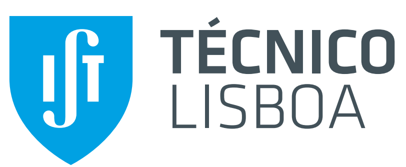


MockingPot: Generate and Integrate Honeypots Into Existing Web Applications

Pedro Alexandre Paixão Alves

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Eng. Diogo Miguel Reis Silva

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. Miguel Nuno Dias Alves Pupo Correia

December 2021

Dedicated to José Manuel Freixo Nunes, who inspired me to pursue computer science and taught me the basics of programming which helped me in my journey during these past five years.

Acknowledgments

I would like to express my great appreciation to my supervisor, professor Pedro Miguel dos Santos Alves Madeira Adão, who kept in touch with me quite regularly in order to make sure that the project behind this thesis was developed smoothly. He also helped me getting in contact with multiple people who had a positive impact in the outcome of this thesis.

I am particularly grateful for the assistance that my co-supervisor Diogo Miguel Reis Silva gave me. He came up with many incredible ideas which I ended up using, and he always helped me find a solution when I faced difficult problems. Even though he was constantly busy with work he was still always able to set apart some time to give me constructive feedback and to answer all of my questions.

I also wish to thank professor Miguel Nuno Dias Alves Pupo Correia for reading my thesis proposal and for giving me constructive feedback and improvement suggestions.

Assistance provided by Simão Pedro Patrício da Silva was greatly appreciated. He helped me setting up my solution with one of the production applications used by the university, which yielded me some important data.

I would like to acknowledge the two pen-testers who helped me conduct a crucial study, as well as the 100+ participants who answered a questionnaire that we designed to evaluate a specific feature on the final solution.

Finally, I want to thank my parents and my grandparents. Even though they weren't directly involved in the development of this thesis, they gave me a great deal of emotional support and encouraged me during tough times.

Resumo

Honeypots são recursos “isco” de computador destinados a serem atacados com o objetivo de desviar a atenção de recursos mais valiosos e/ou coletar dados sobre ataques recebidos. Existem várias ferramentas de *honeypots* que ajudam os defensores a protegerem os seus sistemas. No entanto, a maioria dessas ferramentas cria *honeypots* isolados que são executados em paralelo com esses sistemas. Se os *honeypots* fossem incorporados nos próprios sistemas, eles deveriam ser mais facilmente encontrados por atacantes e também passariam mais facilmente como uma parte legítima desses sistemas, resultando assim na coleção de mais dados.

Neste trabalho, usamos uma metodologia não convencional na qual, em vez de criar *honeypots* autónomos, os integramos automaticamente em aplicações web existentes. Algumas soluções já tentaram esta abordagem mas, até onde sabemos, a nossa consegue criar *honeypots* mais customizáveis do que as mesmas. A nossa solução é composta por três componentes principais: (i) um proxy/balanceador de carga que intercepta o tráfego entre os utilizadores e a aplicação web e determina quando os utilizadores devem ser direcionados para um *honeypot*; (ii) um servidor chamado MockingPot que gera e dispõe as páginas *honeypot* conforme especificadas pelos defensores num ficheiro de configuração declarativo; e (iii) o mTANNER, que analisa os pedidos enviados aos *honeypots*, coleta dados maliciosos e produz respostas que os atacantes esperariam ao enviar pedidos maliciosos.

Os *honeypots* são apresentados aos utilizadores como páginas web que podem ser amplamente configuradas pelos defensores. Configurar a nossa solução é simples, pois não requer nenhuma modificação às próprias aplicações, mas sim aos proxies reversos / balanceadores de carga. Além disso, desenvolvemos um método de camuflagem das páginas *honeypot*, em que as estilizamos automaticamente com base em informações extraídas das aplicações que os defensores pretendem proteger. Efetuámos um estudo para determinar a eficácia deste processo, o qual revelou que os participantes não conseguiram diferenciar as páginas camufladas dos *honeypots* de páginas reais.

A nossa avaliação mostrou que a carga computacional introduzida ao usar esta solução é insignificante e que nosso método para camuflar as páginas *honeypot* é eficaz. Também mostrámos que o mTANNER é capaz de gerar respostas que mantêm atacantes interessados por longos períodos de tempo. Para além disso, esta solução deve ser compatível com qualquer aplicação web, permitindo que defensores configurem e integrem facilmente vários *honeypots* nas suas aplicações para as proteger.

Palavras-chave: *Honeypots* web, Integração automática, Emulação de vulnerabilidades

Abstract

Honeypots are "decoy" computer resources meant to be attacked in order to divert attention of attackers from more valuable resources and/or collect data on incoming attacks. There are many honeypot tools that can help defenders protect their systems. However, most of these tools deploy isolated honeypots that run in parallel with those systems. If the honeypots were instead incorporated into the systems themselves, they should be more easily found by attackers and also more likely to pass as a legitimate part of those systems, consequently resulting in more data being collected.

In this work we used an unconventional methodology where instead of deploying stand-alone honeypots we automatically incorporate them into existing web applications. A few solutions have already attempted this approach but, to the best of our knowledge, ours can deploy more customizable honeypots than them. Our solution is composed by three main modules: (i) a proxy/load balancer which intercepts traffic between users and the web application and determines when users should be directed to a honeypot; (ii) a server called MockingPot which generates the honeypots as specified by the defenders in a declarative configuration file; and (iii) mTANNER which analyzes the requests sent to the honeypots, collects incoming malicious data, and generates responses that attackers would expect when they send malicious payloads.

Honeypots are presented to the users as web pages which can be extensively configured by defenders. Deploying our solution is very simple as it does not require any modifications to the applications themselves but rather to the reverse proxies/load balancers. Additionally, we developed a method of camouflaging the honeypot pages by automatically styling them based on information collected from the applications that defenders intend to protect. We conducted a study to determine the effectiveness of this process, and it revealed that the participants could not tell apart the camouflaged honeypot pages from real web pages.

Our evaluation showed that the overhead introduced when using this solution is negligible in terms of performance. We also showed that mTANNER is capable of generating responses which keep attackers engaged for long periods of time. Furthermore, this solution should be compatible with any web application, allowing defenders to easily configure and integrate multiple honeypots into their applications in order to protect them.

Keywords: Web-based honeypots, Automatic integration, Vulnerability emulation

Contents

Acknowledgments . . . v
Resumo . . . vii
Abstract . . . ix
List of Tables . . . xiii
List of Figures . . . xv
Listings . . . xvii
Glossary . . . xix

1 Introduction **1**
1.1 Types of honeypots . . . 3
1.1.1 Design criteria . . . 3
1.1.2 Deployment criteria . . . 4
1.2 Motivation . . . 4
1.3 Objectives . . . 5
1.4 Our solution . . . 6
1.5 Thesis Outline . . . 7

2 Related Work **9**
2.1 WordPress honeypot module . . . 9
2.2 B.Hive: A Zero Configuration Forms Honeypot for Productive Web Applications . . . 11
2.3 CanaryTokens . . . 12
2.4 HIHAT . . . 14
2.5 SNARE & TANNER . . . 15
2.6 Requirements . . . 17

3 Solution **19**
3.1 Solution designs . . . 19
3.2 Solution architecture . . . 21
3.3 Solution behavior . . . 21
3.4 Reverse proxy / Load balancer . . . 24
3.5 MockingPot . . . 25
3.5.1 Custom honeypots . . . 27

3.5.2 Template honeypots . . . 28
3.5.3 Text honeypots . . . 31
3.5.4 Page camouflaging . . . 33
3.6 mTANNER . . . 35
3.6.1 Limiting the emulation process . . . 35
3.6.2 Additional dummy database customization . . . 37
3.6.3 Serving files in LFI honeypots . . . 40
3.6.4 Isolating Docker-based emulation . . . 41
3.6.5 Other modifications . . . 42

4 Results **43**
4.1 Page camouflaging study . . . 43
4.1.1 Results . . . 44
4.2 mTANNER modifications study . . . 46
4.2.1 Results . . . 47
4.3 Proxy overhead study . . . 51
4.3.1 Results . . . 51
4.4 Requirements . . . 52

5 Conclusions **53**
5.1 Achievements . . . 54
5.2 Future Work . . . 55

Bibliography **57**

A Screenshots **61**

B Technical Diagrams **66**
B.1 BPMN diagram representing the behaviour of our solution . . . 66

C MockingPot configuration parameters **68**

List of Tables

2.1 Summary table showing which requirements were met by the previous solutions 18

4.1 Average time per request obtained across multiple benchmarks 51
4.2 Summary table showing how our solution meets the requirements 52

List of Figures

3.1 Solution architecture . . . 21
3.2 Honeypot log in page example . . . 22
3.3 Honeypot log in page with an error message . . . 23
3.4 Honeypot log in page example with SQL Injection emulation results . . . 23
3.5 Template honeypot page without any variables defined . . . 30
3.6 Template honeypot (IST) . . . 31
3.7 IST log in basePage . . . 31
3.8 Template honeypot (ULisboa)[1] . . . 31
3.9 ULisboa log in basePage . . . 31
3.10 Text honeypot simulating PHP script when first visited . . . 32
3.11 Text honeypot simulating PHP script when a LFI payload is sent . . . 32
3.12 Text honeypot simulating PHP script using the original TANNER tool . . . 35
3.13 Text honeypot simulating PHP script using mTANNER . . . 36
3.14 Text honeypot simulating PHP script incorrectly emulating SQL Injection . . . 36

4.1 Question from the control form . . . 44
4.2 Question from the experimental form[2] . . . 44
4.3 Percentage of correct answers in both questionnaires . . . 45

A.1 Honeypot log in page with an error message . . . 62
A.2 Honeypot log in page example with SQL Injection emulation results . . . 62
A.3 Template honeypot page based on the IST log in page . . . 63
A.4 IST log in page . . . 63
A.5 Template honeypot page based on the ULisboa log in page . . . 64
A.6 ULisboa log in page . . . 64
A.7 Question from the control group form . . . 65
A.8 Question from the experimental group form . . . 65

Listings

3.1 NGINX configuration example 24
3.2 Honeypot configuration example 26
3.3 Custom honeypot configuration example 27
3.4 Handlebars template file example 28
3.5 Example of a recognized Handlebars condition 29
3.6 Text honeypot configuration example 32
3.7 Database configuration and dump produced using TANNER 38
3.8 Database configuration and dump produced using mTANNER 39
3.9 Text honeypot configuration using the lfi_files modifier 40
4.1 SQL Injection exploit from the control group 48
4.2 Python exploit used to obtain the admin's password 49
C.1 All supported honeypot configurations 68

Glossary

BPMN Business Process Model and Notation

CE Command Execution

CSRF Cross-Site Request Forgery

DOM Document Object Module

IDS Intrusion Detection System

IPS Intrusion Prevention System

IST Instituto Superior Técnico

LFI Local File Inclusion

RFI Remote File Inclusion

SQL Structured Query Language

ULisboa Universidade de Lisboa

WAF Web Application Firewall

XSS Cross-Site Scripting

XXE XML External Entity

Chapter 1

Introduction

Cyber-security is a major concern in the modern world. According to Risk Based Security [1], 7098 security breaches were reported in 2019 and over 15.1 billion records were exposed. Furthermore, the number of exposed records in 2020 exceeded 37 billion, which represents a 141% increase in comparison to 2019. Due to security breaches, several companies suffer monetary losses every year. Therefore, it is critical for companies to invest in reliable defense mechanisms. Currently, three technologies are commonly used to defend systems and networks:

- *Intrusion Detection Systems (IDS)*: There are several types of IDS, which can operate at different layers. Their goal is to constantly monitor traffic within a certain network or system while searching for malicious activity. Once they detect something suspicious, they will generate some type of alert in order to notify the defenders and allow them to take action (when necessary);

- *Intrusion Prevention Systems (IPS)*: IPS operate in the network layer. Similarly to IDS, they are also used to monitor network traffic. However, once a threat is detected, an IPS is capable of taking action against it without any intervention from the defenders. IPS can control accesses to the network in which they are deployed, and so they are able to block attacks before they reach the targeted systems;

- *Web Application Firewalls (WAF)*: These defense mechanisms operate in the application layer. WAF protect web applications by monitoring incoming requests and matching them against sets of rules. If a request does not comply with those rules, then it will not be given access to the application. Therefore, WAF can be used to protect against common web attacks [2] such as Cross-Site Scripting (XSS) and Structured Query Language (SQL) Injection.

These technologies are meant to be used in conjunction. All of them have different functionalities, but their ultimate goal is to improve security and protect resources. However, all of them face certain challenges. WAF can heavily impact the performance of a web application depending on the analysis that is performed for each request [3]. A thorough analysis can negatively affect the users' experience, while a superficial one might let some malicious requests pass.

Common IDS and IPS solutions usually flag a higher number of false positives than the number of true positives (actual attacks) that they detect [4]. Therefore, IDS regularly generate erroneous alerts and IPS often reject legitimate traffic since they can take immediate action without needing consultation from the defenders. Furthermore, since these solutions commonly have to deal with all incoming packets/requests, they can often under-perform in high traffic networks due to not being able to process all of the incoming data [5].

Some of these technologies also require complex configuration processes, regular maintenance, and fairly high costs (with some IDS reaching tens of thousands of dollars). Nevertheless, there is another defense mechanism that is often forgotten and undervalued by defenders, which is not affected by any of these issues: honeypots. A honeypot is a resource whose value lies in being attacked. These resources are usually computational systems or even simple pieces of data that don't actually contain any valuable information, but are deployed in an attempt to attract attacker activity to them. Honeypots can be created for multiple different purposes at a time:

- *Collect data from attacks:* honeypots can be placed into attractive environments with the purpose of collecting incoming malicious requests. Among other uses [6], these requests can then be analyzed manually or with automatic methods in order to gather information about the attempted attacks. This information can then be used by defenders in order to further secure their systems against such attacks, either by improving the system's defenses or by revoking the attackers' access (which can, however, be easily countered with the use of VPNs or anonymous proxies). Most honeypots will emulate vulnerabilities or use other deception techniques [7, 8] in order to maximize their attractiveness and to prompt attackers to execute their exploits. Furthermore, effective honeypots are deployed into locations that would not be usually accessed by legitimate users under any circumstance. When placed into such locations, almost all of the data collected by a honeypot is guaranteed to come from malicious users, making it much more reliable and useful than data collected with Intrusion Detection Systems, as we will discuss further ahead in this chapter;

- *Keep attackers away from real resources:* to understand this purpose, let us imagine a scenario where a seemingly vulnerable honeypot resource is deployed into a publicly accessible network. Before conducting their exploits, attackers need to perform a reconnaissance phase on the targeted environment in order to discover the best entry points into the system [9]. This means that attackers will most likely recognize that resource as the best path towards breaking the security of the system, thus focusing all of their attention towards attacking it until they realize its true decoy nature. Therefore, the introduction of this honeypot resource protects the remaining resources from most of the malicious activity directed to the network. Furthermore, by collecting data from such an early attack stage, honeypots become even more effective tools to understand the attackers' motives and to allow defenders to counter their attacks before they even happened;

- *Waste attackers' time:* this purpose is tightly connected to the previous two. The longer attackers interact with a honeypot, the more information on them will be collected. In addition, if the attackers waste long periods of time with the fake resources, they will consequently have less time and often less patience to deal with the real ones. Deception strategies can be employed to increase the average time periods that attackers spend on honeypots, such as embedding fake sensitive files into systems for intruders to explore [7].

A collection of honeypots used within the same network is called a honeynet, and it can be used to monitor large network areas that can't be fully covered by a single honeypot. While building a honeynet, defenders can even use a combination of different types of honeypots at the same time in order to achieve an enhanced coverage, since each one has different applications, strengths and weaknesses. In the next section, we will introduce some of those types according to two different criteria: design (interaction level) and deployment (purpose of use). It is important to note that some authors consider the existence of different types and/or criteria [10].

1.1 Types of honeypots

1.1.1 Design criteria

Pure: pure honeypots are fully functional physical production systems that provide real services and have some sort of tap installed, such as a keylogger, in order to monitor malicious activities that target them (*ssh* attempts, keystrokes after gaining access, file uploads and so on). Setting up pure honeypots can be both a time-consuming and complex operation but they provide the highest levels of authenticity and, if the tap is well concealed, they are very difficult to detect. However, they yield the highest risk level since these honeypots correspond to real systems running fully-fledged operating systems that can be used to attack others if compromised;

High-interaction: these honeypots correspond to virtual machines that imitate production systems by running real operating systems and hosting a large variety of services that attackers would expect to find. Several of these can be hosted on a single physical machine, which is much more cost-effective than using pure honeypots. Additionally, high-interaction honeypots also yield a lower risk level since the ones that are compromised can be easily sandboxed and restored before being used as a stepping stone for further attacks. However, there are quite a few methods that can be used to detect virtualized environments, which results in these honeypots being easier to detect than the previous ones;

Low-interaction: these honeypots emulate the most important functionalities of a set of services that attackers would expect to find or that the defenders are interested in. Since these emulations do not offer the full functionality of the services they imitate, they can be easily deployed with fewer resources than the previous types and they yield very low risk levels. However, they can often be detected by experienced attackers after extended interaction periods with the emulated functionalities. Therefore,

they should normally collect less data than the previous types due to the limited interaction capabilities with the attackers.

1.1.2 Deployment criteria

Research: research honeypots are deployed into isolated environments with the purpose of collecting as much information as possible regarding the techniques and motives of the attacker community. They usually contain data that can be traced after it is stolen in order to further analyze attacks. This information can then be passed onto several organizations in order to make them aware of the threats that they should expect and to help them fortify their defenses against the detected attack patterns;

Production: these honeypots are deployed into production environments with the main purpose of keeping attackers away from the real resources. Usually, low-interaction honeypots are used since the risk of compromise is small, they are easy to deploy, and the amount of collected data is normally enough to help mitigate existent vulnerabilities.

1.2 Motivation

By using honeypots in conjunction with the defense technologies that we mentioned earlier, defenders can further secure their resources. Simple honeypots are easy to deploy (both for research and production purposes), they require no further maintenance and no additional costs. Despite this, they can be a very effective technique for protecting against attacks. Unlike some WAF [3], honeypots can be associated with web applications without having a noticeable impact on their performance. Unlike IDS and IPS [4], honeypots can detect malicious activity with a very small rate of false positives, thus making manual log analysis much faster and more efficient. Furthermore, honeypots normally deal with a very small subset of the total incoming traffic, and so they should perform well even with high-speed traffic.

Several tools have been developed over the years to help defenders deploy different types of honeypots automatically. However, to the best of our knowledge, most of these tools deploy isolated honeypots which either must be manually associated with existing systems (in order to protect and/or collect data sent to them) or they serve as systems of their own (for instance, in order to collect attacker data for generic research purposes). In this first scenario, integrating honeypots into the system that is being protected is an essential step, as they will be more easily found by attackers and they will be more likely to pass as a legitimate part of the system, which should consequently result in more data being collected. However, integrating honeypots manually into an existing system can be a tedious task, especially when they are large and complex. Therefore, we thought that this integration process should also be one of the tasks performed by such tools. If a honeypot can be fully generated in an automated manner, it also makes sense to automatically integrate it into the system that it is meant to protect.

Considering this, we started focusing on this scenario and looking for tools that were capable of deploying honeypots and performing such integration. We found very few solutions, all of which were fairly limited due to the compromises made regarding the honeypots' features in order to facilitate the integration process. For example, one of those tools [11] (that we present further ahead in Section 2.2) is only capable of deploying honeypots which detect payloads sent within HTML form fields. As such, we decided to attempt the development of a solution that would allow us to integrate more customizable honeypots into existing systems. Since most modern systems present some sort of web interface to their users, we decided to focus on integrating web-based honeypots into those interfaces. This allowed us to develop a unified strategy that introduces honeypots into systems of different complexities and sizes without having to worry about their implementation details.

1.3 Objectives

In order to further explore the integration process of web-based honeypots into existing systems, we defined our goal to be the creation of a new solution which implements the following functionalities:

- Easy generation of highly customizable web-based honeypots;
- Automatic integration of honeypots into web applications;
- Maximization of the honeypots' data collection and distraction capabilities.

To meet the last functionality, we decided to perform attack classification and vulnerability emulation in the honeypots. By classifying attacks, this solution will increase the effectiveness of the manual log analysis process. Defenders will be able to act more swiftly after malicious activity is detected by the honeypot due to the information introduced by the classifier. Furthermore, by emulating responses to the detected attacks, the honeypots will create the illusion that the web application executes incoming payloads (without actually being vulnerable to them), which will engage attackers and incite them to send more payloads. This will maximize their data collection, distraction and time wasting capabilities.

In order to fully comprehend the effectiveness of a system designed with these functionalities in mind, we believe it is necessary to define a set of requirements that should be met in an ideal version of such a system. As such, to assess each of the solutions that will be mentioned in the Related Work section with regard to our goals, as well as our own solution which will be introduced later on, we will consider the following set of requirements. They are based on the ones proposed by Müter et al. in their paper *A Generic Toolkit for Converting Web Applications Into High-Interaction Honeypots* [12].

- *Functionality:* After the honeypots are deployed, the web application should offer at least the same functionality that it offered originally;
- *Performance:* Introducing the honeypots should have a negligible impact on the performance of the existing web application, even in production environments;

- *Information coverage:* The honeypot should be able to collect information from any method that attackers can use to attack them;
- *Security:* Attackers should not be able to use the honeypots in order to harm other users or systems;
- *Compatibility:* The solution should be compatible with most web applications, regardless of the programming languages used to implement them;
- *Longevity:* The system should be able to keep attackers interested for long periods of time, thus maximizing the number of collected interactions;
- *Undetectability:* The presence of the honeypot should not be easily recognizable by those who access the web application;
- *Data Analysis:* The solution should automatically analyze the data collected by the honeypots in order to support manual log analysis.

In Chapter 2, in order to evaluate the effectiveness of related solutions with regard to our goals, we will point out which of these requirements they met. Successfully achieving all of the requirements is a difficult task since some of them could contradict another depending on how the solutions are implemented. It should also be noted that these requirements are specific to our objectives. Therefore, even if one of the presented solutions doesn't meet any of them, this doesn't mean that it is inadequate since it might have simply been originally constructed with different objectives in mind.

1.4 Our solution

In this section we will briefly introduce the solution that we came up with in order to fulfil our objectives and to meet as many of these requirements as possible, which is composed by three modules. The first one is a *reverse proxy/load balancer* that intercepts traffic between an existing web application and its users. It is responsible for directing all incoming requests to that application and for determining whether they should also be sent to the honeypots based on the application's responses.

The second module is *MockingPot*, a server where the honeypots will be hosted. Defenders can extensively configure the behaviors of these honeypots in a declarative configuration file. The specified behaviors are then automatically implemented into honeypots, thus allowing defenders to generate those honeypots easily and without having to deal with implementation details. Honeypots are shown to attackers as web pages whose contents can also be entirely defined by the defenders, as we will explain in detail in Chapter 3.

The last module is *mTANNER*, which provides the attack classification and vulnerability emulation that we mentioned previously. mTANNER is an extension of an existing tool which we present in Section 2.5. It is responsible for finding malicious payloads and sending them to the vulnerability emulators. These emulators will then produce responses that attackers would expect after sending a malicious payload, which are later included into the honeypot web pages and sent back to them.

Our solution is easy to set up, since there is no need to modify existing web applications (only existing reverse proxies/load balancers). It should be compatible with any web application (regardless of the programming language it was implemented in) and it also automatically integrates honeypots into existing web applications by presenting them to the attackers as if they were web pages served by the application itself (from their point of view). Additionally, we developed a method of camouflaging our honeypot pages by automatically styling them based on information collected from the protected web applications.

In Chapter 3 we will discuss in detail how we implemented this solution. We will explain how the reverse proxy/load balancer determines when users should be directed to a honeypot, as well as many other details and features that we haven't introduced yet.

1.5 Thesis Outline

This thesis is structured as follows: Chapter 2 (Related Work) presents papers and tools that are related to the objectives that we defined. Chapter 3 (Solution) explains the implementation details of our final solution: first we discuss its general architecture and behavior, and then we explain how each module works; Chapter 4 (Results) discusses all the studies that were conducted to evaluate the effectiveness of our solution (page camouflaging effectiveness, mTANNER modifications and performance overhead); and in Chapter 5 (Conclusions) we present the conclusions that can be taken from this thesis.

Chapter 2

Related Work

We will now present some of the papers/tools regarding web-based honeypot techniques that we analyzed during the research that was conducted for this thesis. We chose these solutions because they were the most related to the objectives that we had previously defined. In the next sections we will describe them along with which vulnerabilities they target and what their strengths/weaknesses are. Some of them also attempt to integrate honeypots into web applications, as we will discuss further ahead.

2.1 WordPress honeypot module

WordPress [13] is an open-source Content Management System that is currently used by more than 60 million websites. It is mainly used for blogging, but it can also be used to create other types of web content, such as online stores and forums. As such, WordPress web applications are a popular target for attackers [14], especially since many organizations use this software to create a blog page.

Users can extend and modify the core functionality offered by WordPress by installing plugins. Plugins are packages of code that can be developed by anyone and shared with other users. Therefore, unlike the actual WordPress source code which receives constant updates and fixes by many developers and from members of the community, it is quite common for these plugins to contain vulnerabilities and introduce them into the web applications in which they are installed. As such, third party plugins are the most common places where attackers will look for vulnerabilities.

WordPress honeypots usually consist on creating WordPress applications which intentionally use insecure plugins [15] with the intent of collecting data on attacks that explore vulnerabilities in those same plugins (either for protection or research purposes). This method is quite effective when facing attacks against vulnerabilities in those plugins, but it does not allow to detect attacks against others that are not installed. If an attacker wants to exploit a vulnerability in a plugin that is not currently installed, the reconnaissance phase will indicate that the web application is not vulnerable and the opportunity to collect valuable information will be missed. In order to collect information from these scenarios, researchers need to create honeypots that cover all of the targeted resources and different software versions.

Cernica and Popescu [16] propose a honeypot module that extends existing WordPress high-interaction honeypots in order to allow them to collect data from undisclosed vulnerabilities. This module acts as a proxy between the incoming malicious requests and the actual web application, and it can identify the requests that ask for specific resources (either plugins or themes). Afterwards, it will automatically search for vulnerable versions of those resources by looking for keywords (such as "SQL", "RCE", etc.) in the "changelog" sections of the pages where the resources can be downloaded. If some of those keywords are found for a specific version, then that same version is considered vulnerable and is installed on the running instance of WordPress before the request is processed by the application. If none of the keywords are found in any version, the latest one is installed.

With this implementation, from an attacker's points of view, the preliminary attack phase will be successful most of the time since it will conclude that all of the requested resources that are meant to be exploited are available in the web application. As such, the actual attack phase will most likely ensue, which allows the honeypot to collect data that would have been previously unobtainable with traditional implementations.

There are still, however, a few intrinsic issues with this module. One of them is mentioned by the authors: sometimes the "changelog" page of a resource can indicate that a vulnerability was fixed without using any of the keywords that the module looks for. For instance, in the testing phase that was conducted by the authors, some of the exploits failed because in the "changelog" it was only mentioned that "security issues" had been solved, instead of referring to the actual types of vulnerabilities that had been patched. By adding more words to the list, this method will eventually work with almost every resource, but there will still be others that won't be correctly interpreted due to the use of unexpected terminology.

Another issue that is related to this version selection procedure is that the authors want to target 0day exploits with the honeypots but, by installing a dated and vulnerable version of a resource, it is very unlikely that an attacker will exploit an undisclosed vulnerability when there are other known vulnerabilities that can also be targeted. If attackers possess knowledge of a 0day, then they are most likely to only use it after all other exploits fail (and assuming that the systems they want to access are valuable enough to justify it). For this reason, even though the module makes it possible to collect data on undisclosed vulnerabilities, it will probably be a rare occurrence.

Finally, using this module leads to a very generic honeypot. Since this solution attempts to install every plugin that is requested, the attackers will probably grow suspicious of the fact that each resource that they need is conveniently installed on the targeted web application. This effect would be especially felt while running WordPress vulnerability scanning tools in an implementation of this module, since they would most likely return that the target is vulnerable to an unrealistic amount of attacks, which is an enormous giveaway of its true nature as a honeypot. Furthermore, the time needed to download a resource could also make attackers aware of this reality. It is essential for the honeypot to remain unnoticed for as long as possible, since the collected data will be more valuable the longer the attacker interacts with it.

Regarding our requirements, only *Information Coverage* was met by this solution since the honeypot created with this module can collect attack data regardless of the requested resources (as long as they

exist). Most of our requirements were not met because this solution is not meant to be associated with regular web applications. As mentioned, this module is only supposed to be used with applications that act as high-interaction WordPress honeypots. However, a similar module could probably be designed to deal with all of the honeypot features while maintaining a regular WordPress application behind it. For example, this hypothetical module would be able to create a new instance of the application for each attacker (during the reconnaissance phase or when an attack was detected), install the requested resources and direct the malicious traffic there. In contrast, legitimate traffic would be sent to the original WordPress application, which would maintain its intended functionality without being exposed to any of the additional features introduced by the proxy.

2.2 B.Hive: A Zero Configuration Forms Honeypot for Productive Web Applications

In this paper [11], Pohl et al. present B.Hive, an approach that can automatically blend honeypot components into existing web applications without altering them. Those components are invisible to legitimate users and, as such, it is likely that only attackers will interact with them. This solution is capable of detecting attacks that inject malicious data through form fields, such as XSS and Cross-Site Request Forgery (CSRF) attacks, in order to exploit some of the most prevalent vulnerabilities in web applications.

In order to achieve this, B.Hive acts as a proxy between an existing web application and the end users. When a user requests a web page from the application, B.Hive intercepts that page and, if it locates any forms inside, modifies them by injecting additional fields. These new fields are invisible to legitimate users, and so they will not be filled under normal circumstances. However, it is common for vulnerability scanners to fill all fields with malicious payloads in order to search for vulnerabilities. Therefore, if B.Hive intercepts a request from a user where a honeypot field is not empty, it is likely to contain a malicious payload. Field manipulation is one of the techniques most commonly used during the preliminary attack phase, and so this technique allows to detect attacks at an early point in time and gives defenders better chances to act against them. B.Hive also intercepts the HTTP requests produced after these modified forms are filled by the users and removes the injected fields before directing them to the application, thus abstracting it from this honeypot functionality.

This solution uses several techniques in order to adapt the injected form fields to the context of the web page, thus allowing it to remain undetected even after being manually inspected by attackers. Some examples of these techniques are: selecting plausible field names from a database containing web forms from the 10000 most popular websites [17]; determining a plausible position to inject the forged fields; and even computing the ordering of the attributes that is most frequently used in the original pages. The consistency achieved between the injected and the original forms leads B.Hive to successfully trick attackers into interacting with the system without realizing its true honeypot nature. Furthermore, since this solution provides all of the same functionality that is offered by the original application, the system is more likely to be attacked than a standalone honeypot.

Results from the evaluation conducted by the authors shows that 96.22% of the forms that were tested can be protected by B.Hive, and that humans are unable to identify the injected form fields when looking at the field names. Besides this, it was also shown that, in all tested cases, penetration testing tools recognized the injected fields as possible attack targets. Finally, the performance of this solution was also evaluated, and it was concluded that B.Hive could be successfully used to protect large scale web applications while using both caching and parsing.

There are however a few shortcomings of this solution. The first one is that it focus on intercepting malicious requests, but afterwards nothing is done with the data collected by the honeypot. For instance, an algorithm could be executed in order to detect and classify attack payloads in the intercepted requests. This would be a useful feature since this solution is based on existing web applications and, as such, attack classification could allow defenders to be aware of actual attacks and act sooner. In contrast, a stand-alone honeypot would not benefit as much from such an algorithm because, since these kinds of systems do not hold any sensitive and important data, they do not need to be actively monitored and analysis of the collected data can be performed at a later date.

Another shortcoming is that B.Hive can only detect malicious requests with payloads that are sent to the server as form data, specifically within the injected fields. This means that B.Hive is unable to detect payloads sent through other locations, such as the URL parameters or the original form fields.

Finally, we speculate that B.Hive is only capable of performing this field injection operation if the forms on the web pages served by the application are rendered server-side. If a form on a web page is rendered on the client's browser, this solution won't be able to detect it and, therefore, the injection operation would be unsuccessful. Since client-side rendering has become an emerging trend with the rise of single page application frameworks such as React and Vue.js, this limitation affects the effectiveness of this solution.

Regarding our requirements, the following ones were fully met by this solution:

- *Functionality:* the functionalities of existing web applications can be kept;
- *Performance:* evaluation results indicate that this solution has negligible effects on the performance of the associated applications;
- *Security:* the injected form fields can't be used to attack other systems;
- *Undetectability:* the injected form fields are hard to detect due to their generation process.

The following one was partially met:

- *Compatibility:* this solution can be associated with existing web applications independently of the programming language as long as they render forms server-side.

2.3 CanaryTokens

CanaryTokens [18] are open-source resources developed by Thinkst Applied Research that can help defenders track malicious activity on a network. These resources (tokens) can be manually embedded

into systems to quickly detect security breaches. They are created by specifying a note and an e-mail address or webhook URL. Upon deployment, if they are ever triggered, they will send an alert with that same note to the associated address or URL.

There are several types of tokens and each of them is triggered in a different manner. For example, both the HTTP and DNS tokens are accessed through a URL that is generated after their creation, although the first one is triggered by the request and the second one is triggered by the DNS lookup. There are also many other types that don't generate URLs, such as the SQL Server token which sends an alert when an action is performed in a specific SQL Server table.

Since there is such a large variety of tokens, the defenders can usually find ways to insert them into various places in the system that need to be monitored. Tokens should be deployed in sensitive locations that are not supposed to be visited by regular users in order to achieve maximum efficiency. For example, an image token can be introduced into a web page containing a company's fake financial information. Then, if an attacker is able to access that page, the image will use the same method as HTTP tokens to alert the defenders. In such a scenario, it is very likely that the generated alert corresponds to a security breach. After receiving alerts, the defenders can then evaluate the situations and restrict the access of any attackers that might have infiltrated a sensitive area of the system.

Distributing these resources throughout a system requires imagination and abundant knowledge of the system's structure. Even though it is easy to deploy, its effectiveness relies heavily on the capacities of the defenders. Capable defenders can use CanaryTokens to detect accesses to sensitive documents, changes to SQL tables and many other security breaches such as Local File Inclusion (LFI) attacks. Furthermore, they allow defenders to introduce honeypots into existing web applications without having to perform any major modifications in the source code.

Despite all of these possibilities, there is a major drawback of using this solution. While effective at detecting such accesses, these resources are unable to collect data on the methods used to reach those locations. It is essential for those methods to be discovered in order to perceive the exploited vulnerabilities and patch them, thus preventing attackers from performing those same unintended accesses again. Otherwise, the vulnerabilities that have been exploited could remain undiscovered and continue to be utilized in order to threaten the security of the system. Furthermore, even though these resources can be easily embedded into web applications, they still must be manually integrated by defenders.

Regarding our requirements, the following ones were fully met by this solution:

- *Functionality:* the functionalities of existing web applications are not affected;
- *Performance:* the performance of existing web applications are not affected;
- *Security:* attackers can't use this solution to attack other systems;
- *Compatibility:* this solution can be associated with any web application, regardless of the programming language used to implement it;
- *Undetectability:* CanaryTokens should be undetectable by attackers.

2.4 HIHAT

HIHAT [19] is an open-source tool that allows defenders to automatically transform PHP applications into high-interaction web-based honeypots, while keeping the original functionalities offered by them. This tool is composed by three major components. The first one is the Logserver, which contains the MySQL databases where the data collected by the honeypots is stored. It is important for the Logserver to be deployed in a remote and secure location to ensure the protection of that data. Otherwise, since the application is turned into a high-interaction honeypot meant to attract attackers, they could eventually gain complete access to the system and alter the data stored in such databases (rendering it useless). A script is provided by the developers in order to automatically create the required MySQL tables.

The second one is Honeypot-Creator, a Java program that alters a target PHP application by inserting logging code into its original source code. This additional code is inserted into the beginning of each file and it will automatically read data sent to the application with GET and POST requests and insert it into the databases located in the specified Logserver. It will never interfere with the intended behavior of the original application and it should be undetectable by attackers. This will allow defenders to monitor all actions and accesses performed during attacks.

The third and final component is the Analysis Tool, which is mostly coded in PHP. It automatically filters and structures data to be more easily inspected by the defenders. Comparatively to the raw data collected by the honeypot, the data produced by the Analysis Tool also ensures a higher detection rate for attacks during the manual inspection process and a smaller rate of false positives. Some examples of the specific functionalities offered by this component are: filtering known attacks by searching for specific attack patterns via regular expressions; automatically detecting other tools and resources that attackers might try to download; and automatically tracking the location of attackers' IP addresses.

There are however some factors that threaten the effectiveness of this solution. Its development has been ceased since 2007, and so it will most likely have some incompatibilities with recent applications that could have been fixed if this tool had continued to receive updates. Furthermore, HIHAT is incompatible with web applications implemented in other programming languages.

Finally, this tool is meant to produce high-interaction honeypots, by inserting logging code into intentionally vulnerable web applications. Despite this, it could also be used with an existing secure PHP application in order to automatically provide it with the traffic logging and data analysis functionalities that were mentioned. However, the injected code causes a small performance overhead on all interactions with the web application (some of which can reach a 10% time increase, as presented by Müter et al. in their paper [12] which is based on this tool), which would make it undesirable for large-scale production environments.

Regarding our requirements, the following ones were fully met by this solution:

- *Functionality:* the functionalities of existing web applications are not affected;
- *Information Coverage:* all interactions with the application are logged, which encompasses attack data;

- *Security:* the injected code itself can't be used to attack other systems;
- *Undetectability:* the logging code should be undetectable by attackers;
- *Data Analysis:* collected data is automatically processed by the Analysis Tool.

2.5 SNARE & TANNER

The MushMush Foundation [20] is an organization dedicated to developing open-source software. Currently, they have two active projects and are maintaining a third one, with all of them being related to honeypot deployment. Among those projects, one in particular became a source of inspiration for the theme of this thesis. That solution is comprised by two different tools that are meant to be used together and which shall be introduced in the following paragraphs. Both of them are coded in Python and can be easily built with `docker-compose` [21].

The first tool is SNARE [22], which comprises the frontend part of the solution. SNARE is the successor of another project from MushMush called Glastopf [23], containing many of the same features along with the capability of turning existing web pages into attack surfaces. In order to achieve this, SNARE implements a cloner module, which can be used to clone pages at specified URLs. Afterwards, it creates a separate web server which displays those pages in order to imitate the cloned web application, and it automatically routes the received HTTP requests to its backend counterpart.

The other tool is TANNER [24], which comprises the backend component that implements most of the functionalities which make this solution so effective. TANNER is responsible for analyzing the HTTP requests sent to it by SNARE, by searching for attack payloads within them. It parses the parameters and the body of the received requests using regular expressions to find those payloads. If a match with a pattern of a specific type of attack is detected, then the request will be sent to an emulator designed for that same type of vulnerabilities.

Each emulator implements a different strategy for predicting the output of the detected payload in a vulnerable system. For example, a Remote File Inclusion (RFI) attack is emulated by downloading the malicious file from the specified remote location, executing it in a PHP sandbox and extracting its output. In contrast, SQL Injection attacks are emulated by injecting the payload into a vulnerable SQL query, executing it in a clone of a randomly populated database and extracting its outcome.

After the predicted output of an attack payload is obtained, TANNER sends it to SNARE, which appends it to the web page visited by the attacker. This creates the illusion that the server is vulnerable to that type of attack and that it executes the malicious payloads that are sent to it, which is often enough to engage attackers and lead them into sending more payloads to the server. Consequently, more data can be gathered by the honeypot and more of the attackers' time is wasted on exploiting these non-important resources.

TANNER also has another feature that contributes to the efficiency of this solution: the creation and management of ”sessions” using the requests sent by SNARE. A session is kept in the process memory while active, and is identified by an IP address, userAgent and session UUID. It also contains other

attributes, such as a list of the detected attacks, the timestamp of the session's creation and the cookies sent by the client or set by the server. This feature simplifies the analysis of the requests by grouping the ones that originated from the same IP and userAgent into a single object, until that object is considered to have expired (if some time passes without a new request being associated with that session).

After a session expires, it can be evaluated since no more requests will be associated with it. As such, a few more attributes are added to it, for instance, the expiry timestamp, the number of requests per second and a list of possible owners. This last attribute contains a prediction for the entity responsible for that session, and it can assume four different values: *admin*, which means that the requests were sent from the localhost; *crawler*, which means that the userAgent on the requests matched those used by popular web crawlers; *tool*, which means that requests were sent in fast succession but their userAgent does not match any of the ones used by the registered web crawlers (indicating the usage of an automatic tool); and *attacker*, which means that the malicious requests were sent manually. This prediction enriches the data collected by the honeypot since it can directly identify the responsible entity without requiring manual analysis of the requests. More complex entity classification processes have been proposed [25], but this one still seems to produce decent results.

For now we have only described this solution and pointed out its strengths but, during the research for this thesis, some issues have been found to reduce the desired level of efficiency. To mention a few of those issues:

- While running common web vulnerability scanners, it is reported that the server is vulnerable to almost every single attack since its aim is to seem exploitable by all the malicious payload that it can detect, which is a huge giveaway that the web server is a honeypot. Knowledgeable attackers will also often perceive this even when manually sending the requests. For example, if an attacker accessed the honeypot route using a parameter with value `<script>alert(1);</script>` and then used the same value with a different parameter, both payloads would be executed and the attacker would easily deduct the server to be a honeypot;

- TANNER populates a database with dummy records in order to emulate SQL Injection attacks, but the methods used in order to generate that data are too modular and lead to obvious incongruences. For example, all e-mail addresses follow the format `<string><year>` and the corresponding usernames follow the format `<string><optional separator><year>`, but the two strings and years are normally completely unrelated, which is an enormous giveaway that the data is fake. Furthermore, the generated credentials would be more believable if the language used to generate them could be specified;

- The provided web interface could be more organized. As it is, defenders might have difficulty analyzing the collected data. For example, if payloads are sent in the body of a POST request, they will not be visible in the web interface.

This solution creates a generic honeypot that attempts to seem vulnerable to as many payloads as it can detect. It strives to engage attackers by emulating the outputs of their payloads and injecting them into the visited web pages, but ultimately this effort is damped by the factors that reveal the true nature

of the server. Furthermore, the XSS emulation provided by this solution is undesirable with our goals in mind, since it would introduce actual XSS vulnerabilities into the associated web applications. This would allow attackers to embed malicious scripts into the honeypot web pages, which could be sent to legitimate users in an attempt to harm them.

Since this solution clones web pages from URLs and creates its own separate server, it is also less effective than a solution that is directly integrated into an existing web application (especially if it is meant to be used as a production honeypot).

Regarding our requirements, the following ones were fully met by this solution:

- *Information Coverage:* all malicious interactions with the honeypots are logged;
- *Compatibility:* should be able to clone the web interface of any application;
- *Data Analysis:* collected data is automatically processed by TANNER to generate valuable information.

The following ones were partially met:

- *Security:* attackers shouldn't be able to use the honeypots in order to harm other systems assuming XSS emulation is disabled;
- *Longevity:* responses to detected payloads are emulated, which keeps attackers interested, but the honeypots are easily detectable after a few requests are sent.

Despite this, if TANNER was slightly modified, we believe that it would be a suitable backend component to provide the analysis and emulation features that we desire. If the issues that we mentioned were mitigated, it could easily meet other requirements that it does not achieve as it is. Therefore, we decided to modify and use it as the provider for our data analysis and response emulation, as we will discuss in the next chapter.

2.6 Requirements

To conclude this chapter, we will now present a review table (table 2.1) that summarizes which of our requirements were met by the solutions that we just presented. The values used in the table are the following: Y (Yes) if a requirements is fully met; P (Partial) if it was partially met; And N (No) if it was not met at all.

The WordPress module (section 2.1) was able to meet the Information Coverage requirement since it can collect attack data regardless of the requested resources, as it can dynamically install them. However, we believe that this makes the honeypots too generic. Furthermore, this module is supposed to be used with applications that already act as high-interaction WordPress honeypots, and so none of the other requirements that we defined were met.

	WordPress module	B.Hive	CanaryTokens	HIHAT	SNARE & TANNER
Functionality	N	Y	Y	Y	N
Performance	N	Y	Y	N	N
Information Coverage	Y	N	N	Y	Y
Security	N	Y	Y	Y	P
Compatibility	N	P	Y	N	Y
Longevity	N	N	N	N	P
Undetectability	N	Y	Y	Y	N
Data Analysis	N	N	N	Y	Y

Table 2.1: Summary table showing which requirements were met by the previous solutions

B.Hive (section 2.2) was more tightly related to the objectives that we had previously defined and it was able to meet four requirements. This solution is less generic than the previous one but it does not meet Information Coverage since it is only capable of detecting malicious payloads sent via forms. It also does not fully meet the Compatibility requirement since it shouldn't work with forms rendered client-side.

CanaryTokens (section 2.3) were able to meet five of our requirements, but this solution does not share our objectives of automatic integration and data collection. Integrating CanaryTokens into a system requires the defenders to manually deploy them in tactical locations. Furthermore, these resources are unable to collect data on the methods used to reach those locations as their main purpose is to simply alert defenders of security breaches.

HIHAT (section 2.4) also met five of our requirements. Compatibility was not met since it is only compatible with PHP applications. Performance was not met either since the injected code causes a small performance overhead on all interactions with the web application. This solution is also meant to turn intentionally vulnerable web applications into high-interaction honeypots, while our goal is to integrate honeypots into regular web applications.

Finally, SNARE & TANNER (section 2.5) were not able to meet as many requirements as the previous solutions. However, TANNER seems like a good starting point for the attack classification and response emulation process that we desire, and with some more modifications it could meet some of the requirements that it currently doesn't.

Chapter 3

Solution

As mentioned before, our solution was designed with three main goals in mind: easy generation of highly customizable web-based honeypots, automatic integration of them into existing web applications, and maximization of protection/data collection capabilities. We also wanted our solution to meet as many of the requirements that were defined in Section 1.3 as possible. Before starting our implementation, we came up with two potential designs which we named based on designations from the Web Application Firewall Evaluation Criteria [26]. We will now briefly present their advantages and disadvantages.

3.1 Solution designs

Embedded design

This design consists on the web application importing a library that contains functions which detect the HTTP routes in use (for example, by recurring to some sort of ”server” object which contains such information) and deploy honeypots on available routes. These functions require some arguments to customize the honeypots, such as the specific route in which traffic will be handled by a honeypot and the vulnerabilities that it will emulate. Therefore, additional lines of code need to be added to the original source code of the application in order to call the functions in this library with the desired arguments. From the point of view of the defenders, adding those few lines of code to the web application automatically deploys the desired honeypots.

With this design, our solution would gain access to unique information and features which are only available within the program’s runtime logic (for example, the list of active routes). Furthermore, the pages generated by the honeypots are subjected to the same conditions imposed by the programming language. Therefore, they will be consistent with the pages displayed by the original web application.

However, defenders do need to make modifications in the source code in order to configure the honeypots, which can be considered undesirable in a system that already has deployed applications. Each web framework of each programming language also requires a custom frontend module to deal with the honeypot configuration and the communication with the backend module, and the modifications required to call the functions that configure the honeypots are different for each language.

Reverse proxy design

This design consists on using a web server as a proxy to intercept traffic between the users and the original web application. A separate server is also used to deploy the honeypots, which will be generated based on the details specified on a declarative configuration file. However, each honeypot will be reachable through a specific route in the same URL used by the application. If the web application receives a request targeting a non-existent route and it returns a response with the status code 404 (Not Found), the proxy will automatically intercept that response and send the original request to the honeypot server. Once the requests arrive in the server, they will only be handled by the honeypots if the requested routes match the ones specified in the configuration file. Otherwise, the error response provided by the original web application will be returned instead. With this implementation, the proxy won't need to be re-configured when new honeypots are added or when a web page is changed. Alternatively, if a traditional proxy implementation was used, the defenders would need to provide it with the updated list of active honeypot routes.

With this design, honeypots can be integrated into existing web applications without modifying them. The solution does not have direct access to the information that can be obtained with the previous design, but some of that information can be defined manually by the defenders to partially circumvent this issue. Furthermore, this design allows a single solution to be compatible with many programming languages.

However, different languages can have different ways of constructing and displaying web pages. As such, sometimes a proxy could generate responses that are inconsistent with the ones produced by certain applications, depending on the languages that they are implemented on. For example, React automatically inserts specific HTML tags into the head and body of the displayed web pages, which would need to be forged by the proxy in order to achieve consistency (but doing so would require a specific procedure for each language, which is also one of the disadvantages faced by the previous design).

Considering this comparison between both methods, we opted with the reverse proxy design due to its high compatibility with existing web applications and the fact that they do not need to be modified at all. We believe that its advantages outweigh its disadvantages, and that altering an application's source code as required with the embedded design makes it a less preferable choice. Furthermore, the additional information gained using the embedded design depends on the programming language, and in some cases it is quite limited.

After establishing this decision, we were finally able to start implementing our solution. We will present its architecture in Section 3.2, explain its behavior upon deployment in Section 3.3, how each of the three modules were implemented and how they perform their tasks in Sections 3.4, 3.5 and 3.6, and finally discuss if the requirements that we previously defined were met in Section 4.4.

3.2 Solution architecture

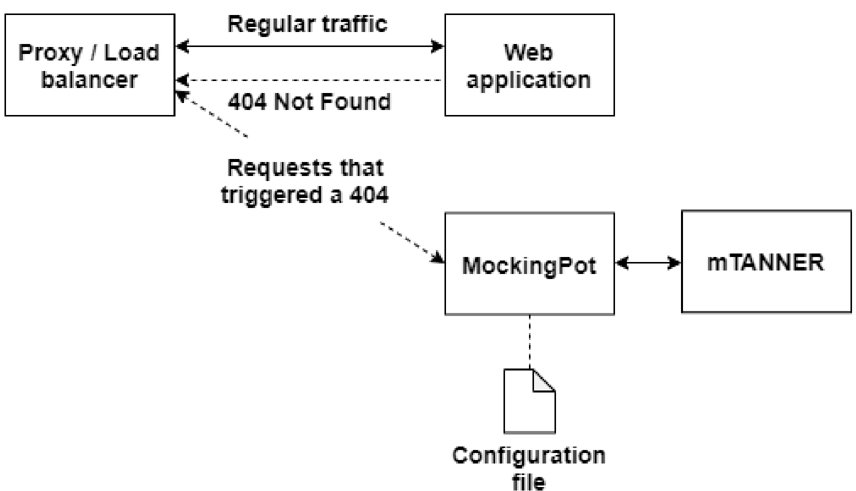


Figure 3.1: Solution architecture

The diagram shown in Figure 3.1 depicts the architecture of our solution, which is formed by the following components which we will introduce once again:

Reverse proxy/Load balancer: this component is a web server that intercepts traffic between the users and the existing web application. It is responsible for directing incoming requests to the web application and for determining whether they should also be sent to the honeypots based on the application's responses. If reverse proxies/load balancers (NGINX, HAProxy, etc.) are already used to send traffic to the application instances, then they could be modified to include our proxy logic.

MockingPot: this component creates a server where it deploys the honeypots as specified in a declarative configuration file. The defenders are able to configure the routes where they want the honeypots to be hosted, as well as other important details like the vulnerabilities that should be emulated and the specific parameters that should be inspected for payloads. The honeypots won't interfere with the web application and so this component can be deployed without any help from the development team.

mTANNER (modified TANNER): this component handles the data sent to the honeypot routes and analyses it. It is responsible for finding malicious payloads, detecting targeted vulnerabilities and emulating exploits in order to determine believable responses to malicious requests. This component also stores the collected data and generates believable dummy resources to be used during the emulation process, such as database schemas for Structured Query Language (SQL) Injection emulation.

3.3 Solution behavior

We will now explain the behavior of each component and how they communicate. This behavior is also represented in a Business Process Model and Notation (BPMN) [27] diagram which we included in Appendix B (see Section B.1).

Upon setting up our solution, rather than directly exposing a web application to the internet, the proxy will be responsible for receiving user requests and immediately directing them to that application (keeping the request routes, headers and parameters).

Then, the proxy waits for the application's response. If the web application returns an error (404 Not Found) due to a request targeting a non-existent route, then the proxy sends that request to the honeypot server and waits for its response. Otherwise, the proxy simply returns the application's response to the user. Additionally, defenders can allow requests with a specific HTTP header (by default, X-Testing-Purposes) to be ignored by this component, so that security testers can freely interact with the web application without having to deal with the honeypots (which would cost them time and effort).

While setting up MockingPot, defenders need to configure the honeypots that they wish to deploy by editing a declarative configuration file. There they can define the behavior of each honeypot: which route they will be accessible through, which web page will be presented to the user, which vulnerabilities are likely to be exploited, the parameters that are most likely to contain malicious payloads and many other important details.

Once a request reaches the honeypot server, if no honeypot is configured on the targeted route, an error response (404 Not Found) is returned to the proxy. Afterwards, the proxy returns to the user the original error which was generated by the web application rather than the one generated by the honeypot server itself. In contrast, if the defenders configured a honeypot in the targeted route, the corresponding honeypot web page is returned to the user.

This should lead them to believe that same page was returned by the application itself. By presenting web pages that simulate obvious vulnerabilities, defenders can create honeypots that stimulate attackers into sending malicious requests in attempts to exploit the simulated vulnerabilities. Figure 3.2 shows an example of a simple honeypot log in page which we created to be served in the */admin* route and which should should seem vulnerable to SQL Injection attacks upon some interaction.

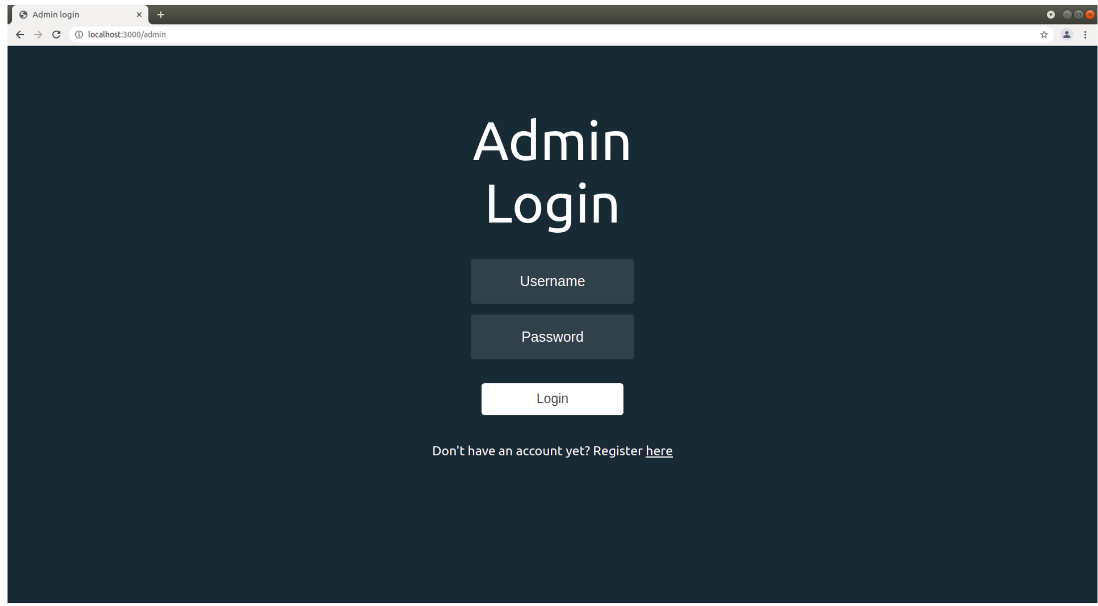


Figure 3.2: Honeypot log in page example

As further requests reach the honeypots, they are then sent to mTANNER. This module searches for malicious payloads within the parameters indicated by the defenders. If a payload is found, a classification process determines the specific vulnerability that is being targeted such as RFI, SQL Injection and others. Afterwards, if the defenders specified that the detected vulnerability should be emulated, the payload is sent to the corresponding emulator, where it is sandboxed and executed in order to determine a viable response.

The output result (string) serves as a prediction of the outcome intended by the attacker. It is sent back to MockingPot and it is placed within the web page which is then returned to the user. As mentioned previously, this creates the illusion that the attack succeeded and that the application executed the malicious payload that was sent, which is often enough to lead the attacker into sending more payloads. In contrast, if no malicious payloads are detected, an error message (which is defined by the defenders in the configuration file) is displayed in the web page instead. Figure 3.3 shows an example of a response where no payloads were detected and Figure 3.4 shows one where a SQL Injection payload was detected. These images are also included in full size in Appendix A (see Figures A.1 and A.2).

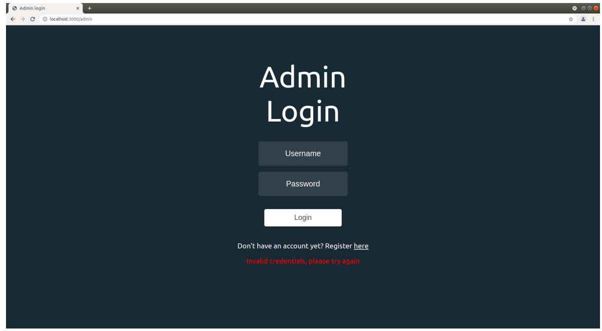


Figure 3.3: Honeypot log in page with an error message

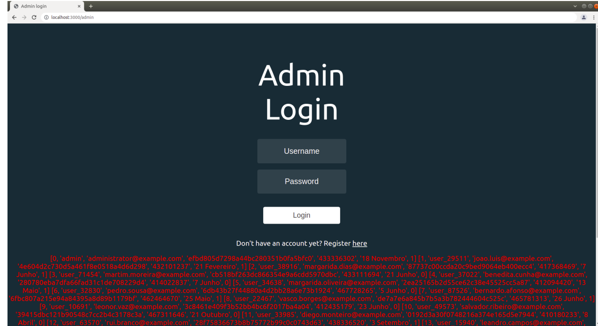


Figure 3.4: Honeypot log in page example with SQL Injection emulation results

Requests detected as malicious by mTANNER are stored and organized into ”sessions” depending on certain details such as IP addresses, user-agents and cookie values. As such, each session should end up containing multiple malicious requests sent by a single attacker. Defenders can use the web interface in order to manually inspect existing sessions at any point in time. Analyzing this data should allow them to discover interesting patterns in attacker activity and to determine effective security countermeasures.

Our solution is easy to set up and compatible with any web application, since only the existing reverse proxies/load balancers need to be modified rather than the web applications themselves. Honeypots can be easily configured in a declarative file and they are automatically integrated into existing web applications since they are presented to attackers as if they were web pages served by the application itself. Defenders can also use our method of camouflaging honeypots so that they blend in with the web pages served by the web application that they want to protect. This should make the resulting honeypots harder to detect visually, as they will be copying the styling used by the actual pages which the application serves. We will explain how the camouflaging process was implemented in Sections 3.5.2 and 3.5.4.

3.4 Reverse proxy / Load balancer

This module can be applied to various reverse proxy / load balancer servers (such as HAProxy and Apache), but the implementation that we will now present is based on a NGINX proxy. NGINX is an open-source and high-performance web server that uses a scalable asynchronous architecture, allowing it to handle thousands of simultaneous requests with better performance and a lower memory footprint than traditional architectures [28]. This means that the impact of our solution on the legitimate requests sent to the original web application will be minimal.

The proxy's behavior is entirely defined in a single configuration file. This file needs to be added to the default NGINX directory *sites-enabled* and configured additionally if defenders see fit (for example, adding certificates to enable HTTPS). If a NGINX reverse proxy / load balancer is already used, then modifying it to run our code should be an easy task. The code block in Listing 3.1 shows a simple example of what that configuration file should look like.

```
1  server {
2      ...
3      recursive_error_pages on;
4      location / {
5          error_page 418 = @testing;
6          if ($http_x_testing_purposes = "someString") {
7              return 418;
8          }
9          error_page 404 = @notfound;
10         proxy_intercept_errors on;
11         proxy_pass http://application$request_uri;
12     }
13     location @testing {
14         proxy_pass http://application$request_uri;
15     }
16     location @notfound {
17         error_page 404 = @recurse;
18         proxy_set_header Cookie sess_uuid=$cookie_SESS_UUID;
19         proxy_set_header X-Real-IP $remote_addr;
20         proxy_intercept_errors on;
21         proxy_pass http://honeypots$request_uri;
22     }
23     location @recurse {
24         proxy_pass http://application$request_uri;
25     }
26 }
```

Listing 3.1: NGINX configuration example

This file defines the four locations (/, @testing, @notfound and @recurse) which implement the proxy logic which was previously explained. First, the / location matches all incoming requests. If the X-Testing-Purposes HTTP header is defined and contains the same value manually defined by the defenders ("someString" in this example), the corresponding request will be sent to the @testing location. Otherwise, the proxy_pass directive is used to forward the request into the web application. The proxy will then wait for the application's response and the error_page directive is used in order to forward all *404 Not Found* error codes to the @notfound location. All responses with other HTTP codes are sent back to the users instead.

The @testing location is used to bypass this logic. In this location, proxy_pass is called to forward requests to the application. Afterwards, all responses (including the ones with *404 Not Found* error code) will be sent back to the users. This feature will allow security testers to freely interact with the web application without having to deal with the deployed honeypots, allowing them to save time and effort.

The @notfound location is responsible for handling all requests that caused the web application to return a *404 Not Found* error. Since these requests could contain sensitive cookies, the proxy_set_header directive is used in order to filter the cookies that are sent to the honeypots, leaving only sess_uuid which is required by mTANNER (which also prevents session hijacking with the data collected by mTANNER). The IP address of the user is also sent through the X-Real-IP header so that it doesn't get lost when multiple proxies are used sequentially. Afterwards, proxy_pass is used to forward the resulting requests into the honeypot server and to wait for a response. The error_page directive is used here in order to forward all *404 Not Found* responses to the final location @recurse. All responses with other HTTP codes should represent successful honeypot interactions and they are sent back to the users.

The final location @recurse handles the requests that did not match any of the routes defined on the honeypot server. The web application's *404 Not Found* errors are fetched once again and sent back to the users rather than the error returned by the honeypot server.

This design allows the proxy component to remain unchanged regardless of the honeypot routes that are chosen. If we were to use a traditional reverse proxy, it would require an updated list of all honeypot routes. This means that defenders would need to re-configure the proxy every time the honeypot routes changed, which would be an undesirable additional step. Furthermore, the performance overhead introduced into existing web applications by this proxy logic is negligible, as shown through our evaluation in Section 4.3.

3.5 MockingPot

MockingPot was implemented with Node.js, which is a popular JavaScript runtime environment used to create web servers with JavaScript code. The hapi.js framework was also used in order to easily configure the honeypot routes securely and without any conflicts.

As mentioned before, the honeypots are defined in a declarative configuration file. Due to its easy readability and intuitive layout, we decided to use the YAML language to implement it. An example of a possible honeypot configuration is shown in Listing 3.2.

```
1  application:
2    protocol: https
3    host: example.com
4
5  analysis:
6    protocol: http
7    host: localhost
8    port: 8090
9
10 honeypots:
11   - category: template
12     route:  '/admin'
13     emulationParameters: { sqli: ['username'] }
14     emulationMethods: ['post']
15     template: 'auto_templates/login1.html'
16     error_post: 'Invalid credentials, please try again'
17     modifier:
18       sqli_blind: true
19     provider: 'Example'
20     basePage: 'https://example.com/login'
21   ...
```

Listing 3.2: Honeypot configuration example

The configuration file is composed by three blocks:

- *Application block:* specifies the host, port, route and protocol used in order to access the web application which will be associated with MockingPot. This information is used in order to retrieve useful information about the application's behavior, such as its response headers. These headers will then be included in the honeypots' responses later on in order to mimic the application's behavior as closely as possible and with negligible performance penalty. By returning the same headers as legitimate application routes, the honeypots will be harder to detect;
- *Analysis block:* specifies the host, port and protocol used in order to access mTANNER. This information is required for these two modules to communicate, consequently allowing honeypot responses to be emulated and incoming data to be stored;
- *Honeypots block:* this is the block where the parameters of the honeypots are specified and, as such, it will be the lengthiest of the three blocks.

The first parameter that must be specified about a honeypot is its `category`. This module can deploy three categories of honeypots (custom, template and text) which differ in their behaviors and in how they are specified. All categories of honeypots require the following parameters:

- the `route` in which they should be deployed;

- an object containing the vulnerabilities that should be emulated and the list of parameters where payloads should be sent (emulationParameters);
- the list of HTTP methods where emulation is allowed (emulationMethods);
- the error message that should be presented to the users if the emulation process failed (if no malicious payloads are found or if no valid responses are generated).

Defenders can use modifiers which alter the behavior of the honeypots. One of those modifiers is sqli_blind which simulates a blind SQL Injection vulnerability by only showing SQL-related errors to the attackers and omits the success cases. This modifier as well as others will be introduced throughout this chapter.

There are also some other parameters that are required depending on the chosen category. For example, template honeypots require the template parameter and, optionally, provider and basePage which we included in the configuration example above. We will explain the differences between the three categories of honeypots as well as their exclusive parameters in the next three sub-sections.

3.5.1 Custom honeypots

This category's main particularity is that it requires the path to a valid HTML file (page parameter) which will be presented to the users who target the honeypot. That file only requires frontend logic, since all backend connections (to mTANNER) will be automatically configured. The honeypot shown in Figures 3.2, 3.3 and 3.4 was created using this category with the following configuration (Listing 3.3):

```
1 honeypots:
2   - category: custom
3     route: '/admin'
4     emulationParameters: { sqli: ['username'] }
5     emulationMethods: ['post']
6     error_post: 'Invalid credentials, please try again'
7     page: 'admin.html'
8     injectionId: 'errorMessage'
```

Listing 3.3: Custom honeypot configuration example

Defenders must specify an injectionId, which corresponds to the id of the HTML element in which the emulated responses (or the error message specified in error_post) should be inserted and returned to the users. If no response is emulated then that HTML element is removed from the web page.

The simulated vulnerabilities will have a heavy impact on the attention that the honeypots attract. As such, the honeypots' effectiveness will depend on the effort and creativity that the defenders use in order to construct the pages that they present.

Furthermore, these honeypot pages will be most believable and hard to detect when their style resembles that of the actual pages served by the associated web application. For this reason, we

decided to develop a method of camouflaging honeypots so that they automatically blend in with those web pages, which is a feature of the template honeypots that we will introduce next.

3.5.2 Template honeypots

This category generates web pages which can automatically mimic the style of other ones served by the application, as we mentioned in the previous section. To create a template honeypot, defenders must first specify a template (path to a template file). These templates are basically HTML pages containing Handlebars [29] expressions (defined with four curly braces). These expressions can be used for multiple purposes, one of which is to insert variables into those pages at any point in time. For instance, <p>fffoogg</p> is a valid Handlebars template which, when rendered with a variable foo="bar", generates a paragraph HTML element containing the string "bar". Listing 3.4 shows an example of how these templates can be used in order to dynamically generate and stylize the honeypot web pages.

```
<head>
    <title> {{provider}} </title>
    <link rel="icon" href="{{icon}}">
    <style>
        body { background: {{background}}; }
        p {
            color: {{color}};
            font-family: {{font}};
        }
    </style>
    ...
</head>
<body>
    <img src="{{logo}}">
    <p> Welcome to the {{provider}} page! </p>
    ...
</body>
```

Listing 3.4: Handlebars template file example

The following variables are recognized by this module within the templates:

- provider: the entity that (hypothetically) serves the honeypot page. Modern web pages usually mention in multiple locations the entity responsible for serving those pages. For example, a Facebook web page will have multiple occurrences of the word "Facebook", as it allows users to quickly know what site they are currently visiting;
- icon: the URL to a shortcut icon/favicon;
- background: the CSS property that defines the honeypot page's background;

• `color`: the predominant text color (CSS property) to be used in the page;

• `font`: the predominant text font-family (CSS property) to be used in the page;

• `logo`: the URL to a logotype to be placed somewhere in the page.

We will further explain how each variable is determined in Section 3.5.4. Each of these variables can be manually defined by the defenders within the configuration file. However, the main appeal of this category is that the values for the last five variables can be automatically extracted from existing web pages in order to mimic their style, as we will discuss in the next paragraphs.

Additionally, this module will also recognize certain Handlebars conditionals [30] which can be used in order to only invoke certain sections of the template if a condition is met. For example, the code block shown in Listing 3.5 will introduce a placeholder image into the final web page if the `logo` variable is not defined (otherwise, it will introduce that variable's value).

```
1 {{#ifDefined logo}}
2     <img src="{{logo}}">
3 {{/ifDefined}}
4 {{#ifNotDefined logo}}
5     <img src="https://via.placeholder.com/150">
6 {{/ifNotDefined}}
```

Listing 3.5: Example of a recognized Handlebars condition

Conditionals such as this can be used to set default values for variables that could not be automatically extracted and that were not manually specified. This contributes to the generated honeypot pages being more consistent, since missing variables will still result in explicitly defined behaviors. The conditionals that are registered by default are the following (where `var1` and `var2` are two of the variables which we mentioned previously):

• `ifDefined var1` - executes block if `var1` is different from *undefined*;

• `ifNotDefined var1` - executes block if `var1` is equal to *undefined*;

• `ifEquals var1 var2` - executes block if `var1` is equal to `var2`;

• `ifNotEquals var1 var2` - executes block if `var1` is different from `var2`.

If a `basePage` (URL to an existing web page) is specified in the configuration file, this module will scrape that page in an attempt to retrieve values for the previously mentioned variables. The method used to extract these values will be discussed in detail in Section 3.5.4. After the extraction, those variables will then be used in order to generate the desired honeypot web page. Since the variables use values extracted from the `basePage`, it is likely that the style of the resulting honeypot page will end up resembling its own. Therefore, defenders can use this feature to create believable honeypot pages

which mimic actual pages served by the application. If for some reason one of the extracted variables does not satisfy the defenders, they can also be individually overwritten in the configuration file.

There are many ways to introduce these Handlebars expressions into the templates, with some being more discreet than others. For example, the `logo` variable can be placed within the `src` attribute of a HTML image element (as shown in the previous code block) or it can be placed within the `background-image` attribute of another HTML element. Many scenarios can be used to achieve the same outcome, and it is entirely up to the defenders to decide how they want to insert the variables into the resulting honeypot pages.

Figure 3.5 shows an example of a page generated with a template honeypot (using a configuration similar to the one in Listing 3.2) where the `basePage` was not defined and, consequently, the default behaviors were used. Even though the page is styled consistently, it is very generic since there are no indicators of the specific domain which is being visited: the favicon is not specified; the background simply uses a pre-defined color; all the text is styled using a default color and font; a placeholder image was used to fill in for the missing logo; and there is no mention anywhere of the site provider.

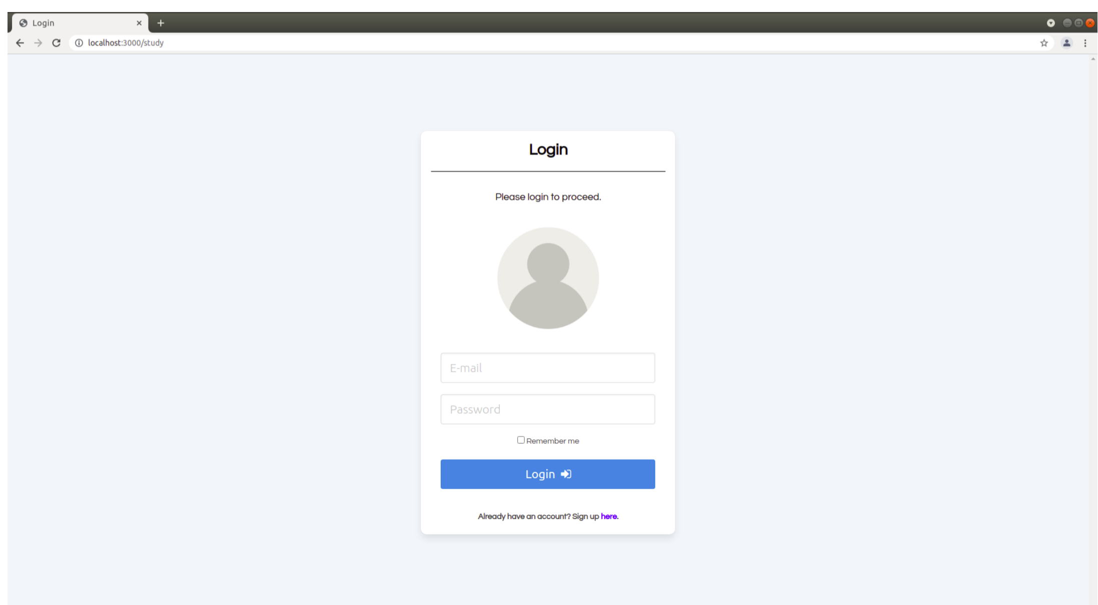


Figure 3.5: Template honeypot page without any variables defined

Figures 3.6 and 3.8 show two honeypot pages (both generated using the previous template) where all the variables were automatically extracted from the Instituto Superior Técnico (IST) and Universidade de Lisboa (ULisboa) log in pages, respectively. Depending on the chosen *basePage*, a single template can achieve many different styles. In both pages, the extracted features that are most noticeable are the background image and the logotype. However they also have different favicons, text colors and fonts, all of which contribute to how believable the end result is. Furthermore, the fact that ”IST” and ”ULisboa” are repeated throughout the pages using the `provider` variable plays a major role in the consistency of the web page. Figures 3.7 and 3.9 show the *basePages* that were used to automatically stylize each of the honeypot pages. Figures 3.6 through 3.9 are also included in full size in Appendix A (see Figures A.3 through A.6).

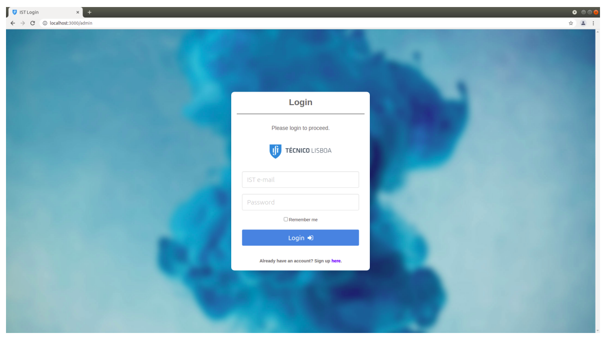


Figure 3.6: Template honeypot (IST)

Figure 3.7: IST log in basePage

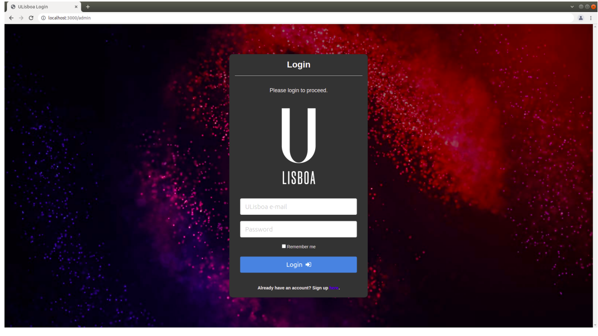


Figure 3.8: Template honeypot (ULisboa)[1]

Figure 3.9: ULisboa log in basePage

Since a single template can generate fairly different pages for different web applications, templates can even be shared amongst defenders. This way defenders don't even need to create the templates themselves, as they can use existing templates that others created in order to simulate certain vulnerabilities. This allows defenders that lack creativity to generate powerful honeypots in a fairly effortless manner. However, using shared templates will naturally increase the risk of attackers discovering that they are interacting with honeypots due to their increased public exposure.

In order to evaluate the effectiveness of this page camouflaging process we conducted a study with over 80 participants. We describe it and present its results in Section 4.1.

3.5.3 Text honeypots

This is the simplest category of honeypots that can be deployed with MockingPot. These honeypots do not require any additional parameters and they do not present the users with a web page. Instead, payloads are only sent via URL or body parameters. Afterwards, the results of the emulations are sent back to the users in plain text.

Even though it is quite simple, this category can still generate powerful and versatile honeypots. Listing 3.6 shows an example of an effective honeypot configuration that can be deployed with this category: a seemingly vulnerable PHP script.

[1] Color of the box containing the log in form was automatically darkened with Javascript in order to make the white letters visible

```
1  honeypots:
2    - category: text
3      route: '/getpage.php'
4      error_get: " $_GET['file'] not found"
5      emulationMethods: ['get']
6      emulationParameters: { 'lfi': ['file']}
```

Listing 3.6: Text honeypot con guration example

According to this con guration, a text honeypot will be deployed on the /getpage.php route. As mentioned previously, the error _get parameter contains the message that will be presented to the users if no response is emulated. Since emulationMethods is only set to 'get', this route won't be accessible via POST requests, which matches the behavior implied in the error message. The other parameter indicates that mTANNER should only look for LFI payloads in a ' le' URL parameter.

From the point of view of the users (Figure 3.10), this page will seem like a simple PHP script that is being executed and returning either an error or a le that exists on some remote server. After recognizing this behavior, some attackers will be tempted to verify if the script is vulnerable to LFI attacks. If an attacker then decides to send a LFI payload within the de ned URL parameter, mTANNER will attempt to emulate the content of that le and present it back to them (Figure 3.11). This is done by sending LFI payloads to fresh isolated Docker containers running a Linux-based operating system (BusyBox [31]) inside of a cat command. The result of this command's execution should then re ect the responses intended by the attackers without actually accessing any les in the host.

Figure 3.10: Text honeypot simulating PHP script when rst visited

Figure 3.11: Text honeypot simulating PHP script when a LFI payload is sent

In this scenario, the attackers will be lead to believe that their attacks have succeeded, and so they will be tempted to send more payloads to the server in hopes of nding sensitive les. However there is no actual way for them to access any important les, and they will only end up wasting their time and yielding their behavioral data to the defenders.

Now that we have presented all three categories of honeypots, it is important to note that they do not necessarily need to be isolated from each other. In fact, different categories should be combined in order to achieve maximum effectiveness. For example, a custom honeypot page can have a hyperlink to a template honeypot that emulates a LFI vulnerability. At the same time, it can also send requests on form submission to a route containing a text honeypot which emulates a PHP script vulnerable to SQL Injection attacks. The more complex the deployed honeypots, the harder they will be to detect and the more attractive they will become.

3.5.4 Page camou aging

Finally, we will discuss how the variable extraction process works and how their values are determined. First, a request is sent to the URL speci ed in the basePage con guration le parameter in order to obtain the target web page. We decided to use puppeteer in order to do this, rather than sending a simple request and analysing the response manually. Puppeteer is a NodeJS library which allows us to control instances of headless Chromium. By using an actual browser, we have access to a lot of useful information and features which are essential for determining the layout of the basePage For instance, we have access to Chromium's Document Object Module (DOM), which allows us to quickly determine all HTML elements affected by a certain CSS rule (using the Window interface's getComputedStyle method) or containing a certain HTML attribute. Furthermore, any scripts located in the web page will be executed before giving us access to the resulting web page. This means that any visual changes conducted via scripts will be re ected in the web page, and so we will have access to the nal web page as it is expected by users.

Since variables are only extracted before the honeypots are established, there is no need for this process to have a performance constraint. As such, we conduct an extensive search (within the target web page) for certain indicators which allow us to determine the layout of the basePage We will now brie y explain how the value for each variable is determined.

Icon This is the easiest variable to nd since the favicon is de ned in speci c locations: we simply have to look for the HTML tags containing the rel attribute with value icon or shortcut icon, and then we extract the URLs in the href attribute. If no HTML tags meet this criteria, then the icon will most likely be located in the default location (the /favicon.ico route). In order to con rm this, an additional request is sent to that location and, if a response with status code 200 is returned, that URL is extracted;

Background We use a two-phase process in order to nd the value for this variable. In the rst phase, we attempt to look for a background image. To do this, we start by looking for HTML elements with an id, class or name that matches a variation of the word ”background” (such as ”backg” and ”bgnd”, in

a speci c order according to their likelihood). If no matches are found, the body element will be used instead. Then, we loop through those elements and we use getComputedStyle to obtain the values de ned for their background-image CSS property. If a valid image is returned, we must verify if it is big enough to cover the background. In order to do so, we obtain the area of the corresponding HTML element using its width and height CSS properties and, if it occupies at least a certain percentage (by default it's 70%) of the entire viewport area, we extract the value of its background property. This property combines background-image with other properties that affect it such as background-position and background-size, thus making the extracted result more complete.

If this process is unsuccessful, we look for all HTML elements whose background-image value (usually a URL) contains the same variations of the word ”background” that were used earlier. This additional step is very effective since it is quite usual for the URLs of the background images to contain one of those keywords. If an element meets this condition as well as the size requirements, the value of its background property is extracted.

When no background images are found on the target page, the second phase commences: searching for an appropriate background color. The process is identical to the rst step of the previous phase but it analyses the background-color property rather than background-image;

Font and color Both of these variables are extracted simultaneously using this process. Most web pages predominantly use a single HTML element (or a small set of them) to display text. Therefore, we rst organize the HTML elements that usually contain text (for example, <a> and <p>) from the most occurrences to the least. Afterwards, we loop through the elements in that order and we use getComputedStyle to obtain the values de ned for their font-family and color CSS properties. Once we nd an element where both of those properties are de ned, we extract the corresponding values;

Logo In order to nd a value for this variable, we start by looking for HTML elements with an id, class, name or alt that matches a variation of the word ”logotype” (such as ”logo”). Then, we loop through those elements and we extract either the values of their src attribute or background-image CSS property.

If no matches are found with this step, we look for all images whose src URL contains the same variations of the word ”logotype” that were used earlier (and we extract them). This step is very effective since it is quite usual for the URLs of logotype images to contain one of these keywords. Finally, if nothing has been found until this point, we repeat the last step but we target the background-image CSS property of all HTML elements rather than the src attributes of all images.

During all of these extraction processes, we begin by exploring the most common scenarios (such as some Search Engine Optimization[2] conventions [32]). If no values are found, we then gradually transition into less probable scenarios. Even though we attempt to make this search thorough, we are also careful not to make it too elaborate in order to prevent incoherent results. However, as there is no uni ed and awless strategy for obtaining such values from a web page, there will always be some basePages that yield incoherent results due to them using unconventional structures or styling strategies.

[2]Set of strategies with the objective of positioning web pages among the top results yielded by search engines

3.6 mTANNER

As the name suggests, we modi ed the TANNER tool that we presented in Section 2.5 in order to implement this module since it already implemented some of the functionality that we desired. By using TANNER, our solution bene ts from both its attacker identity recognition and session creation functionalities. However, we extended it in order to mitigate some of the main issues that we mentioned previously and to increase its general effectiveness.

In the next subsections we will discuss those modi cations and their consequences. We will also introduce some new modi ers which can be used in MockingPot's con guration le in order to alter how the responses produced by mTANNER are shown to the attackers. To avoid showing an example of a con guration le for each scenario, we will list all of the parameters and their possible values in Appendix C (see Listing C.1).

3.6.1 Limiting the emulation process

TANNER originally searches for payloads within all of the parameters de ned in the incoming requests. For GET requests, those parameters are extracted from the URL and for POST requests they are extracted from the body instead. However, this leads to inconsistencies that could allow attackers to quickly grow suspicious of the deployed honeypots, as exempli ed in Figure 3.12.

Figure 3.12: Text honeypot simulating PHP script using the original TANNER tool

This honeypot uses the same con guration that was shown in Listing 3.6, but it is using the original TANNER tool rather than our modi ed version. Regardlessly of the parameter speci ed in the URL (in this case it's foobar), the payload will always be detected and used in the emulation process. Even though this makes it more likely for malicious payloads to be found, it can heavily impact how believable the honeypot is. As mentioned previously, this page is meant to mimic a simple PHP script that parses the value in the le parameter and returns a le in a remote server in the corresponding location (if it exists). However, if any parameter can be used to achieve the same outcome, this will obviously seem inconsistent from the point of view of the attackers as the expected behavior will differ from the observed one. This is made especially obvious through the use of automatic vulnerability scanning tools. In ordinary web applications, these tools will often nd a small amount of vulnerabilities (if any). However, a honeypot associated with the original TANNER tool will ag an unbelievable amount of vulnerabilities

due to any payload sent to any route being used for the emulation process. After verifying these results, any experienced attacker will immediately grow suspicious of the honeypot and will likely avoid it entirely.

By limiting the parameters where this module should search for malicious payloads, the resulting honeypots should look more consistent and believable from the point of view of the attackers. The honeypot in Figure 3.13 is similar to the previous one, but it uses our mTANNER instead of the original TANNER tool. In the con guration le (see Listing 3.6), we de ned emulationParameters as f 'l ': [' le'] g. This object is sent to mTANNER along with each request, and only the parameters speci ed in it will be inspected for malicious payloads. In this case, payloads are only searched for in the le URL parameter, which matches the behavior expected by the attackers. Therefore, even though the foobar parameter contains a valid LFI payload, it is ignored by this module and an error is simply returned. As such, the results produced by automatic vulnerability scanners will also resemble those yielded by an ordinary vulnerable web application.

Figure 3.13: Text honeypot simulating PHP script using mTANNER

The emulationParameters object also de nes which vulnerabilities mTANNER should be emulating. This allows us to avoid scenarios such as SQL Injection vulnerabilities being emulated in pages that should only contain LFI vulnerabilities. In contrast, the original TANNER tool attempts to emulate every available type of vulnerability simultaneously, often causing the observed behavior to differ from the one expected by attackers and resulting in scenarios such as the one shown in Figure 3.14.

Figure 3.14: Text honeypot simulating PHP script incorrectly emulating SQL Injection

It is also important to note that the requests are ltered by the targeted routes and by the used HTTP methods before being sent to this module. The original TANNER tool is meant to be used with the SNARE tool that we also presented previously. This combination allows payloads to be emulated in any route independently of the HTTP method, thus making the resulting honeypots inconsistent and detectable. In contrast, with MockingPot defenders must preemptively de ne the routes where they want

honeypots to be deployed. Furthermore, defenders can also de ne the HTTP methods that should be allowed for each route, which can add an extra level of consistency to the honeypots. For example, it wouldn't make sense for the honeypot in Figure 3.13 to be accessible via POST requests when it speci cally states that the le parameter is obtained through the PHP $_GETarray [33].

All of these limitations that we imposed in the emulation process will naturally make it harder for attackers to nd the simulated vulnerabilities, since some payloads that would be recognized by the original TANNER will now be ignored instead. However, we believe it is essential to guarantee that the honeypots are as consistent and believable as possible, which should contribute towards maximizing their time-wasting and data-capturing capabilities.

3.6.2 Additional dummy database customization

As mentioned previously, TANNER emulates SQL Injection attacks by injecting the detected payload into a vulnerable SQL query, which is then executed in a dummy database. In order to generate and populate that same database, defenders must edit a con guration le named db_con g.json . This is where the actual tables are speci ed, along with the column names and data types. Furthermore, defenders must also associate each column with a data token, which represents a speci c type of dummy data (generated with the mimesis [34] python library) to be inserted into a certain column. There are ve data tokens that are natively recognized by TANNER:

- I - integer: number that starts in 0 and increases by 1 with each row;
- L - log in: username composed by a string, an optional separator and a four-digit number;
- E - e-mail: address composed by a string followed by a four-digit number and a random popular e-mail provider (@gmail.com, @yahoo.com, @outlook.com, etc.);
- P - password: xed-length (8 char) string of random characters (including symbols);
- T - piece of text: sequence of random sentences.

Our issue with this method is that each resulting row in the dummy databases often contains many inconsistencies, some of which we will now enumerate:

- Usernames and e-mail addresses contain a string and a four-digit number, but the two strings and numbers are normally completely unrelated to each other, which is a huge giveaway that the data is fake;
- Usernames might not comply with contextual naming conventions and there is no native way to de ne speci c templates/masks;
- E-mail addresses might not comply with contextual naming conventions and there is no native way to choose the desired e-mail providers;
- The generated passwords are too complicated to be realistic;

- Data is always generated in English and there is no native way to specify another language, which can be unrealistic in non-English applications.

In order to mitigate these inconsistencies we added a few options to the con g.yaml le, which the defenders can edit in order to maximize the consistency of their dummy databases. We added the locale option, which indicates the locale code (pt, es, fr, etc.) of the language in which the dummy data should be generated; the domains option which should contain an array of the e-mail providers to be used while generating e-mail addresses; the username_format option which speci es a speci c template/mask to be followed while generating usernames; and the rst_names and last_names options which speci es the location of a le containing rst and last names to be used in the e-mail addresses.

We also made some changes to the existing data tokens. All passwords are now hashed by default, thus hiding the inconsistencies of the original password values. Furthermore we hope that, upon seeing a hashed password, some attackers will attempt to decode them using rainbow tables. In order to fuel this temptation, we even implemented the introduction of an ”administrator” row into each table. We believe that this strategy alone can successfully attract a lot of attention from attackers and waste a lot of their time.

Furthermore, we thought that these ve tokens were not enough to populate complex databases. As such, we implemented eight more native tokens to be used in db_con g.json : PN, random phone numbers or numbers based on the templates/masks speci ed in the new phone_masks option; CC, random credit card numbers; A, an address from the locale de ned by the defenders; D, a date containing a day and a month; AD, an academic degree; CS, an international clothing size; U, a UUID4 which can be used for multiple purposes; and B, a simple boolean value.

We will now show a comparison between a database generated with the original TANNER and another one generated with our mTANNER, in order to show how these modi cations can affect the SQL Injection emulation process. Listing 3.7 shows the contents of a db_con g.json and the corresponding database dump produced by a SQL Injection emulation with the original TANNER.

```
1  {
2    "table_name": "users",
3    "schema": "CREATE TABLE users (id INTEGER PRIMARY KEY, username text,
4       email text, password text);",
5    "data_tokens": "I,L,E,P"
6  }
7
8  [0, 'batara.1970', 'bagels1976@outlook.com', 'gP1N@e#C']
9  [1, 'DearaujoNone.2047', 'fairish1917@gmail.com', '7AHd{@{n']
10 [2, 'Tonic-1940', 'anterograde1938@live.com', 'YZ_R"UZ|']
11 ...
```

Listing 3.7: Database con guration and dump produced using TANNER

This is the default database that comes con gured with TANNER. It uses four out of the ve original data tokens in order to generate a simple dummy table that stores user information. However, some of the inconsistencies that we previously mentioned are quite noticeable: the usernames and e-mail addresses are generic and completely unrelated, and the passwords are too complicated to be believable. Next, Listing 3.8 shows another con guration that uses our new tokens and the corresponding database dump produced using mTANNER.

```
{
  "table_name": "users",
  "schema": "CREATE TABLE users (id INTEGER PRIMARY KEY, username text,
    email text, password text, phone text, join_date text, teaching bool);",
  "data_tokens": "I,L,E,P,PN,D,B"
}

locale: pt
domains: ['@universidade.pt']
username_format: 'estudante_%I%I%I%I%I'
first_names: '/opt/tanner/tanner/data/pt_firstnames.txt'
phone_masks: ['41#######', '43#######', '46#######']

[0, 'admin', 'administrator@universidade.pt',
'efbd805d7298a44bc280351b0fa5bfc0', '414502312', '18 Junho', 0]
[1, 'estudante_21492', 'constanca.dias@universidade.pt',
'c1c8bd888c6b7a7dc6f7a448daded787', '430541258', '1 Julho', 0]
[2, 'estudante_29138', 'ruben.barbosa@universidade.pt',
'649a78eba0dc25e529014918e1098538', '438640103', '26 Novembro', 0]
...
```

Listing 3.8: Database con guration and dump produced using mTANNER

We customized this database to make it seem like it holds user information related to a Portuguese university, which is made possible by using the customization options that we implemented. We also used some of the additional data tokens that we created in order to increase the complexity of the database in comparison to the one that we showed previously. As you can see, the inconsistencies in the previous database are not noticeable in this one: all the usernames and e-mails follow speci c formats that are contextually appropriate and the passwords are hashed. The administrator account should also attract the interest of some attackers as we mentioned previously, and the additional columns that we populated with the new data tokens should make the dummy database more believable.

Additionally to these customization options, defenders can also use the sqli_blind modi er in the honeypot con guration le in order to emulate a "blind" SQL Injection vulnerability. This means that, rather than receiving database dumps in response to successful malicious payloads, attackers will instead only receive feedback from the server in the form of errors, such as syntactic and semantic errors in their

payloads. Therefore, they will still be able to access the database dumps but it will require more effort on their part (for example, by sending payloads letter-by-letter and using the triggered errors in order to determine the entirety of the database's contents).

3.6.3 Serving les in LFI honeypots

As we have mentioned previously, TANNER uses isolated Docker containers running BusyBox in order to emulate the LFI vulnerability. When a LFI payload is detected using a speci c regex expression, a cat command followed by that payload is sent to a container in order to retrieve the contents of the targeted le (if it exists within that same container). As such, the result of this command should re ect the responses intended by attackers. An example of a text honeypot emulating this vulnerability was already shown in Figures 3.10 and 3.11.

Our main issue with this emulator is that there was no easy way to add external les to the containers in which the cat command is run. This means that the only les which could be presented to attackers were the default les that came with the BusyBox image. Therefore, we decided to add a feature that allows defenders to easily choose local les to be served by these honeypot routes in speci c directories. This new feature is accessible through the lfi_files modi er in the honeypot con guration. For example, let us consider the scenario in Listing 3.9 which is similar to the one in Listing 3.6 but also takes advantage of this feature:

```
1 honeypots:
2   - category: text
3     route: '/getpage.php'
4     error_get: " $_GET['file'] not found"
5     emulationMethods: ['get']
6     emulationParameters: { 'lfi': ['file']}
7     modifiers:
8       lfi_files: true
```

Listing 3.9: Text honeypot con guration using the lfi_files modi er

All les that defenders want to use in LFI honeypots must be located in a speci c folder used by MockingPot (which is also named l _ les). By setting this modi er to true, all the les in that same directory will now also be accessible through the LFI honeypot. For example, if that directory contains a le named img.jpg and attackers send a GET request to /getpage.php? le=img.jpg , they will be presented with the contents of that le. The advantage of this feature is that, if attackers nd img.jpg, they should be further lead to believe that they are interacting with a real application rather than a honeypot since they witnessed an example of its "correct" usage. Furthermore, les served by LFI honeypots can be included in other honeypot pages in order to further increase the likelihood of attackers nding them.

3.6.4 Isolating Docker-based emulation

TANNER can be easily run as a multi-container Docker application using the Compose tool. The docker-compose.yml file defines a set of services to be run in different containers and a network that allows those containers to communicate with each other. Originally, TANNER is set to be run in five different containers: tanner_redis which sets up and maintains the redis server that will be used to store the data collected using the honeypots; tanner_phpox which runs the PHP sandbox that is used to emulate PHP-related vulnerabilities such as PHP Code Injection and RFI; tanner_api which runs code that allows it to access the redis server and return the stored data according to different parameters and filters (for example, return all sessions corresponding to a specific IP address or return the latest sessions); tanner_web which serves the web interface that defenders can use to inspect the collected data in an organized manner (using the previous API to retrieve that data); and tanner which handles the remaining tasks such as inspecting incoming requests for malicious payloads, executing emulators corresponding to other vulnerabilities like SQL Injection and LFI, and sending data to the redis server.

Considering the tasks that each container executes, the tanner container will usually have the heaviest workload. Furthermore, since it is responsible for running multiple vulnerability emulators, this container should also be the one in the most danger of being taken over by an attacker. If this were to happen the infiltrator would be able to access the other containers and, for example, tamper with the redis server by flooding it with fake data. This risk (although unlikely) is to be expected and there is no way around it other than occasionally backing up the redis server and destroying the containers.

However, there is a bigger risk associated with this infiltration scenario. The tanner container is responsible for performing Docker-based emulation for vulnerabilities such as LFI and Command Execution (CE). This type of emulation creates new isolated Docker containers using aiodocker [35] in order to execute malicious payloads with the objective of obtaining plausible responses (such as the contents of a certain file within that container or the result of running a certain command). In order to create and interact with those new containers, the tanner container requires access to the Docker socket, which can be done by mounting it as a volume. In this situation, if an attacker managed to gain access to the tanner container, they would also be able to access the Docker socket on the host machine. Consequently, they would be able to gain root access privileges on the host and to manipulate containers as they please.

Considering that the tanner container has the highest workload and constantly handles malicious payloads, giving it access to the Docker socket is a solution that should be avoided if possible. As such, we modified TANNER so that it would run with a sixth container called tanner_aiodocker, which would be the only one able to access the Docker socket. This is our lengthiest modification but it is mostly security-oriented, so it does not produce any major results functionality-wise.

Originally, the Docker-based emulators could simply instantiate an object of the AIODockerHelper class and directly call functions that access the Docker socket in order to perform operations such as creating/deleting containers and executing commands on them. However, since we wanted to avoid giving the tanner container access to that socket, we migrated that class (and the volume containing the socket) to tanner_aiodocker. In order to make the previous operations available to the emulators, we coded an API server that allows them to be executed via HTTP requests. This server will run in this

sixth container, which will be in the same network as the tanner container. Therefore, the Docker-based emulators will still function without having direct access to the socket.

It is important to note that, if an infiltration scenario like the one that we mentioned earlier does happen, the infiltrator will still be able to access some Docker operations via HTTP requests. However, at least they won't have full access to the socket and they will be limited to the operations made available through the API server. As such, we limited the number of available operations and made them as impactless as possible. Furthermore, tanner_aiodocker should be much harder to infiltrate, since all it does is run a simple server that sends the payloads to other isolated containers.

3.6.5 Other modifications

We also introduced some other smaller features and modifications, which we will now briefly enumerate since they do not justify entire subsections of their own:

- Disabled emulation of XSS payloads since the process could introduce actual XSS vulnerabilities into exising web applications;
- Made visual and logical improvements to the web interface, such as displaying the body payloads on POST requests when inspecting a session;
- Updated multiple patterns to detect malicious payloads more reliably;
- Payloads in POST requests without parameters will now be correctly parsed rather than triggering an internal error;
- Fixed internal errors thrown by multiple emulators for certain payloads (for example, SQL injection emulator threw an undesired exception when attackers tried to run multiple statements);
- Fixed certain emulators (XML External Entity (XXE) and PHP Object Injection) not working due to missing parameters in their handle methods;
- Blocked some of the PHP functions that could be executed by the PHP-related emulators, such as phpinfo() which revealed more information than we desired;
- Implemented periodic snapshots of the redis database (where the collected data is stored), which will now appear in the dumps folder.

Chapter 4

Results

In order to evaluate the effectiveness of our solution we performed three different studies, each targeting a specific part which we wanted to empirically verify. Section 4.1 presents the results of our evaluation regarding the effectiveness of the page camouflaging technique that we developed. Section 4.2 describes the study that we conducted to understand how our modifications to mTANNER affected the data collected by the resulting honeypots. Finally, in Section 4.3 we perform a benchmark in order to measure the overhead introduced by our solution into web applications.

4.1 Page camouflaging study

In order to verify if our templates could blend in with existing web sites using the automatic process that we described in Section 3.5.4, we created two questionnaires using Google Forms: one meant for the experimental group of participants and another one for the control group. Both questionnaires had the same types of questions, but there were some differences in the answers shown to the participants.

In both questionnaires, participants were shown screenshots of web pages. Some of those pages were taken from fairly popular web sites (”real”) whereas some others were automatically generated using our templates (”fake”). We briefly explained the purpose of the study as well as the two types of questions that the participants would be asked: in the first half they would be shown individual screenshots which they had to identify as “real” or ”fake”; in the second half they would be shown two screenshots, one ”real” and one ”fake”, and they had to identify the ”real” page. Note that, in each of these questions, we also mentioned the supposed domain of the pages in the screenshots so that they would know what visual clues to search for. For instance, if participants were asked Is the following a real or fake Zoom page? they would understand that they should look for Zoom-related attributes (logos, keywords, etc.).

The only difference between the experimental and control group questionnaires were the screenshots used as the ”fake” pages. In the control group we used the base form of our templates (without being subjected to the variable extraction process), and in the experimental group we used those same templates but attempted to mimic a ”real” page belonging to the domain mentioned in the question (as shown in Figures 4.1 and 4.2, Figures A.7 and A.8 for the full size). Therefore, we expected most of the participants from the control group to be able to identify the ”fake” pages fairly easily, which should imply

that the questions containing them would be correctly answered almost 100% of the time. In contrast, assuming the templates could correctly mimic the domains, we expected the participants from the experimental group to not be able to distinguish the ”fake” pages from the ”real” ones. As such they would ideally only answer the questions correctly around 50% of the time, which would be similar to choosing randomly, thus proving that the participants were unable to distinguish the pages shown to them.

Figure 4.1: Question from the control form

Figure 4.2: Question from the experimental form[1]

We also collected data regarding the web browsing literacy (a number from ”1” to ”5”) of the participants. This was just meant to give us an insight to how knowledgeable they were, which should directly correlate to how reliable the collected data is. Ideally the median of these values would be close to 5 since this would be the appropriate value for most of the attackers that are expected to interact with the template honeypots. Furthermore, it would also be ideal for both of the questionnaires to have a similar value distribution of answers to this question, which would mean that the experimental and control group were similar in terms of aptitude.

4.1.1 Results

We managed to get results from over 100 participants in total (in both of the questionnaires). As we mentioned previously, we wanted the median of the web browsing literacy answers to be close to 5. Therefore, we decided to exclude all of the participants who responded with less than ”4”. We then ended up with 80 participants in total, 40 in the experimental group and 40 in the control group. The percentage of correct answers from both groups is shown on the graphs in Figure 4.3.

From the 40 participants in the control group, 13 of them answered with ”4” and 27 with ”5”, making the median of this value 4.675. The total percentage of correct answers across all of the questions was 98.13%, which corresponds to the scenario that we had previously envisioned. All of the questions with two screenshots were answered correctly 100% of the time since the differences between the base templates that we used and the ”real” pages were too obvious, while the single screenshot questions were correctly answered 95-100% of the time.

[1] Fake web page created in order to mimic the Roblox log in page for research purposes and used only in this private study

Figure 4.3: Percentage of correct answers in both questionnaires

From the 40 participants in the experimental group, 14 of them answered with ”4” and 26 with ”5”, making the median of this value 4.65. The total percentage of correct answers across all of the questions was 50,31%. In most questions participants were correct 40-60% of the time but there were two distinct outliers. The rst outlier was the Roblox question shown in Figure 4.2, which only 30% of the participants managed to answer correctly. The combination of the template with the extracted attributes shown in that screenshot was the most realistic in the entire questionnaire and it seems that a big majority of participants believed that it was ”real”.

In contrast, the second outlier was a question with two screenshots which 80% of participants answered correctly. This probably happened due to the high popularity of the screenshot that we used, which contained the Microsoft log in page. We also got feedback from some of the participants saying that they had used the page recently, which is why they knew how to answer correctly.

Since the ”fake” pages were so easy to spot in the control group form, this also affected the single screenshot questions with ”real” pages shared by both questionnaires. During the control group questionnaire, the participants must have learned that all pages with domain-speci c attributes (logos, keywords, etc.) were ”real”, which does not happen in the experimental group questionnaire since all screenshots contain these attributes. This explains why the question showing a ”real” Walmart page in both questionnaires was answered correctly by 97.5% of the control group but was only answered correctly by 50% of the experimental group (as shown in Figure 4.3 on the graph on the left).

In general, the results that we obtained fell into the scenario that we had predicted previously. An error rate of almost 50% should mean that the participants were generally not able to distinguish the ”real” screenshots from the ”fake” ones, and so they responded semi-randomly. Furthermore, the median web browsing literacy was fairly close to the value that we wanted, and so the collected data should also apply to the attackers that will interact with the honeypots.

In conclusion, applying the page camou aging process to our base templates was enough to drastically improve how believable they were in the context of each chosen web domain. While the control group was able to single out the ”fake” pages with ease, the experimental group could not distinguish them from the ”real” pages, which resulted in random-like choices. As such, the page camou aging process in template honeypots should be viable enough to make them blend in with the desired domains, but their ef ciency will obviously depend on how the base template is constructed.

4.2 mTANNER modi cations study

In Section 3.6 we discussed the modi cations that we made to TANNER in order to mitigate some of the issues that we had previously found. The objective of these modi cations was to make the resulting honeypots more believable and to consequently maximize their time-wasting and data-capturing capabilities at the trade-off of some coverage. In theory, this should lead our solution to fully meet both the Longevity and Undetectability requirements that we de ned previously, but we wanted to verify it. Therefore, we decided to conduct a study to understand how the data collected by the original TANNER compares to the data collected by our mTANNER.

In order to do this we deployed our solution alongside one of IST's production web applications which serves web pages related to nine teaching departments. Their infrastructure already used a NGINX load balancer, and so introducing our own proxy logic was as simple as copying and pasting a few lines of code into their NGINX con guration le. Afterwards, the proxy was immediately working as intended and all that was left to do was launching both MockingPot and mTANNER. We then deployed these two components in their private cloud platform via a Chef [36] recipe and both of them ran smoothly from then on.

Our intention was to collect data on the interactions with the deployed honeypots for a few weeks. Afterwards, we would then switch the mTANNER instance with the original TANNER and continue collecting data for a similar period of time. However, the web application that we used didn't receive any requests to the honeypot routes that we had de ned during the two weeks that followed.

As such, we came up with a second plan in order to, hopefully, attract more attackers and consequently collect the data that we needed. This plan consisted in registering four sub-domains under IST's web domain which should be appealing to potential attackers: phpmyadmin.tecnico.ulisboa.pt, mantis.tecnico.ulisboa.pt, mantisbt.tecnico.ulisboa.pt and bugtracker.tecnico.ulisboa.pt. All of these domains were directed to a single machine where we installed two popular applications, PHPMyAdmin [37] and Mantis Bug Tracker [38]. These are two web applications that are commonly targeted by attackers since they usually run with elevated permissions and developers sometimes con gure them poorly (default credentials, weak passwords, allowing logging in from remote servers, etc.). Since PHPMyAdmin is a more popular target than Mantis, we directed one domain to PHPMyAdmin (the rst one) and the other three to Mantis as an attempt to balance the accesses to both applications.

Then, we set up two instances of our solution simultaneously, one for each application. The instance associated with PHPMyAdmin used mTANNER while the one associated with Mantis used the original TANNER, and the honeypot routes were the same in both applications. This con guration would allow us to collect data from mTANNER and TANNER simultaneously, and so it was much more time-effective than our original plan in which we would have to switch them midway through. Unfortunately, even after two weeks we had still been unable to collect any data. Initially we thought that this might have been due to the honeypots not being suf ciently attractive, but when we checked the machine's access logs we found out that there had been no accesses to our sub-domains at all during that time interval.

Considering this, we moved on to our third and nal plan as we simply couldn't wait for the previous ones to show results due to time constraints. This plan consisted in asking two pen-testers that were highly knowledgeable with web vulnerabilities to attack the two applications (one for each of them) that we had used for our second plan. They belonged to the same security team and had similar skill sets. They were told that they would be testing the security of web applications that belonged to IST, and we never revealed that they would actually be interacting with honeypots.

Similarly to the second study, this should allow us to collect data from mTANNER and TANNER simultaneously. Even though we weren't collecting data from spontaneous attackers, the traf c that we obtained from these pen-testers should still serve as data for our study, considering that their aim is to look for vulnerabilities and exploit them just like regular attackers. By analyzing the requests collected by the honeypots we should be able to perceive the key differences between how both pen-testers interacted with them. In particular, we hoped that each honeypot associated with mTANNER would receive a larger amount of requests (and for a longer period of time) than its TANNER counterpart. This would imply that our modi cations had improved the longevity of the resulting honeypots.

4.2.1 Results

We will now present the results of our study with the two pen-testers that were asked to test the security of our two web applications. From here on we will be referring to the study with TANNER as the control group and the study with mTANNER as the experimental group. Besides collecting the requests that they sent to our honeypots, we also asked them to take notes of any vulnerabilities that they found and how they were able to exploit them. Additionally, we asked them to write down anything that they thought was behaving unexpectedly.

Both pen-testers were easily able to nd all of the honeypots that we had prepared since we attempted to deploy them on routes that are commonly fuzzed during an attacker's reconnaissance phase[2]. In total there were 22 honeypots emulating 5 different vulnerabilities (SQL Injection, PHP Code Injection, LFI, RFI and XXE) for each web application, with each vulnerability being scattered across multiple honeypot routes in order to increase the likelihood of nding them. However, there was enough data collected using two of the emulated vulnerabilities (SQL Injection and PHP Code Injection) to fully explain the differences between the results produced by the control group and the experimental group. The data collected by the honeypots emulating the other three vulnerabilities was redundant. Therefore, in order to present our results in a manner that is easier to follow, we will discuss the results obtained from those two vulnerabilities in detail in the following paragraphs and then we will present our nal remarks.

SQL Injection honeypots

The rst honeypot found by both pen-testers was a template honeypot with a con guration similar to the one in Listing 3.2. This honeypot presented attackers with a log in page similar to the one shown in Figures 3.6 and A.3, as it also attempted to mimic the styling used by IST's log in page. This was

[2] This serves as further proof that our two previous studies failed to collect any results simply because there were no accesses with malicious intents in the time period during which they were conducted

the honeypot where the differences between the control group and the experimental group were most noticeable, as we will explain in the next paragraphs.

First of all, the pen-tester from the control group was able to obtain a dump of the entire database in only 11 requests, without using any tools. These requests were sent within a time frame of 5 minutes and 22 seconds, which was the time that it took him to obtain the database dump. Furthermore, since the passwords generated by TANNER are not hashed, he immediately attempted to log in with the credentials that he got from the dump. Listing 4.1 shows two of the recorded requests: in the first one the exploit was successful and, immediately after, there was an attempt to use the dumped credentials in order to log in. The pen-tester noted that the credentials looked randomly generated, which was one of the problems we mentioned in Sub-section 3.6.2. Furthermore, he was convinced that his log in attempt should have worked since he used a valid username and password.

```
1  {
2      "path":"http://mantis.tecnico.ulisboa.pt/admin",
3      "timestamp":"29-10-2021 21:21:12",
4      "response_status":{"  $numberInt":"200"},
5      "attack_type":"sqli",
6      "post_data":{"username":"admin","password":"admin\" or \"1\"=\"1"},
7      "result":"[0, 'alist1980', 'burrus2038@outlook.com', 'B^[m|-)>']
8                [1, 'Allergy1829', 'bachelor2041@live.com', \"26>{='*b\"]
9                [2, 'anasa.1963', 'coerced2026@outlook.com', '9putoWh.'] ..."
10 }
11 {
12     "path":"http://mantis.tecnico.ulisboa.pt/admin",
13     "timestamp":"29-10-2021 21:22:29",
14     "response_status":{"  $numberInt":"200"},
15     "attack_type":"unknown",
16     "post_data":{"username":"alist1980","password":"B^[m|-)>"}
17 }
```

Listing 4.1: SQL Injection exploit from the control group

In contrast, the pen-tester from the experimental group was only able to dump a single password after 663 requests. The first 28 requests corresponded to him attempting some generic payloads manually. Even though the pen-tester in the control group was able to obtain a database dump in only 11 requests, this was not the case here since the experimental group was using our sqli_blind modifier. The experimental pen-tester ran an automated tool in order to perform some additional reconnaissance and it revealed the existence of a blind SQL Injection vulnerability in the username parameter. This tool produced an additional 359 requests.

Afterwards, this pen-tester coded an exploit in order to brute-force the password of the admin account, which is shown in Listing 4.2. This exploit uses the SQL LIKE operator in order to brute-force the admin's password character by character. Coding the exploit only took him about five minutes, but in that time period the control pen-tester had already obtained the entirety of the database's contents.

```
import requests

s = requests.Session()
url = "https://phpmyadmin.tecnico.ulisboa.pt/admin"

def exploit(payload):
    username = 'admin" AND password LIKE "{}%" -- '.format(payload)
    payload = {"username":username, "password": "asd"}
    r = s.post(url,data=payload)
    return "Invalid" in r.text

import string
password = ""

for i in range(100):
    found = False
    for c in string.ascii_letters + string.digits:
        if not exploit(password+c):
            password+=c
            found = True
            break
    if not found:
        break

print(password)
```

Listing 4.2: Python exploit used to obtain the admin's password

Running this exploit produced an additional 276 requests. He was able to obtain the admin's password but it was hashed since mTANNER hashes the passwords in the dummy database. After obtaining this hashed password, he attempted (without success) to use Crackstation [39] in order to crack the actual password, which is a behavior we predicted in Sub-section 3.6.2. Considering the time spent on coding the exploit, this pen-tester spent around 15 minutes on this honeypot alone and got much less information than the pen-tester from the control group.

PHP Code Injection honeypots

There were two differences that are worth mentioning between the two groups regarding these honeypots. The first one is that the pen-tester from the control group noticed that any parameter could be used in order to exploit this vulnerability, similarly to the scenario shown in Figure 3.12, and reported it as a weird occurrence. However, this didn't happen in the experimental group due to the fact that mTANNER only searches for malicious payloads in the parameters specified in MockingPot's configuration file (as shown in Figure 3.13).

The second one is that the pen-tester from the control group also noticed that these honeypots were vulnerable to payloads that targeted other vulnerabilities, similarly to the scenario presented in Figure 3.14 but with LFI instead of SQL Injection. He reported it as a weird occurrence since it didn't make sense for LFI payloads to work in the context of the error shown to him. This didn't happen in the experimental group since mTANNER only emulates the vulnerabilities that are specified in MockingPot's configuration file for each honeypot.

The honeypots that emulated these vulnerabilities registered a similar number of requests between the control group and the experimental group: the control group registered a total of 53 requests targeting PHP Code Injection honeypots, while the experimental group registered 42. In this case the honeypot from the control group registered a few more requests than the honeypot from the experimental group. However, this is explained by the fact that the control pen-tester attempted to exploit LFI in addition to the intended PHP Code Injection vulnerability.

Other honeypots and final remarks

The other honeypots ended up producing similar results in the control and experimental groups. The lfi_files modifier used in the experimental group ended up not influencing the collected data since the experimental pen-tester was able to find the honeypot easily without relying on this feature (he simply found it by fuzzing the route with a list of popular directories). The RFI and XXE honeypots also produced similar results in both groups and with no additional observations.

The data collected in the SQL Injection honeypots showed that the number of requests sent to the honeypots associated with mTANNER (663) was much higher than the number of requests sent to the honeypots associated with the original TANNER (11). Furthermore, the pen-tester that interacted with mTANNER spent around 15 minutes on that honeypot alone, which is almost three times longer than the interval spent there by the pen-tester from the control group (5 minutes and 22 seconds). Therefore, this seems to support our claim that the modifications introduced into mTANNER increase the time-wasting capabilities of the resulting honeypots and contribute to our solution fully meeting the Longevity requirement.

At the same time, the pen-tester from the control group reported multiple unexpected behaviors: the first one were the credentials that he obtained through the SQL Injection honeypot, since they didn't match the credentials that he would expect from an application related to IST; the second one was the fact that any parameter could be used in order to trigger a PHP Code Injection exploit; and the final one was that the same honeypot was also vulnerable to LFI payloads, which didn't make sense considering the error shown by the page. Since the pen-tester from the experimental group didn't report any strange behaviors related to the honeypots, this should serve as evidence that our modifications heavily contribute towards making the resulting honeypots much harder to detect. Therefore, we can conclude that our solution also meets the Undetectability requirement.

4.3 Proxy overhead study

One of the requirements that we had previously de ned for our solution was that it should have a negligible impact on the performance of existing web applications, even in production environments. Therefore, we decided to do some performance benchmarks in order to prove that our solution meets this requirement. This study was conducted on the same web application that was used for the previous study (discussed in Section 4.2).

In order to perform these benchmarks we used ApacheBench [40], a simple command-line tool commonly used to conduct load testing and performance evaluation on web servers. Essentially, this tool sends multiple requests to a target URL and collects the respective responses. Afterwards, it computes certain metrics from the obtained responses such as the number of requests handled per second and the mean time per request.

The objective of this study was to compare how the web application performed in two different scenarios: one where it remained unaltered and another one where we associated it with our solution. We then used this tool to simulate and evaluate three different levels of traf c: low (10 non-concurrent requests), moderate (1000 total requests composed by batches of 100 concurrent requests) and high (5000 total requests composed by batches of 500 concurrent requests). These requests would be sent to a non-honeypot route and, ideally, we would see very similar results in terms of the mean time per request in both scenarios and for each level of traf c. We expected this to be the case since the only difference between these two scenarios is that the second one uses our NGINX proxy logic (see Section 3.4) while the rst one doesn't (which in theory should only introduce a negligible overhead).

Rather than performing a single benchmark for each level of traf c, we decided to perform ve and we considered the average of all the obtained results. The reasoning for this is that web servers often perform inconsistently, and it is common to see variations in the metrics that we mentioned previously.

4.3.1 Results

Table 4.1 shows the results that we obtained from this study, for both scenarios and the three levels of traf c that we simulated.

	Low traf c (concurrency=1)	Moderate traf c (concurrency=100)	High traf c (concurrency=500)
Scenario 1 (without proxy logic)	9.258 ms / request	312.697 ms / batch 3.127 ms / request	1669.864 ms / batch 3.340 ms / request
Scenario 2 (with proxy logic)	9.278 ms / request	321.354 ms / batch 3.214 ms / request	1695.620 ms / batch 3.391 ms / request
Overhead	0.020 ms / request	8.657 ms / batch 0.087 ms / request	25.726 ms / batch 0.051 ms / request

Table 4.1: Average time per request obtained across multiple benchmarks

The average times per request obtained for the second scenario were all marginally higher than the ones obtained for the rst scenario. However, this difference was only of a few tens of nanoseconds for each request in all levels of traf c, which should be considered a negligible overhead. Furthermore, the

relation between this overhead and the traf c levels was not linear: the biggest overhead was registered for moderate traf c, followed by high and then low. We expected this time difference to grow linearly with the concurrency of incoming requests but this was not the case, most likely, due to the inconsistencies that we mentioned. The results obtained for the high level of traf c should also be indicative of how our solution affects fairly active production environments. Therefore, we concluded that our solution meets the Performance requirement as we did not register any signi cant overhead with our benchmarks, even for high levels of traf c.

4.4 Requirements

To conclude this chapter, we will now present a review table (Table 4.2) that shows how our solution met our requirements in comparison to the ones which we introduced in Chapter 2. It should be noted that this table assumes an appropriate honeypot con guration.

	WordPress module	B.Hive	CanaryTokens	HIHAT	SNARE & TANNER	Our solution
Functionality	N	Y	Y	Y	N	Y
Performance	N	Y	Y	N	N	Y
Information Coverage	Y	N	N	Y	Y	P
Security	N	Y	Y	Y	P	Y
Compatibility	N	P	Y	N	Y	Y
Longevity	N	N	N	N	P	Y
Undetectability	N	Y	Y	Y	N	Y
Data Analysis	N	N	N	Y	Y	Y

Table 4.2: Summary table showing how our solution meets the requirements

We managed to fully meet seven of the requirements that we had previously de ned, even though the Information Coverage requirement is now only partially met due to the limitations that we imposed on the emulation process. However, we believe that this change was essential since it allowed us to fully meet both Longevity and Undetectability, as evidenced by the study presented in Section 4.2.

Our solution also seems to have satis ed more requirements than the others that we presented in Chapter 2, surpassing both CanaryTokens and HIHAT which only fully meet ve requirements. Furthermore, by modifying TANNER and using it with MockingPot rather than SNARE we were able to fully meet the Functionality and Performance requirements that they previously did not.

Chapter 5

Conclusions

A honeypot is a resource whose value lies in being attacked. There are many tools that allow defenders to deploy honeypots in order to protect their systems. However, most of these tools deploy isolated honeypots that run in parallel with those systems. After being incorporated into the systems themselves, the honeypots should be more easily found by attackers and also more likely to pass as a legitimate part of them, consequently resulting in more data being collected.

As such, we started looking for tools that were capable of automatically integrating honeypots into existing systems but we found very few. Furthermore, the few that we did nd were fairly limited in terms of what the honeypots could do. For example, B.Hive (which we presented in Section 2.2) can integrate honeypots into web applications but those honeypots have a xed behavior and can only detect payloads sent within HTML form elds.

We then started designing our own honeypot solution with three objectives in mind. First, we wanted our solution to allow defenders to have an extensive control over the resulting honeypots while still keeping them simple to generate. Additionally, our solution should be able to automatically integrate the generated honeypots into existing web applications, so that they can bene t from the advantages that we enumerated above. Finally, the honeypots should also attempt to maximize the number of interactions with attackers in order to collect as much data as possible.

Our solution is composed by three main modules: (i) a proxy/load balancer which intercepts traf c between users and the web application and determines when users should be directed to a honeypot; (ii) a server called MockingPot which generates and serves the honeypot pages as speci ed by the defenders in a declarative con guration le; and (iii) mTANNER which analyzes the requests sent to the honeypots, collects incoming malicious data and emulates vulnerabilities in order to attract attackers for long periods of time. The interactions between these modules are represented in a BPMN diagram which we included in Appendix B.

MockingPot is also capable of generating honeypot pages that automatically copy the styling used by actual pages served by the web application that is being protected. This feature allows honeypots to adapt to the surrounding environments, thus resulting in more believable and harder to detect honeypot pages. The effectiveness of this process was proven through the study described in Section 4.1.

Our evaluation showed that the overhead introduced by using this solution is negligible (see Section 4.3). As such, defenders can use it in order to easily con gure multiple honeypots which will defend their web applications with negligible drawbacks. Furthermore, well-con gured honeypots will be hard to detect and should keep attackers interested for long periods of time by emulating responses to their malicious payloads as shown by the study described in Section 4.2.

5.1 Achievements

In this section we summarize the most important merits of our solution:

- It is easy to set up and it should also be compatible with any existing web application;
- Our solution does not affect the regular functioning of the web application behind it;
- Using our solution doesn't require any modi cations to existing web applications . Therefore, it can be done by operation/security teams alone without the intervention of the development teams;
- Setting up new honeypots is as easy as editing the MockingPot con guration le and creating optional HTML pages (not required for text honeypots);
- It can automatically integrate honeypots into existing web applications while still allowing defenders to extensively customize those honeypots, unlike the previous solutions that we found;
- Our proxy is implemented in an unorthodox manner so that it doesn't need to be changed when honeypot routes are added and/or removed;
- There is no signi cant performance overhead associated with the use of our solution (even in production environments) as shown by the study described in Section 4.3;
- We use a method of camou aging honeypot pages by automatically styling them based on information collected from the applications (template honeypots). The effectiveness of this method was shown through the study described in Section 4.1. Since a single template can generate fairly different pages for different web applications, templates can also be shared amongst defenders in order to further save time and effort;
- mTANNER imposes some limitations on the vulnerability emulators which make the honeypots harder to detect. Unlike TANNER, mTANNER only looks for payloads in the parameters speci ed for each honeypot in the MockingPot con guration le, which prevents undesired scenarios such as the one shown in Figure 3.12. Furthermore, mTANNER also only emulates the vulnerabilities speci ed for each honeypot, which prevents the scenario shown in Figure 3.14;
- mTANNER provides more options for defenders to customize the dummy databases used in the SQL Injection emulator, which should allow them to create more believable databases. The sqli_blind modi er can also be used in order to emulate a blind SQL Injection vulnerability, which

can greatly improve the data collected by honeypots that emulate this vulnerability (as show by the study described in Section 4.2);

- mTANNER runs in six Docker containers rather than the ve containers used by TANNER as an additional security measure. We also implemented multiple other smaller features and modi cations which were brie y described in Section 3.6.5;
- Well-con gured honeypots should be able to keep attackers interested for long periods of time without being detected due to the response emulation provided by mTANNER. We proved this through the study described in Section 4.2.

5.2 Future Work

There were a few ideas that we came up with while developing our solution but could not implement due to time constraints. The rst one is formatting and sending logs to common logging pipelines such as Logstash [41] and Rsyslog [42]. mTANNER/TANNER already supports logging with the hpfeeds [43] protocol, but the pipelines that we mentioned are more popular for processing logs.

We also want to enhance the attacker recognition algorithm used by mTANNER/TANNER, which is only vaguely capable of determining if the entity responsible for a session is an administrator, a crawler, a tool or a manual attacker (see Section 2.5 for more information). If we used a more in-depth classi cation system [25], we would also be able to determine which logs should be sent to the pipelines with greater accuracy. Furthermore, we could even implement some browser ngerprinting techniques [44, 45] in order to obtain some additional information from attackers.

Finally, we also considered setting honeypot routes dynamically according to incoming traf c. For example, if 10 requests were sent to a speci c route that does not currently contain a honeypot, one of the existing honeypots would be duplicated and assigned to that route. Doing this would bring its share of challenges but it would also be very rewarding since popular routes would automatically become honeypots without needing any intervention from the defenders. Despite this, more complex methods of dynamically generating honeypots have obviously been proposed [46, 47].

Bibliography

[1] Risk Based Security. https://www.riskbasedsecurity.com/ , 2011 (last accessed 9 September 2021).

[2] M. Dermann, M. Dziadzka, B. Hemkemeier, A. Hoffmann, A. Meisel, M. Rohr, and T. Schreiber. OWASP Papers Program Best Practice: Use of Web Application Firewalls Best Practices: Use of Web Application Firewalls. URL https://owasp.org/www-pdf-archive/Best_Practices_WAF_v105.en.pdf .

[3] D. Thomas-Reynolds and S. Butakov. Factors Affecting the Performance of Web Application Firewall. URL https://aisel.aisnet.org/wisp2020/8 .

[4] N. Clarke, S. Furnell, G. Tjhai, and M. Papadaki. Investigating the problem of IDS false alarms: An experimental study using Snort. volume 278, 07 2008. doi: 10.1007/978-0-387-09699-5_17.

[5] W. Bulajoul, A. James, and M. Pannu. Network Intrusion Detection Systems in High-Speed Traf c in Computer Networks. In 2013 IEEE 10th International Conference on e-Business Engineering, pages 168–175, 2013. doi: 10.1109/ICEBE.2013.26.

[6] C. Leita, V. Pham, O. Thonnard, E. Ramirez-Silva, F. Pouget, E. Kirda, and M. Dacier. The Leurre.com Project: Collecting Internet Threats Information Using a Worldwide Distributed Honeynet. In 2008 WOMBAT Workshop on Information Security Threats Data Collection and Sharing, pages 40–57, 2008. doi: 10.1109/WISTDCS.2008.8.

[7] M. T. Qassrawi and Z. Hongli. Deception Methodology in Virtual Honeypots. In 2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing, volume 2, pages 462–467, 2010. doi: 10.1109/NSWCTC.2010.266.

[8] T. Yagi, N. Tanimoto, T. Hariu, and M. Itoh. Enhanced attack collection scheme on high-interaction web honeypots. In The IEEE symposium on Computers and Communications, pages 81–86, 2010. doi: 10.1109/ISCC.2010.5546700.

[9] H. Sanghvi and M. Dahiya. Cyber Reconnaissance: An Alarm before Cyber Attack. International Journal of Computer Applications, 63:36–38, 02 2013. doi: 10.5120/10472-5202.

[10] N. Provos and T. Holz. Virtual Honeypots - From Botnet Tracking to Intrusion Detection. 01 2008. ISBN 978-0-321-33632-3.

[11] C. Pohl, A. Zugenmaier, M. Meier, and H.-J. Hof. B.Hive: A Zero Con guration Forms Honeypot for Productive Web Applications. volume 455, 05 2015. doi: 10.1007/978-3-319-18467-8_18.

[12] M. Müter, F. Freiling, T. Holz, and J. Matthews. A generic toolkit for converting web applications into high-interaction honeypots. 01 2008.

[13] WordPress. https://wordpress.com/ , 2003 (last accessed 2 December 2020).

[14] H. Trunde and E. Weippl. WordPress Security: An Analysis Based on Publicly Available Exploits. In Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services, iiWAS '15, New York, NY, USA, 2015. Association for Computing Machinery. doi: 10.1145/2837185.2837195.

[15] I. Medeiros, N. Neves, and M. Correia. Equipping WAP with WEAPONS to Detect Vulnerabilities: Practical Experience Report. In 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 630–637, 2016. doi: 10.1109/DSN.2016.63.

[16] I. Cernica and N. Popescu. Wordpress Honeypot Module. In 2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC), pages 9–13, 2018. doi: 10.1109/EUC.2018.00009.

[17] Top sites - Alexa. https://www.alexa.com/topsites , 2010 (last accessed 20 August 2021).

[18] CanaryTokens. https://canarytokens.org/ , 2015 (last accessed 26 November 2020).

[19] High Interaction Honeypot Analysis Toolkit. http://hihat.sourceforge.net/ , 2006 (last accessed 18 December 2020).

[20] MushMush Foundation. http://mushmush.org/ , (last accessed 19 November 2020).

[21] Docker Compose. https://docs.docker.com/compose/ , (last accessed 18 September 2021).

[22] Super Next generation Advanced Reactive honEypot. https://github.com/mushorg/snare , 2015 (last accessed 2 March 2021).

[23] Glastopf. https://github.com/mushorg/glastopf , 2009 (last accessed 19 November 2020).

[24] TANNER. https://github.com/mushorg/tanner , 2015 (last accessed 1 September 2021).

[25] N. Kuze, S. Ishikura, T. Yagi, D. Chiba, and M. Murata. Detection of Vulnerability Scanning Using Features of Collective Accesses Based on Information Collected from Multiple Honeypots. In NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium. IEEE Press, 2016. doi: 10.1109/NOMS.2016.7502962.

[26] R. A. et al. Web Application Firewall Evaluation Criteria. In Web Application Security Consortium, 2006. Version 1.0.

[27] Business Process Model and Notation. https://www.bpmn.org/ , (last accessed 22 October 2021).

[28] X. Chi, B. Liu, Q. Niu, and Q. Wu. Web Load Balance and Cache Optimization Design Based Nginx under High-Concurrency Environment. In 2012 Third International Conference on Digital Manufacturing Automation, pages 1029–1032, 2012. doi: 10.1109/ICDMA.2012.241.

[29] Handlebars. https://handlebarsjs.com/ , (last accessed 12 March 2021).

[30] Handlebars conditionals. https://handlebarsjs.com/guide/block-helpers.html#conditionals , (last accessed 12 March 2021).

[31] BusyBox. https://www.busybox.net/ , (last accessed 16 September 2021).

[32] H. Davis. Search Engine Optimization. O'Reilly Media, 2006. ISBN 9781491905883.

[33] PHP Manual - $_GET. https://www.php.net/manual/en/reserved.variables.get.php , (last accessed 12 September 2021).

[34] Mimesis - Fake Data Generator. https://mimesis.readthedocs.io/ , (last accessed 5 August 2021).

[35] Aiodocker - Python Docker API. https://aiodocker.readthedocs.io/en/latest/ , (last accessed 27 August 2021).

[36] Chef Software DevOps Automation Solutions. https://www.chef.io/ , (last accessed 24 October 2021).

[37] PHPMyAdmin. https://www.phpmyadmin.net/ , (last accessed 24 October 2021).

[38] Mantis Bug Tracker. https://www.mantisbt.org/ , (last accessed 24 October 2021).

[39] Crackstation - Online Password Hash Cracking. https://crackstation.net/ , (last accessed 29 October 2021).

[40] ApacheBench - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html , (last accessed 15 October 2021).

[41] Logstash. https://www.elastic.co/logstash/ , (last accessed 24 October 2021).

[42] Rsyslog. https://www.rsyslog.com/ , (last accessed 24 October 2021).

[43] hpfeeds. https://hpfeeds.org/ , (last accessed 24 October 2021).

[44] R. Upathilake, Y. Li, and A. Matrawy. A classi cation of web browser ngerprinting techniques. In 2015 7th International Conference on New Technologies, Mobility and Security (NTMS), pages 1–5, 2015. doi: 10.1109/NTMS.2015.7266460.

[45] N. Kaur, S. Azam, K. Kannoorpatti, K. C. Yeo, and B. Shanmugam. Browser Fingerprinting as user tracking technology. In 2017 11th International Conference on Intelligent Systems and Control (ISCO), pages 103–111, 2017. doi: 10.1109/ISCO.2017.7855963.

[46] G. Wagener, R. State, T. Engel, and A. Dulaunoy. Adaptive and self-con gurable honeypots. pages 345–352, 05 2011. doi: 10.1109/INM.2011.5990710.

[47] D. Fraunholz, M. Zimmermann, and H. D. Schotten. An adaptive honeypot con guration, deployment and maintenance strategy. In 2017 19th International Conference on Advanced Communication Technology (ICACT), pages 53–57, 2017. doi: 10.23919/ICACT.2017.7890056.

Appendix A

Screenshots

Figure A.1: Honeypot log in page with an error message

Figure A.2: Honeypot log in page example with SQL Injection emulation results

Figure A.3: Template honeypot page based on the IST log in page

Figure A.4: IST log in page

Figure A.5: Template honeypot page based on the ULisboa log in page

Figure A.6: ULisboa log in page

Figure A.7: Question from the control group form

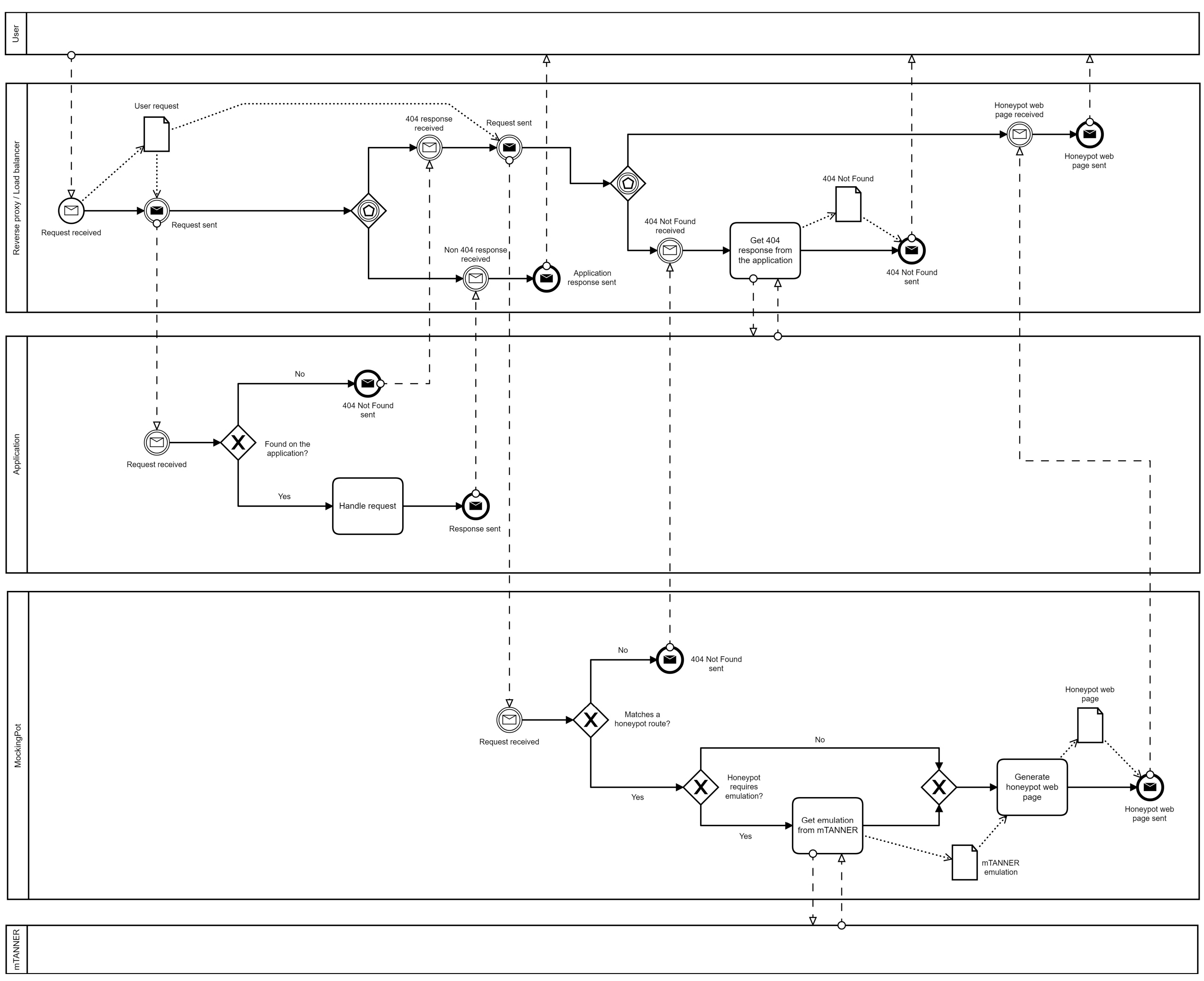
Figure A.8: Question from the experimental group form

Appendix B

Technical Diagrams

B.1 BPMN diagram representing the behaviour of our solution

(See next page in landscape layout)


User
Reverse proxy / Load balancer
User request
Request received
Request sent
404 response received
Request sent
Non 404 response received
Application response sent
Honeypot web page received
Honeypot web page sent
404 Not Found received
Get 404 response from the application
404 Not Found
404 Not Found sent
Application
No
404 Not Found sent
Request received
Found on the application?
Yes
Handle request
Response sent
MockingPot
No
404 Not Found sent
Request received
Matches a honeypot route?
No
Honeypot requires emulation?
Yes
Yes
Get emulation from mTANNER
Generate honeypot web page
Honeypot web page
Honeypot web page sent
mTANNER emulation
mTANNER

Appendix C

MockingPot configuration parameters

```
1  application:
2    protocol: <'http'|'https'>
3    host: <host>
4    route: <route> (optional)
5    port: <port> (optional)
6
7  analysis:
8    protocol: <'http'|'https'>
9    host: <host>
10   port: <port> (optional)
11
12 honeypots:
13   # CUSTOM HONEYPOTS
14   - category: custom
15     route:  <route>
16     page: <HTML file location>
17     error_get: <error message>
18     error_post: <error message>
19     emulationMethods: [<'get'|'post'>]
20     emulationParameters: { <'sqli'|'rfi'|'lfi'|'php_code_injection'
21        |'php_object_injection'|'cmd_exec'|'crlf'|'xxe_injection'
22        |'template_injection'>: [<parameter>] }
23     modifiers: (optional)
24       sqli_blind: <true|false>
25       success_message: <success message> (replaces mTANNER responses)
26       disallow_get: <true|false> (disables handling GET requests)
27     injectionId: <ID of a HTML tag>
```

```
28
29  # TEMPLATE HONEYPOTS
30  - category: template
31    route:  <route>
32    template: <template file location>
33    error_get: <error message>
34    error_post: <error message>
35    emulationMethods: [<'get'|'post'>]
36    emulationParameters: { <'sqli'|'rfi'|'lfi'|'php_code_injection'
37       |'php_object_injection'|'cmd_exec'|'crlf'|'xxe_injection'
38       |'template_injection'>: [<parameter>] }
39    modifiers: (optional)
40      sqli_blind: <true|false>
41      success_message: <success message>
42      disallow_get: <true|false>
43    provider: <provider> (optional)
44    basePage: <URL to web page> (optional)
45    icon: <URL to favicon> (optional)
46    background: <background CSS property> (optional)
47    font: <font-family CSS property> (optional)
48    color: <color CSS property> (optional)
49    logo: <URL to logotype image> (optional)
50
51  # TEXT HONEYPOTS
52  - category: text
53    route:  <route>
54    error_get: <error message>
55    error_post: <error message>
56    emulationMethods: [<'get'|'post'>]
57    emulationParameters: { <'sqli'|'rfi'|'lfi'|'php_code_injection'
58       |'php_object_injection'|'cmd_exec'|'crlf'|'xxe_injection'
59       |'template_injection'>: [<parameter>] }
60    modifiers: (optional)
61      sqli_blind: <true|false>
62      lfi_files: <true|false>
63      success_message: <success message>
64      disallow_get: <true|false>
```

Listing C.1: All supported honeypot configurations