200` or a more specific response based on the resource being requested. The following points highlight the significance of the subsequent requests received by the system:

- Analyzing these requests will allow for the identification of the types of exploitation and web attacks on the Internet.
- By studying the requests and determining the resources being requested, one is able to develop new or improved modules to provide a more realistic response.

Responding to any URL request with an `HTTP/200` message does little in the way of masquerading as a web server running the full service. It simply signals to the client that the request was successful and the resource exists. To successfully monitor attack methods and identify potential vulnerabilities in services, Weems needs respond with more realistic responses back to the client to make the client-server interaction process closer to the real service.

The overall design of Weems can be conceptualized into three main components. Each component plays a crucial role in the system as a whole. Figure 1 shows the three components

[1]http://flask.pocoo.org/
[2]http://jinja.pocoo.org/

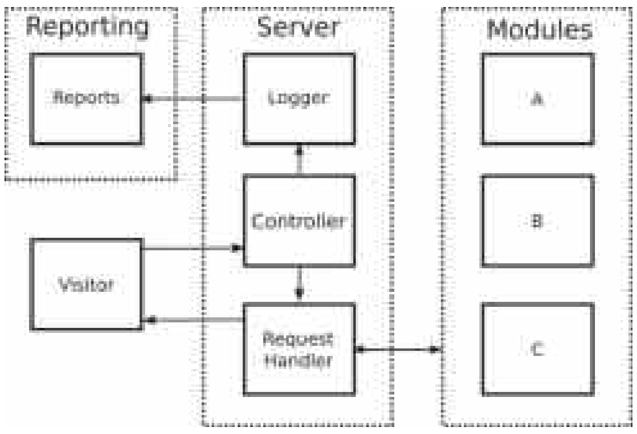

Fig. 1. System Overview

and the communication paths occurring between them, the Server component, the Reporting component and the Module components. The visitor in Figure 1 is the client, interacting with the web server making requests and receiving responses.

The components can be further divided into sub-components. The Server component consists of the *Controller*, *Request Handler* and *Logger* sub-component. It is responsible for handling HTTP requests and sending HTTP responses. The Module component consists of any number of sub-components depending on the vulnerabilities being replicated at the time of running Weems. These modules are made known to the *Request Handler* at runtime allowing for requests to be passed to their specific modules to be processed.

In staying with the modularity and extensibility aspect, each component bar the *Controller* can be redesigned, reconfigured and replaced with another more suitable component. For example, the *Logger* which is discussed in Section IV-E is responsible for logging each request made to Weems. How the logging is done and where the request is logged to is fully dependent on the requirements at the time of running the instance.

For logical explanatory purposes the *Controller* and *Request Handler* are discussed and represented separately. However, in reality the *Controller* and *Request Handler* operate as a single unit, relying on each other to form an operational web server.

A. The Controller

The *Controller* component is the core of Weems, it is responsible for the overall running of a web server instance. When an instance of Weems is executed it is assigned a unique IP address, which is a combination of an IP address and TCP port number for example 127.0.0.1:8080 or 192.192.192.3:80. Any connection and subsequent HTTP request to this Port/IP endpoint will be accepted and processed by the system. On receiving a request, the *Controller* will execute two actions:

1) Call the *Logger* to create a log entry for the request received.
2) Pass the full request to the *Request Handler*. When the *Controller* has passed the request to the *Request Handler* its role in the system is complete.

B. The Request Handler

The *Request Handler* receives the request from the Controller. When a request is received, the *Request Handler* must process it accordingly. At this point the *Request Handler* must decide which module the request will be passed to. One of two choice will be made by. The *Request Handler* can call either a specific module or the default module (see Section IV-D). If there is no module for the requested path, the handler invokes the default handler function to create a generic `HTTP /200` response. If however, there is a module present for the requested URL or path, the *Request Handler* will call the relevant module to handle the request further.

Each module is responsible for how to handle the request and the type of response to send back to the client. The *Request Handler* simply decides who should handle the request.

C. Response Handler

The Response Handler is a logical component of the Flask web framework. It forms part of the *Request Handler*. Its role is to send a suitable response to the web client. The Response Handler is included in the Flask web framework; the list below according to [10], shows the four steps Flask uses when determining how to handle response.

- If a response is of the correct type such as `text/html`, it is returned to the client.
- If the response is a string (e.g. `return ('Flask response')`), Flask must create a response object with the data provided and the default parameters.
- Flask allows tuples to be returned in the form `( response, status, headers)`. If a tuple is returned, there must be at least one item in the tuple.
  - Response: is the actual response to be sent
  - Status: is the HTTP response code which will override the default response code sent by Flask.
  - Headers: are the HTTP header fields. These can be in the form of a list or dictionary.
- The default behavior will be for Flask to treat the response as a Web Server Gateway Interface (WSGI) and convert it into a response object. WSGI is a simple interface between web servers and web applications or frameworks. The goal of WSGI is to provide a universal interface capable of supporting interactions between the web server and the web framework [11].

D. Default Handler

If the *Request Handler* cannot find a module for a requested path Weems must still be able to accept the request and send back some response. The requesting client must still receive a response signaling the request was successful. The Default Handler is responsible for such cases where there is no implemented module which can handle the request. The Default Handler must first determine if the request method is a GET or POST method. If the request is a POST request, the Default Handler will attempt to save any extra data in the request.

E. The Default Logger

When the *Controller* calls the Logger, it does not know how the *Logger* creates the logs or what logs are created. The *Controller* simply makes the call to the Logger; the *Controller* does not know what the *Logger* does or how it logs the requests. Although a logging module is always required by Weems, it is up to the user as to what this logging module does. When the *Controller* calls the *Logger* it does not require any response, it simply passes the full request to the *Logger* and continues with its operation.

For this project the *Logger* has been implemented with three types of logs.

1) **Access Logs:** This project has chosen to use the apache Common Log Format (CLF) for the access logs, because it is a widely adopted logging format amongst web servers. As such, there is a widely available set of analysis tools and techniques available to use when analyzing the requests logged to Weems.
2) **Request Logs:** The request logs serve the purpose of providing additional information on the request being received. The access logs simply log the request URI and requesting client, the Request logs provide an ability to determine what type of client was making the request, ie, a web browser, a bot or a web crawler.
3) **POST Logs:** The POST logs serve the purpose of dumping the full HTTP POST message to a text file for future analysis. If the logging handler determines the method is a POST request, it will dump the entire contents of the request message to a file. The *Logger* does this to provide as much information on POST requests as possible.

F. The Reporter

The *Reporter* is responsible for providing information back to the administrator of Weems. The *Reporter* is a web accessible list of available access logs. Access to these reports are limited to a specific clients browsing from a specific IP address and key combination. This module is considered an administration module and is therefore only accessible by administrators of Weems. The following all have to be true to allow the client to access the reporting page of Weems:

- The client must be accessing the instance from a predefined administration IP address.
- The client must make a request to a URL, which has been determined in the configuration of the module. This can be any value set by the developer.

G. The Module Controller

The Module component is composed of any number of modules for specific vulnerabilities. If a vulnerable URL or path is being monitored it is likely there will be a module implemented; in such a situation the module takes responsibility for processing the request. Each module must be registered with the module controller before the *Request Handler* knows about the module.

This project implemented four example modules targeting vulnerabilities and one additional module for uploading files. The vulnerability modules implemented were the WordPress Administration, WordPress Login, SQL Injection and Remote File Inclusion modules. The additional module was the File Upload module. The remaining sections will briefly discuss each module.

H. SQL Injection

In code injection attacks, attackers are attempting to inject their own malicious code into an application's database. The goal of a code injection attack is to force the web server to execute the code on the local server with elevated privileges used by the vulnerable application. Accomplishing this will open the door to further exploitations on the server [12].

When code is injected into an application it is generally done so in a string format with the attempt to get the application to read the string as executable code. The most popular code injection attack is SQL injection (SQLi). This attack happens when an attacker constructs a string containing SQL commands which is then entered into a search box or login form. When the injected code is retrieved from the database in the future it will run as SQL code and force the database to perform unintended actions [13].

I. WordPress

A WordPress server has many known vulnerabilities [14]. For this project two vulnerabilities or potential vulnerabilities were identified and modules were implemented to replicate them. The first module was aimed at capturing exploitation attempts in the "`/wp-admin`" resource and the second module was intended to capture authentication bypass attempts in the "`/wp-login.php`" resource.

1) WordPress Admin: The wp-admin handler serves the purpose of sending a WordPress administration page back to the client if a request is made for `/wp-admin`. The purpose of this is to emulate a successful attempt to open the "`/wp-admin`" page and to entice the attacker to attempt further interactions with the "`/wp-admin`" page. All requests made to this page would be logged in the access logs for record.

2) WordPress Login: When a request is made for the "`/wp-login.php`" resource the Request Handler calls the wp_login handler. The wp_login handler is responsible firstly, for sending the WordPress login page to the client and secondly for recording each username and password combination used to bypass authentication. The purpose of logging the username and password received, for a log in attempt, is to monitor the brute force attempts, or exploitation attempts to gain access to the WordPress administration resource.

J. Remote File Inclusion

A remote file inclusion attack is an attempt by an attacker to force the web server to run malicious code on its web page by retrieving code from a URL located on a remote public server [15].

The Remote File Inclusion (RFI) module is designed to capture any attempts to include files from another URL. If a request is received, containing a URL redirecting to another page for example `127.0.0.1:8080/foo/bar&page=http://example.com` the attacker is attempting to make Weems fetch a resource from `http://www.example.com`.

K. File Upload

The file upload module provides the ability to upload a file to Weems via a simple web page. The purpose of this is to present the client with the perception of being able to upload any file (web shell or script) to the server, which they may use as a spring board for further attacks.

V. Live Deployment

Four geographical locations were chosen to host the virtual servers and instances of Weems. The locations were chosen based on access to virtual server hosting infrastructure and diversities in the countries technological advances. The instances ran for a period of 30 days. Table I shows the four geographical locations and the number of instances at each location, as well as the hosting platform used.

TABLE I
Summary of Instances

Country	Code	Hosting Service	Instances
America	USA	Digital Ocean	1
Singapore	SG	Digital Ocean	1
Germany	DE	Digital Ocean	1
South Africa	ZA	Private Servers	4

Table II summarizes the requests received for each instance. To get a true representation of the number of different requests being made the URLs need to be distinguishable. Identifying unique requests a combination of two parts to be different, the path and the HTTP version. For example, `GET /foo HTTP/1.0` and `GET /foo HTTP/1.1` are considered to be two different requests.

TABLE II
Instance Data

Instance	Unique IP's	Unique URLs	Total Hits
USA	1044	17631	24516
SG	251	123	1946
ZA2	158	57	417
DE	219	77	413
ZA3	143	60	350
ZA1	120	59	329
ZA4	121	39	255

VI. Analysis

An examination of the collected log files revealed a wide range of requests being made to each instance of Weems. This section will highlight some of the notable requests captured by Weems during the deployment stage of the project. Before analyzing the log files, some requests needed to be excluded from the list of possible candidates. This list below shows the criteria used to determine whether to exclude a type of request or not.

1) Requests for "/" where omitted from the logs because these requests topped all the logs and gave and gave no further information about the clients request.
2) Requests "`favicon.ico`" are typically seen when a browser is used to request a web page. This request was seen in conjunction with "/" requests and therefore gives little information as to the intent of the client.
3) Requests originating from web crawlers browsing recycled IP addresses.

After excluding irrelevant requests, the following criteria was used to determine which requests to study further.

1) The total number of requests received containing the URI.
2) Did it form part of a combination of other requests.
3) How out of place did the request path look.
4) Was the requests an obvious exploit.
5) In how many other instances the request appeared.

After applying the selection criteria listed above, Table III summarizes the most frequent requests received common to all the instances.

TABLE III
Most Frequent Requests

Request/Group Name	Instances	No.
Authentication Bypass Exploits	SG, USA, DE, ZA1, ZA2, ZA3, ZA4	7/7
Freaky Ghost	SG, USA, DE, ZA1, ZA2, ZA3, ZA4	7/7
DFind Scanner	SG, USA, DE, ZA1, ZA2, ZA3, ZA4	7/7
i18n Vulnerability	SG, USA, DE, ZA1, ZA2, ZA3, ZA4	7/7
HNAP Vulnerability	SG, DE, ZA1, ZA2, ZA3, ZA4	6/7
CGI Directory Listing	DE, ZA1, ZA2, ZA3, ZA4	5/7
phpMyAdmin	DE, ZA1, ZA2, ZA3, ZA4	5/7
IP Camera Vulnerability	ZA2, ZA3, ZA4	3/7

A. Authentication Bypass Exploits

A common sequence of requests being made to Weems can be seen in Listing 1. The requests were consecutive requests originating from the same IP address. The attack consists of four sequential requests within a second of each other.

```
1  [21/Sep/2016:20:56:53 +0152] ``GET /cgi/
       common.cgi HTTP/1.0''
2  [21/Sep/2016:20:56:53 +0152] ``GET /stssys.
       htm HTTP/1.0''
3  [21/Sep/2016:20:56:54 +0152] ``GET / HTTP
       /1.0''
4  [21/Sep/2016:20:56:54 +0152] ``POST /command
       .php HTTP/1.0''
```

Listing 1. IP Camera Authentication Bypass Exploit

The overall conclusion about the series of attacks is that the attacker is probing for vulnerabilities in network devices such as IP cameras, routers and network printers. The probe is searching for devices which do not require authentication to access certain parts of the web interface [16]. The following points discuss each line in Listing 1.

- Line 1 according to [16] is an attempt to retrieve the `common.cgi` file from an IP camera. The file contains information such as the public IP address, MAC address, name, version number and gateway address of the IP camera [16]. With this information the attacker may return to attempt further attacks on the device.
- Line 2 shows an attempt to exploit a vulnerability in TRENDnet Print Servers where the attacker is able to reset the print server remotely to factory details. If the

request returns the expected result the attacker will return to attempt to reset the printer to factory settings in an attempt to gain sensitive information [17].

- Line 4 is an attempt to exploit a hole in D-Link Wireless Routers by means of command injection into the web administration page [18]. Listing 2 shows the body of the POST request. The probe is simply attempting to create a text file in the `/var/tmp` folder and printing the file back to the web page. If the command is successful this signals the Router is vulnerable to injection attacks [18].

```
cd /var/tmp && echo -ne \\x3610cker > 610
    cker.txt && cat 610cker.txt
```

Listing 2. Command POST

B. CGI Directory Listing

CGI (Common Gateway Interface) is the standard for how a web server and client communicate via the Internet. It defines the process of a server passing a web request to an application, receiving the response from the application and returning the data back to the client [19]. The query string is used to issue the command of what to print out. Listing 3 shows example request lines for a CGI bin exploitation. Line 1 shows the server side code which is executed when a request is received for `/cgi-bin/test-cgi`. Lines 2 and 3 shows two variants of how an attacker can give the command to list the contents of any directory. In this example it is the root directory. The first method used is through the web browser and the second example is through a telnet command [19].

```
1  echo QUERY_STRING = $QUERY_STRING
2  GET /cgi-bin/test-cgi/*
3  GET /cgi-bin/test-cgi?x> /*
```

Listing 3. Example CGI Bin Exploit Process (Adapted from [19])

VII. Conclusion

Due to the ever increasing threat from malicious web clients, there is the need to understand the threat patterns and behaviors. Developing a honeypot which addresses each and every threat is not possible. However, developing a platform from which threat specific modules can be built and integrated is more realistic. Weems was designed as a modular, low-interaction web based HTTP honeypot which can be run in a low resource environment for a short to long period of time. The advantages of building a modular system are extensibility, scalability and adaptability, all of which are core features of Weems. The modular design of Weems meant that each component can be redesigned with richer functionality to suit the needs at time.

During the testing phase of the project it was concluded that Weems can indeed be deployed in multiple locations independent of the each other. The success of the project was further confirmed after analysis on the logs revealed a substantial amount of probing attempts by bots searching for vulnerable services on Weems. Although the were many requests to Weems for known vulnerabilities, there were no recorded exploitation attempts to any of the services implemented. Despite this, the results show that Weems can be used as a simple request logging web server as well as a honeypot masquerading multiple web services.

Acknowledgment

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs/CORIANT, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 90243). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author and that none of the above mentioned sponsors accept liability whatsoever in this regard.

References

[1] A. Christoforou, H. Gjermundrød, and I. Dionysiou, "Honeycy: A configurable unified management framework for open-source honeypot services," in *Proceedings of the 19th Panhellenic Conference on Informatics*, ser. PCI 15. New York, NY, USA: ACM, 2015, pp. 161–164.

[2] I. Mokube and M. Adams, "Honeypots: Concepts, approaches, and challenges," in *Proceedings of the 45th Annual Southeast Regional Conference*, ser. ACM-SE 45. New York, NY, USA: ACM, 2007, pp. 321–326.

[3] L. Spitzner, *Honeypots: tracking hackers*, A. Wesley, Ed. Addison-Wesley Reading, September 2002, vol. 1. [Online]. Available: http://www.it-docs.net/ddata/792.pdf

[4] C. Seifert, I. Welch, and P. Komisarczuk, "HoneyC The low-interaction client honeypot," *Proceedings of the 2007 New Zealand Computer Science Research Student Conference, Waikato University, Hamilton, New Zealand*, August 2007.

[5] L. Rist, S. Vetsch, M. Kobin, and M. Mauer. (2010, November) A dynamic, low-interaction web application honeypot. Online. Glastopf. [Accessed: 19 April 2016]. [Online]. Available: http://honeynet.org/papers/KYT_glastopf

[6] G. Wicherski, "Placing a low-interaction honeypot in-the-wild: A review of mwcollectd," *Network Security*, vol. 2010, no. 3, pp. 7 – 8, 2010.

[7] A. M. Shukor. (2012, May) Libemu. Online. [Accessed: 27 April 2016]. [Online]. Available: https://launchpad.net/libemu

[8] Y. Alosefer and O. Rana, "Honeyware: a web-based low interaction client honeypot," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, vol. 3. IEEE, 2010, pp. 410–417.

[9] J. Biswas, "Analysis of client honeypots," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 4, 2014.

[10] A. Ronacher. (2015) Flask. Online. [Accessed: 5 September 2016]. [Online]. Available: http://flask.pocoo.org/docs/0.11/

[11] J. Gardner, "The Web Server Gateway Interface (WSGI)," *The Definitive Guide to Pylons*, pp. 369–388, 2009.

[12] L. Coppolino, S. DAntonio, G. Mazzeo, and L. Romano, "Cloud security: Emerging threats and current solutions," *Computers & Electrical Engineering*, pp. 1–15, March 2016.

[13] D. Dede. (2014, November) Most common attacks affecting todays websites. Online. [Accessed: 25 February 2016]. [Online]. Available: https://blog.sucuri.net/2014/11/most-common-attacks-affecting-todays-websites.html

[14] A. Welss. (2012, April) Top 5 wordpress vulnerabilities and how to fix them. Online. eSecurity PLanet. [Accessed: 13 September 2016]. [Online]. Available: http://www.esecurityplanet.com/open-source-security/top-5-wordpress-vulnerabilities-and-how-to-fix-them.html

[15] O. Katz, "Detecting remote file inclusion attacks," Breach Security, Tech. Rep., May 2009.

[16] W. Campbell, "Security of internet protocol cameras - a case example," in *Australian Digital Forensics Conference*, December 2013, pp. 20–25. [Online]. Available: http://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1115&context=adf

[17] A. Sanadi, "TRENDnet Print Server Authentication Bypass Vulnerability," SecPod, Tech. Rep., 2013.

[18] Neon Prime Time. (2016, August) command.php wget HTTP Post. Online Blog. [Accessed: 14 October 2016]. [Online]. Available: https://neonprimetime.blogspot.co.za/2016/08/commandphp-wget-http-post.html

[19] Mudge. (1996) test-cgi vulnerability. Online. [Accessed: 16 Ovtober 2016]. [Online]. Available: http://insecure.org/sploits/test-cgi.html

Deon Pearson Completed Honours in 2016 and is currently studying towards a Masters in Computer Science at Rhodes University. His research is supervised under Prof. Barry Irwin in the Security and Networks Research Group.