Deception in Web Application Honeypots: Case of Glastopf

Banyatsang Mphago
Dep of CS/IS
BIUST
Email: mphagob@biust.ac.bw

Dimane Mpoeleng
Dep of CS/IS
BIUST
Email: mpoelengd@biust.ac.bw

Shedden Masupe
Senior Member of IEEE
Botswana Institute of Technology Research and Innovation
smasupe@bitri.co.bw

Abstract

Honeypots are special tools designed to help track and understand attacker?s motives and their attack methods. In web applications, several honeypots have been developed and some have since been abandoned by their developers. But as honeypots are deployed more and more within computer networks, malicious attackers also devise techniques to detect and circumvent these security tools and thereby exposing limitations in most web application honeypots. Dynamic honeypots however, are believed to be the future of honeypots due to their abilities to adjust to the changing environments. Glastpof is one of the more popular if not the most, dynamic web application honeypot currently released to the public. But Glastopf has its limitations too. Once deployed, Glastopf can be easily identified by the attackers due to its performance and appearance, and as such become less useful to the security community. This research describes some of the limitations inherent in Glastopf, and then proposes possible ways to make it more deceptive and more attractive to attackers.

Keywords: *Honeypot, Glastopf, Deception, WSGI, Attack Surface, Web Application, Camouflage*

1 Introduction

Ever since the inception of computers, information security has been a major concern for most organizations. Web application attacks in particular, represent the biggest threats in the organization's security [1–3]. Verizon [2], reported that in 2016 web application attacks are the main source of data breaches, ranked number 1 with an estimate of over 40 percent of all the data breaches. WhiteHack Security [6], also reported that web application attacks represent over 40 percent of all the total data breaches in 2015. Imperva reported that in 2015 content management system (CMS) applications were attacked three (3) times more than non-CMS applications, and Wordpress CMS applications were attacked seven (7) times more than any other CMS applications.

However, web application honeypots are special tools designed to help and understand this kind of attacks. Lance Spitzner [4], argued that dynamic and intelligent honeypots are the future solution to the biggest challenge facing our modern security technologies in configuration, where once the honeypot is deployed it adapts to the changing environment. Budiarto et-al and Abraham et-all [5, 6] also agreed with Spitzner that with a dynamic honeypot, one need not to worry about going back to update the configuration of the honeypot but rather the honeypot adjust itself to the changing environment. Glastopf is one such example of a dynamic low interaction web application honeypot, and the most popular honeypot for its ability to imitate thousands of vulnerabilities. However, the designers of Glastopf honeypot (The Honeynet Project) was not directed much at deception to defeat attackers in the tactical sense, but rather, the dedication was more at learning about the tool, the tactics, motives and sharing lessons learnt [7].

This paper seeks to analyze and propose the methods and techniques that can make dynamic web application honeypots, and in particular Glastopf, more deceptive and still remain attractive to the hacker community. A honeypot that can?t hide its true identity is useless as attackers will simply avoid it, and also a honeypot that is of little interest to the attackers is also of little value to the security community.

2 Honeypot Concepts

The definition of a honeypot is not a straightforward one, mainly because a honeypot is a general technology and not a solution and do not solve a specific problem. One of the commonly used definition for honeypots is: any security resource that derives its usefulness from being investigated, attacked and compromised by the attackers [8].

There are several ways at which honeypots can be classified, and some of the more popular classifications are based on purpose and level of interaction with the attackers [8–12]. Based on purpose, a honeypot can either be a research honeypot or production honeypot. Research honeypots are those that do not add direct value to the organization but are used by researchers to gain valuable knowledge about the attackers [11, 13]. The primary goal in this type of honeypots is to learn the methods, techniques, and processes the attackers use and pass that knowledge to the security administrators. Production honeypots on the other hand are honeypots deployed in an organization with the primary objective of alerting security administrators of any potential attacks in real time. Based on level of interaction, a honeypot can either be low, medium, or high interaction honeypot. Low interaction honeypot is the one that imitates only services that can be exploited where the honeypot does not provide an operating system for an attacker to interact with, but rather, only simulates services of a particular system [11]. Medium interaction honeypot, just like low interaction honeypot, does not provide an operating system for an attacker to interact with, but rather, the simulated services are more complex technically than the low interaction honeypot. Medium interaction honeypots imitates a collection of systems rather just one in order to present a more convincing interaction with the attacker while still hiding the operating system from the attacker [9, 10]. A high interaction honeypot is the one that gives the attacker to interact with the actual operating system along with real instances of programs rather than their imitations [9, 10].

3 Related Work

This section explores how different honeypot implementations were detected, and how some of these implementations were camouflaged to remain deceptive to the hacker communities.

3.1 Detecting UML Based Honeypots

Detecting honeypots is mostly dependent on the design of the honeypot itself rather than following a predefined criterion out there. Several researchers have used different techniques in detecting honeypots, and all this detections were dependent on how the honeypot is designed and the tools used to design these honeypots.

Innes and Vanapalli [14] discovered that honeypots that use User Mode Linux as their working environment more easily identifiable by the attackers. This is mainly because UML doesn't store data on hard disks but rather on virtual drives that point to already existing portions of file system. Therefore, UML modules were developed to hide where images are mounted (/dev/ubd*) on the UML system, but the major number identifying the /dev/ubd* devices is not the same as the one used for standard IDE or SCSI systems, and this number is 98(0x63).

Holz and Raynal [15], also discovered that honeypots running on UML can also be identified by looking at the /proc tree where there are a number of anomalies that can alert to the attackers that this isn't a real system. First, the cpuinfo of the UML has a model name listed as UML and the mode being reported is Tracing Thread mode. This is because UML by default executes in tracing thread mode, and this is a clear indication to the attackers that this is not a legitimate system. Holz and Raynal [15], also discovered that another way to detect UML based honeypots is to look at the address space of a process in the maps file. The end of a stack in the host operating system is usually indicated by 0xc0000000, but in the UML based it is indicated by 0xbefff000.

3.1.1 Detecting VMware Based Honeypots

Innes and Vanapalli [14], found that prior to 4.5 version of VMware, the hardware was not configurable and remained at defaults where VMware default values can be identified, and this made VMware based honeypots easily detectable. The other discovery by Innes and Vanapalli [14], on VMware based honeypots is the network MAC address that is bound to the network card. The vendor part of the MAC address (the first three octets) on VMware virtual network interface is always one of the following three (3): 00-05-69-xx-xx-xx, 00-0c-29-xx-xx-xx, and or 00-50-56-xx-xx-xx. Innes and Vanapalli also discovered that VMware has an I/O backdoor that was revealed by an analysis of Agobot that it is used to call backdoor functions.

3.1.2 Detecting chroot and Jails Environment

Securing honeypot binaries are often done by the use of chroot and jail environments. Holz and Raynal [15], discovered that the easiest way to tell whether you are inside a chroot() environment is to run an ls-lia command on the root directory, and look at the inodes of '.' and '..' directories. On a standard system, the inodes of this two directories is always two (2), but if you are in the chroot() environment, the inodes of these directories will be something different.

3.1.3 Timing

Another discovery by Holz and Raynal [15], on detecting honeypots is that honeypots running on sebek can be detected by measuring execution time of the read () of the read system where in a system without sebek, minimal time is around 8225 and a scalar product of 0.776282, and in contrast a system with sebek has minimal time of 29999 and scalar product of 0.009930.

3.1.4 Camouflaging Honeypots

Fu et-al [16], discussed ways at which virtual honeypots such as Honeyd can be camouflaged by designing a honeypot that supports link latency in the order of one microsecond (µs) instead of the original one millisecond default, to avoid timing signature profiles that attackers might profile against the honeypot hence launching timing attacks against it. They achieved this by changing the Honeyd code to make it support the timing resolution of microseconds, and this involved modifying both Honeyd and the event management library (libevent).

4 The Current Status

Glastopf on its default form comes with some preexisting dynamic web templates. These templates however, have some problems that may hinder the honeypot to achieve its intended goals. First, the templates are too basic for experienced attackers to basically see that it is not a legitimate webserver. The templates are basic in the sense that the text in the template is un-organized and does not make sense to whoever is reading the content. This has the potential to be the first alert to the attackers that the server is not intending to relay information to its visitors but just a tool serving another purpose than that of a web server, which can also affect the page rank of the application. For a search engine to retrieve and display the results, the engine must first understand the text on the page [17]. The second problem on the look and feel of the web templates is that they have no working links on them, and there are no images or any graphics on the template. Although the developers of the honeypot suggest that the default templates can be replaced by one's own templates, the server used does not support use of images and other graphic content on the templates and as such any graphics on custom templates would not be rendered by the server on its default form. This inability to have any working hyperlinks or graphics on the templates has a negative impact on the search engine optimization, page rank, and search engine performance of the application. Search engine optimization is the process of improving the visibility of a website or web page in a search engine results. In addition to textual analysis of web content, search engines examine the hyperlinks between the pages to extract information about the quality of the page, and page rank is one such measure [17].

Another weakness with Glastopf in its current form relates to its performance. Although it is difficult to calculate the response time of a server due to several other metrics factored in such as the size of a file transferred, the medium used, cpu capabilities, static or dynamic content, network input/output, etc, the processing time of a sever however, can be measured, and the best way to quantify this component is to measure it directly, bypassing the internet and other factors limiting the quantification of the response time of a server [18]. In our tests, we measured Glastopf directly on the machine it is running on using Autobench and httperf performance measuring tools, and we found that Glastopf can only process less than ten (10) http requests per second. This finding was also reported by ENISA [7]. Figure 1 shows our test results on Glastopf using Autobench performance testing tool. The red line indicates that indeed the application can only processes less than 10 http requests per second. Figure 2 also shows the results of the same test with httperf measuring tool, and it shows that Glastopf can only process about 8.5 http requests per second. These readings however, can be problematic when compared to that of production servers. Titchkosky et-al [19], measured the performance of Apache web server on both static and dynamic technologies, either connecting or not to the database. In their test, they found that a webserver with dynamic web content of a 2 kB file on Apache 5 to have a response rate of about 600 requests per second or more depending on the dynamic platform used, and about 200 http requests per second before full utilization of the CPU on a server that connects to the database. These overheads are increased by a factor 8 on a static payload. Apache, however, is not the fastest production server out there due to its high memory usage. Other production servers such as lighttpd can reach requests rates of up to 3000 or Nginx can even reach request rates of up to 12000. Having such a low http request processing rate for Glastopf compared to other production severs can have two main problems in achieving its goals. First, the low request processing rate has the capability to alert the attackers that this might not be a legitimate webserver. Secondly, a low performing webserver has the capability to be less interesting to the people it is mostly targeting, the hackers, because of its slowness. "A webserver that fails to deliver its content, either in a timely manner or not at all, causes visitors to quickly loose interest [20]."

The problem with Glastopf performance is largely attributed to server it uses. Glastopf use custom made WSGI webserver, which by default is not design to be a server but an interface to other WSGI compatible production servers. In our tests, we used CProfile to profile our entire Glastopf application in order to indentify where the bottleneck might be in the application. Due to the large size of the diagram produce by our CProfile, for the prupose of this paper we only show a small section of that diagram (Figure 3). In this CProfiling, it was evident that the majority of a application's bottleneck was coming from the WSGI server and the application its imports while running as shown by the red and purple components in the diagram.

5 Proposed Framework

The conceptual diagram (Figure 4) represents the overall proposed Glastopf optimization for its deception, while making it more attractive to attackers. The first 3 levels of the diagram reflect the current status of Glastopf as stated in the current status section above, whereas the last 3 levels of the diagram reflect the propositions that can make Glastopf more deceptive, as described in this section below.

To make Glastopf process requests faster like a standard server, we employed Gunicorn wrapped behind an Nginx proxy server. This required re-programming of WSGI so that the WSGI application callable is in the same script with runner file script for Gunicorn to able to run. In Glastopf,

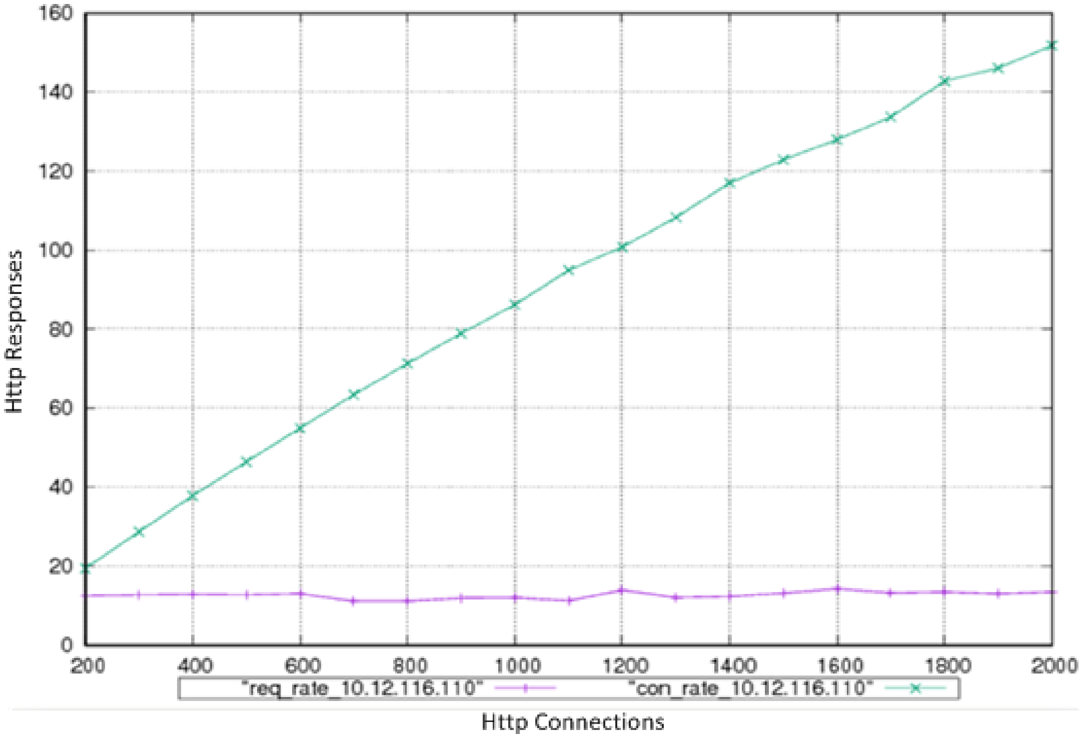
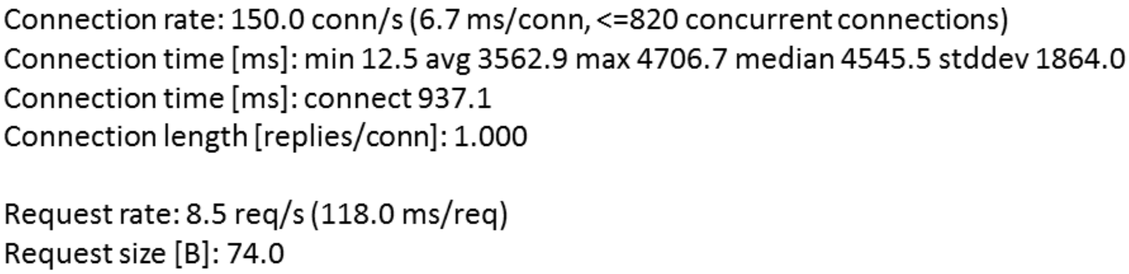


Fig. 1. Glastopf Performance-Autobench



```
Connection rate: 150.0 conn/s (6.7 ms/conn, <=820 concurrent connections)
Connection time [ms]: min 12.5 avg 3562.9 max 4706.7 median 4545.5 stddev 1864.0
Connection time [ms]: connect 937.1
Connection length [replies/conn]: 1.000

Request rate: 8.5 req/s (118.0 ms/req)
Request size [B]: 74.0
```

Fig. 2. Glastopf Performance-httperf

the WSGI application callable is the wsgi_wrapper.py file and runner file is the glastopf-runner.py file. By doing this, we have now moved Glastopf from a development server (WSGI) to a production server (Gunicorn) to gain more performance from our application. Gunicorn is based on a pre-fork worker model where one can specify how many workers to deploy for handling requests. In our tests, based on the fact that our test computer had a quad core processor, we then deployed 9 workers, from the general recommended formula of (2*num-of-core) + 1. Then we used httperf performance tool set at 1000 connection rates to measure our application?s perfomance, and we found that Glastopf now has a request processing rate of 999.1 requests/second. Unlike WSGI server, Nginx has the capabilities to serve static files on a web template. By interfacing Gunicorn and Nginx with WSGI, we have optimized our servers so that serving static files is handled by Nginx and any other contents of the request are forwarded to Gunicorn while WSGI provides the interface to the application.

Although running Glastopf on its default WSGI server in a production environment is not ideal due to performance, the problem of serving static files can still be addressed by creating routes/ router for your WSGI application to serve static files. A router is a script that maps URLs to the codes that handles them, and this directly connects content to what is seen on the webpage. To achieve this, we first created a folder in our application directory and then saved all static files in that folder. Then in our WSGI application callable script, we modified the environment path to point to the static folder created so that whenever a static file is needed, the application can fetch the files using the static route in the static folder.

Another proposition reflected in the conceptual diagram is on the attack surface of Glastopf. An attack surface is any system's resource or any application area that is exposed to attack or can be used to attack an application [21]. The bigger the attack surface is, the more attractive the application is to the attackers. To make Glastopf?s attack larger,

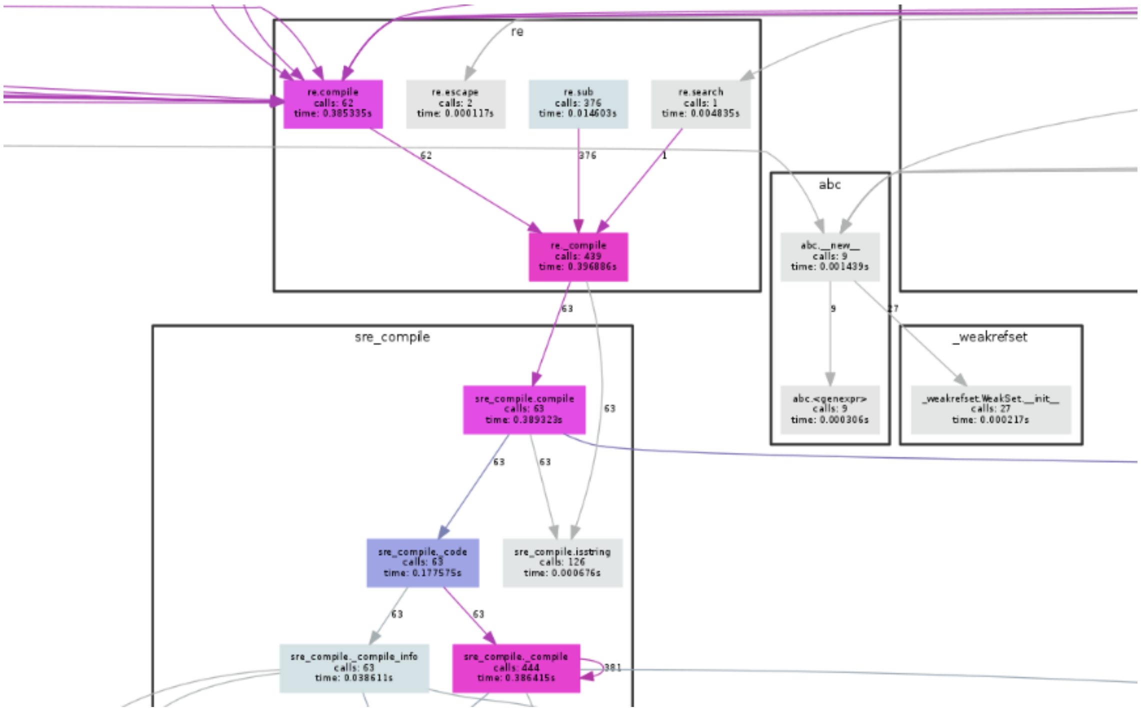


Fig. 3. Glastopf Profile

we determined how attack surface parameters [22, 23] can be deployed in an application in order to make it larger. We achieved this by deploying these attack surface parameters in several formats using different combinations, and then measure the attack surface using Nessus vulnerability scanner. From this test, we found that deploying two same parameters in an application does not increase the size of the attack surface. However, same attack surface parameters with different functionalities would increase the attack surface of an application. Thus, if one attack surface parameter is a subset of another parameter, having both of them in an application would increase the size of the attack surface. Our research revealed that dynamic content in a web application gives the biggest increase in the size of its attack surface than all other attack surface parameters we tested. We also found that if the attack surface parameter does not exist in an application, adding that parameter would automatically increase the attack surface. We created our templates using Wordpress CMS, to bring content based package dependency [?] into play, thereby increasing its attack surface. This however, did not show in our test results due to the fact that current vulnerability scanners do not measure content based package dependencies [24].

Lastly, as reflected in our conceptual framework, we propose improving the PageRank of the application in order to make it more visible to the attackers. PageRank is an algorithm, licensed to Google, for performing links analysis. The algorithm works by promoting, among others, pages with high backlinks [25], as they are seen to be more important than pages with less backlinks. Although this is one of our propositions for optimizing Glastopf, for this paper we were unable to test PageRank because out test bed was on a local machine rather than on a registered domain where PageRank could be tested. Therefore, our next paper would focus on how to make Glastopf rank high on Search Engine Optimization by cheating PageRank, Google Panda and Google Penguin algorithms.

With all the above propositions achieved, a perfect honeypot would have been achieved. A perfect honeypot is the one that performs the same manner or as close as possible to the production webserver for easy camouflage, most vulnerable to be more attractive to the attackers, and with a high PageRank for more visibility.

6 Conclusions and Discussions

We analyzed the deceptive qualities of Glastopf honeypot and found out that Glastopf does not handle http request at the same rate as other production webservers. By using a CProfile profiler we found that the bottleneck in Glastopf is caused by its server: a customized WSGI server that can only process 10 http requests per second. We also found out that another weakness in the deceptive qualities of Glastopf is in its attack surface: thus the interfaces the attacker interact with when attacking the honeypot. Glastopf's attack surface is an HTML page consisting of several dorks, where a dork is a vulnerable path to an application and attackers find this dorks through search engines. However, the HTML page used for the attack surface has some problems too that may limit the honeypot from achieving its intended purpose. We found that the HTML page has no working links and the honeypot cannot render any graphics that might be used in the template. A website that doesn't have any working links nor graphics might send alarms to the attackers that

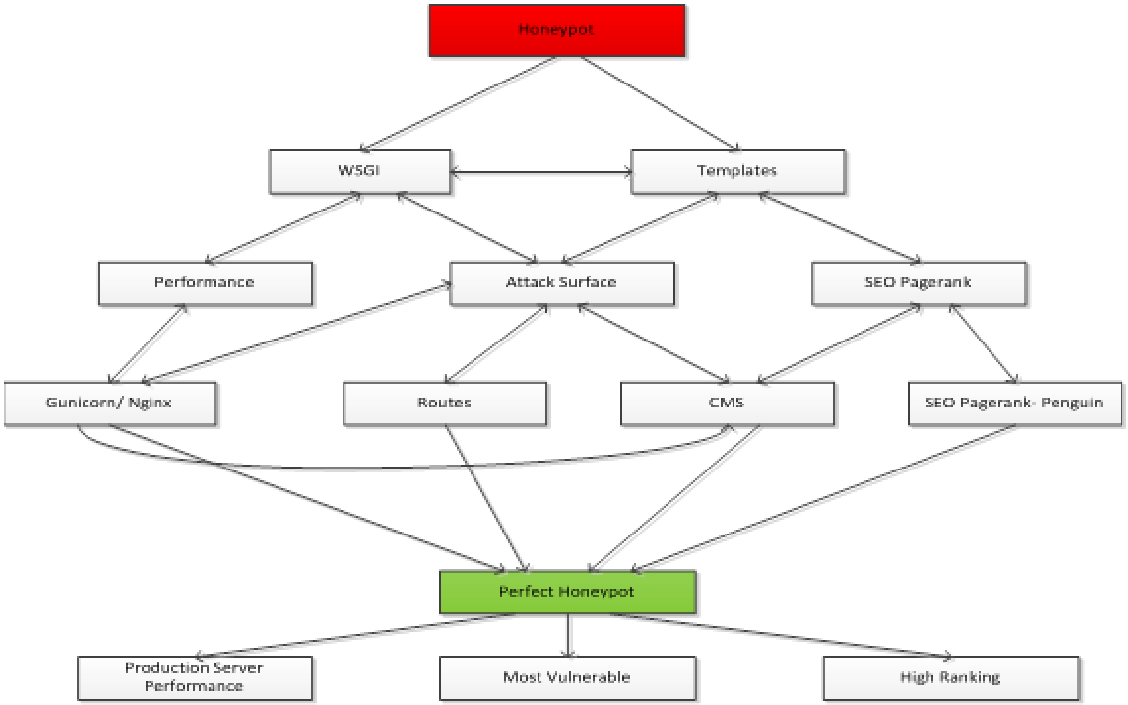


Fig. 4. The Conceptual Diagram

the site might be a honeypot rather than a legitimate website. We found out that the inability of the honeypot to render graphics on web templates was due to its use of WSGI server, which by its designed was built to render dynamic templates and not static files, and therefore cannot render static files such as graphics when not customized. We also found out that the HTML page used for the attack surface has text that doesn't make sense when read. Despite the fact that the honeypot was specifically designed for automated attacks, if it happens that the attacker pulls the interfaces of the site to check what information they have before using the automated tools, he might be able to identify that the website is a honeypot rather than a legitimate website due to the kind of information it has. We therefore proposed a framework that can make the honeypot more deceptive while on the same time making its visibility on the search engines wider. In the conceptual diagram we proposed the use of production web server such as Gunicorn wrapped behind Nginx as a proxy server where the WSGI will act as the interface between the honeypot and the production web server. In this setup, the production server will handle http requests and static files on the web template while the WSGI handles server scripts in the honeypot. From this setup, our honeypot was able to handle about 999.1 http requests per second from the original of about 10 http requests per second, and we can safely say now our honeypot performs like a production server. If production server is used instead of the WSGI, the production server such as Nginx has the capability to render static files, therefore the HTML page can now have graphics and other static files on it so that its attack surface look more like a legitimate webserver. However, if the production server is not used but instead the WSGI is the main server, we proposed proposed creating routes for static files in the WSGI server, and if routes are created, the attack surface will be able to render static files and look more like a legitimate website. We also proposed the use of CMS in creating the attack surface of the honeypot. CMS brings content based package dependency into the honeypot, and content package dependency has the capability to increase the size of the attack surface. We also proposed ways at which attack surface parameters can be used to increase the size of the attack surface of an application. The larger the attack surface of the honeypot, the more visibility it gets when attackers search for vulnerable paths using search engines, and if the honeypot gets more visibility on the search engines, then chances of it being attacked are higher. Lastly on our conceptual diagram we proposed improving the page rank or the honeypot in order to improve its visibility on the search engines. However, due to the requirement of having a public IP to test page rank, in this paper we were unable the page rank of our application.

References

[1] Imperva, "2015 Web Application Attack," Tech. Rep., 2015.

[2] Verizon, "2017 Data Breach Investigations Report Tips on Getting the Most from This Report," Tech. Rep., 2017.

[3] WhiteHat Security, "Web applications security statistics report 2016," 2016.

[4] L. Spitzner, "Dynamic Honeypots," 2003. [Online]. Available: https://www.symantec.com/connect/

articles/dynamic-honeypots

[5] R. Budiarto, A. Samsudin, C. W. Heong, and S. Noori, “Honeypots: why we need a dynamics honeypots?” *Proceedings 2004 International Conference on Information and Communication Technologies From Theory to Applications 2004*, no. May, pp. 565–566, 2004.

[6] R. Prasad, A. Abraham, A. Abhinav, S. V. Gurlahosur, and Y. Srinivasa, “D Esign and Efficient D Eployment of Honeypot and Dynamic Rule Based L Ive,” vol. 3, no. 2, pp. 52–65, 2011.

[7] ENISA, “Proactive Detection of Security Incidents - Honeypots,” p. 181, 2012.

[8] L. Spitzner, *Honeypots: Tracking Hackers*, 2002.

[9] M. Gibbens, “Honeypots,” pp. 1–12, 1999.

[10] R. Baumann and C. Plattner, “White Paper : Honeypots,” 2002.

[11] I. Mokube and M. Adams, “Honeypots : Concepts , Approaches , and Challenges,” pp. 321–326.

[12] G. Wagener, R. State, A. Dulaunoy, and T. Engel, “Self adaptive high interaction honeypots driven by game theory,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5873 LNCS, pp. 741–755, 2009.

[13] M. Shukla and P. Verma, “Honeypot : Concepts , Types and Working,” vol. 3, no. 4, pp. 596–598, 2015.

[14] S. Innes and C. Valli, “Honeypots: How do you know when you are inside one ?” *Australian Digital Forensics Conference*, p. 28, 2006.

[15] T. Holz and F. Raynal, “Detecting honeypots and other suspicious environments,” *Proceedings from the 6th Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC 2005*, vol. 2005, no. June, pp. 29–36, 2005.

[16] X. Fu, B. Graham, D. Cheng, R. Bettati, and W. Zhao, “Camouflaging Virtual Honeypots,” *Work*, pp. 1–17, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.431

[17] D. Gleich, “Models And Algorithms For PageRank Sensitivity,” *PhD Thesis*, 2009.

[18] A. Savoia, “Web Page Response Time,” *Software Testing and Quality Engineering Magazine*, vol. 2001, no. August, pp. 48–53, 2001. [Online]. Available: https://www.stickyminds.com/better-software-magazine/web-page-response-time-101-primer

[19] L. Titchkosky, M. Arlitt, and C. Williamson, “A performance comparison of dynamic Web technologies,” *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[20] D. P. Olshefski, “Measuring and Managing the Remote Client Perceived Response Time for Web Transactions using Server-side Techniques,” 2006.

[21] P. K. Manadhata and J. M. Wing, “An Attack Surface Metric,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2011. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5482589{\%}0Ahttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5482589{\%}0Ahttp://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/2008/CMU-CS-08-152.pdf

[22] S. Goswami, N. R. Krishnan, Mukesh, S. Swarnkar, and P. Mahajan, “Reducing attack surface of a web application by open web application security project compliance,” *Defence Science Journal*, vol. 62, no. 5, pp. 324–330, 2012.

[23] T. Heumann, S. Türpe, and J. Keller., “Quantifying the Attack Surface of a Web Application,” *Sicherheit*, pp. 305–316, 2010.

[24] S. Zahir, J. Pak, J. Singh, J. Pawlick, and Q. Zhu, “Protection and Deception: Discovering Game Theory and Cyber Literacy through a Novel Board Game Experience,” pp. 1–8, 2015. [Online]. Available: http://arxiv.org/abs/1505.05570

[25] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” *World Wide Web Internet And Web Information Systems*, vol. 54, no. 1999-66, pp. 1–17, 1998.