MockingPot: Generate and Integrate Honeypots Into Existing Web Applications

Pedro Alves
pedro.paixao.alves@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2021

Abstract

Honeypots are "decoy" computer resources meant to be attacked in order to divert attention 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 but, to the best of our knowledge, ours can deploy more customizable honeypots than them. Our honeypots are presented to the users as web pages which can be extensively configured by the 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. Our evaluation showed that the overhead introduced when using this solution is negligible and that our method for camouflaging honeypot pages is effective. This solution should also 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

1. Introduction

Cyber-security is a major concern in the modern world. According to Risk Based Security [3], 7098 security breaches were reported in 2019 and over 15.1 billion records were exposed. It is critical for companies to invest in reliable defense mechanisms, and 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 [11] 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 [17]. 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 [10]. 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 [9].

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 [12], 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. Most honeypots will emulate vulnerabilities or use other deception techniques [15, 18] 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;

- *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 [16]. 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 attacks before they even happen;

- *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 [15].

1.1. 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 [17], honeypots can be associated with web applications without having a noticeable impact on their performance. Unlike IDS and IPS [10], 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 [14] (that we present further ahead in Section 2.1) 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.

1.2. 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 implement attack classification and vulnerability emulation into 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.

2. Background

We will now present some of the papers/tools regarding web-based honeypot techniques that we found and analyzed during the research that was conducted for this thesis.

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

In this paper [14], Pohl et al. present B.Hive, an approach that can automatically blend honeypot components into existing web applications without altering them.

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 form fields are invisible to legitimate users and, therefore, changes to them are an indicator of malicious activity. 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.

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.

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. 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.

2.2. HIHAT

HIHAT [1] 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.

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.

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 PHP applications. 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 [13] which is based on this tool), which would make it undesirable for production environments.

2.3. SNARE & TANNER

The MushMush Foundation [8] is an organization dedicated to developing open-source software. One of their projects in particular became a source of inspiration for this thesis. That project is composed by two different tools that are meant to be used together and which shall be introduced in the following paragraphs.

The first tool is SNARE [5], which comprises the frontend part of the solution. SNARE is the successor of another project from MushMush called Glastopf [2], 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 [4], 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).

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 scan-

ners, 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.

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. Therefore, we decided to modify it in order to mitigate the issues that we mentioned and use it as the provider for our data analysis and response emulation, as we will discuss in the next chapter.

3. Implementation

3.1. Solution architecture

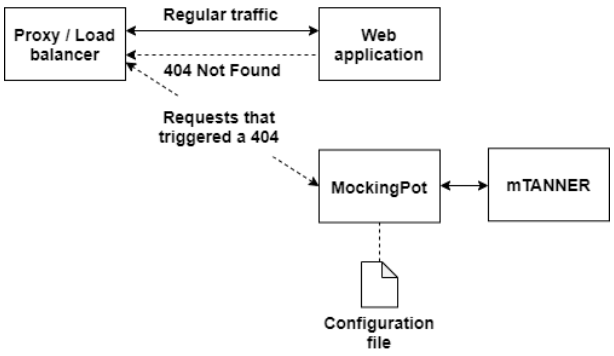

Figure 1: Solution architecture

The diagram shown in Figure 1 depicts the architecture of our solution, which is formed by the following components:

- *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 that 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.

3.2. Solution behavior

We will now explain the behavior of each component and how they communicate. 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 2 shows an example of a simple honeypot login page which we created to be served in the */admin* route and which should should seem vulnerable to SQL Injection attacks upon some interaction.

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 as shown. Figure 2 shows an example of a response where no payloads were detected and figure 3 shows one where a SQL Injection payload was detected.

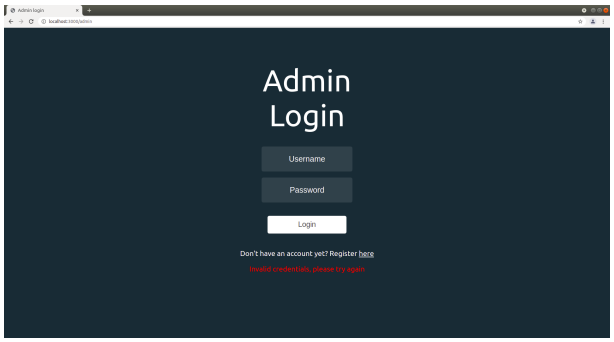

Figure 2: Honeypot login page with an error message

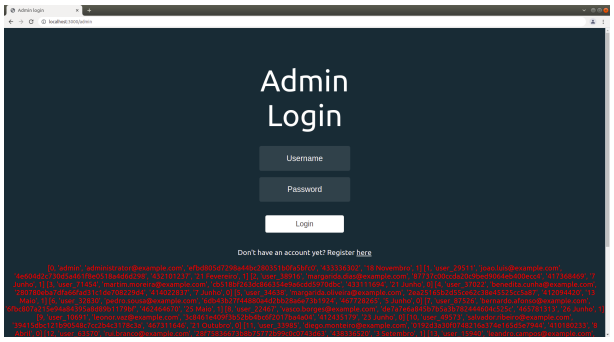

Figure 3: Honeypot login page example with SQL Injection emulation results

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.

3.3. MockingPot

As mentioned before, the honeypots are defined in a declarative configuration file. Within that file defenders can define the parameters of the desired honeypots, such as the route in which they should be deployed, the vulnerabilities that should be emulated, the list of parameters where payloads should be sent, and the error message that should be presented to the users if the emulation process failed (either due to no payloads being found or due to no valid responses being generated). Furthermore, each honeypot must also fall into one of three categories, which we will introduce in the next subsections.

3.3.1 Custom honeypots

This category's main particularity is that it requires the path to a valid HTML file, 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 2 and 3 was created using this category. Defenders must also specify an `injectionId`, which corresponds to the id of the HTML element in which the emulated responses (or the error message mentioned previously) should be inserted and returned to the users.

These honeypots will be most believable when the style of the pages created by the defenders resembles that of other pages served by the application. Furthermore, the simulated vulnerabilities should also have a heavy impact on the attention that they attract. As such, the honeypots' effectiveness will greatly depend on the effort and creativity used in order to construct the pages that they present.

3.3.2 Template honeypots

This category generates web pages which can automatically mimic the style of other ones served by the application. This will allow defenders to bypass some of the effort that was mentioned in the previous category. To create a template honeypot, defenders must first specify the path to a template file. These templates are basically HTML pages containing Handlebars [6] expressions (defined with four curly braces).

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 appropriate values for the most common text font/color used, a background image or color (if no image is found), a logotype image and the shortcut icon (favicon). These values are then introduced into the Handlebars template specified by the defenders, and so it is likely that the style of the resulting honeypot page will end up resembling the one of the scraped page. 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 these values don't satisfy the defenders, they can also be individually overwritten in the configuration file.

Figure 4 shows an example of a page generated with a template honeypot where none of those values were 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; and a placeholder image was used to fill in for the missing logo.

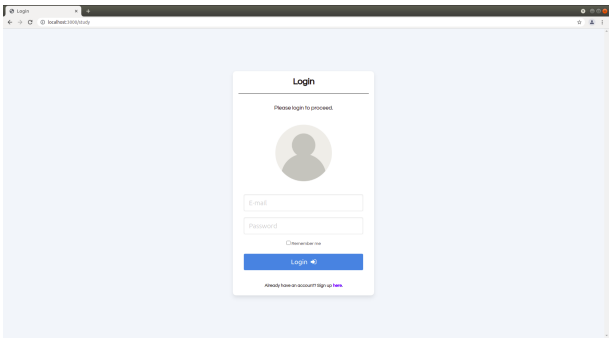

Figure 4: Template honeypot page without a `basePage` defined

Figure 5 shows a honeypot page (generated using the previous template) where all of the mentioned values were automatically extracted from the Instituto Superior Técnico (IST) page. Depending on the chosen `basePage`, a single template can achieve many different styles. In this page, 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. Figure 6 shows the `basePage` that was used to automatically stylize this honeypot page.

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

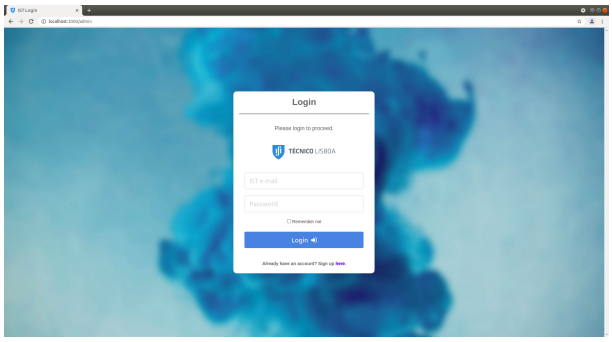

Figure 5: Template honeypot page based on the IST login page

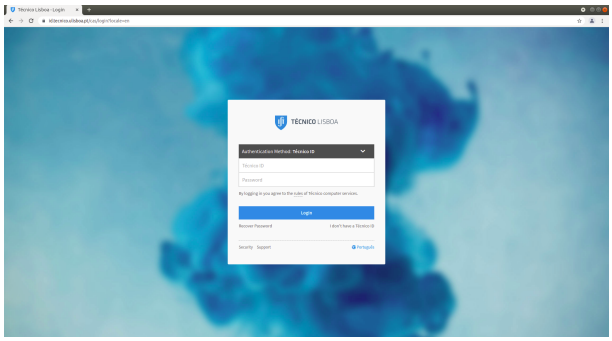

Figure 6: IST login page

they are interacting with honeypots due to their increased public exposure.

3.3.3 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. An example of an effective honeypot configuration that can be deployed with this category is a fake PHP script which returns the error message *$_GET['file'] not found* in order to seem vulnerable to Local File Inclusion (LFI) payloads sent in the *file* parameter.

3.4. mTANNER

As the name suggests, we modified the TANNER tool that we presented in Section 2.3 in order to implement this module since it already implemented some of the functionality that we desired. By using TANNER, our solution benefits from its session creation functionalities. We extended it in order to mitigate some of the main issues that we mentioned previously and to increase its general effectiveness. Next, we will discuss some of the most important modifications and their consequences.

3.4.1 Limiting the emulation process

TANNER originally searches for payloads within all of the parameters defined 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 means that any parameter can be used to exploit the emulated vulnerability, which 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 find a small amount of vulnerabilities (if any). However, a honeypot associated with the original TANNER tool will flag 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 most likely avoid it entirely.

In order to avoid this, mTANNER only searches for payloads in the parameters specified by the defenders in MockingPot's configuration file. By limiting the number of parameters that lead to the emulated vulnerabilities, the resulting honeypots should look more consistent and believable from the point of view of the attackers. This will naturally make it harder for attackers to find 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.

3.4.2 Additional dummy database customization

In order to mitigate the inconsistencies with the dummy databases which we mentioned in Section 2.3, we added a few options to the original database configuration file which the defenders can edit in order to maximize the consistency of their dummy data. 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 specifies a specific template/mask to be followed while generating usernames; and the *first_names* and *last_names* options which specifies

the location of a file containing first and last names to be used in the e-mail addresses.

3.4.3 Other modifications

We also introduced other smaller features and modifications, some of which we will now briefly enumerate:

- Created a separate container for Docker-based emulation due to security concerns;
- RFI emulation can now serve static files to increase the likelihood of attackers triggering it;
- 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 body payloads on POST requests when inspecting sessions;
- 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.

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. We will describe those studies and present the results that we obtained from each of them in the next three sections.

4.1. Page camouflaging study

In order to verify if our template honeypots could blend in with existing web sites, 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, entrants 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"). 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. 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.

We then ended up with 80 entrants in total, 40 in the experimental group and 40 in the control group. In the experimental group, the total percentage of correct answers across all of the questions was 50,31%. In the control group, the total percentage of correct answers across all of the questions was 98.13%, which also corresponds to the scenario that we had previously envisioned. As such, we concluded that the page mimicking process in template honeypots should be viable enough to make them blend in with the desired domains, but their efficiency will obviously depend on how the base template is constructed.

4.2. Tanner modifications study

In Section 3.4 we discussed the modifications that we made to TANNER in order to mitigate some of the issues that we had previously found. The objective of these modifications 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. However, we wanted to empirically verify this and so we decided to conduct a study to understand how the data collected by the original TANNER compares to the data collected by mTANNER.

In order to do this we asked two pen-testers that were highly knowledgeable with web vulnerabilities to attack two applications (one for each) that we had set up. These applications used a similar distribution of honeypot routes, but one of them was using mTANNER in order to emulate vulnerabilities (experimental group) while the other one was using the original TANNER (control group). The pen-testers 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.

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 inter-

acted 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.

At the same time, the pen-tester from the control group reported some unexpected behaviors. He reported that the credentials he obtained through the SQL Injection honeypot seemed randomly generated, and that any parameter could be used in order to trigger the emulated vulnerabilities. We predicted both of these scenarios in Sub-sections 3.4.1 and 3.4.2. 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.

4.3. Proxy overhead study

One of the concerns that we had with our solution was that it should not have a meaningful 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 [7], a simple command-line tool commonly used to conduct load testing and performance evaluation on web servers. 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 traffic: 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). Ideally, we would see very similar results in terms of the mean time per request in both scenarios and for each level of traffic. 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 while the first one doesn't (which in theory should only introduce a negligible overhead).

The average times per request obtained for the second scenario were all marginally higher than the ones obtained for the first scenario. However, this difference was only of a few tens of nanoseconds for each request in all levels of traffic, which should be considered a negligible overhead. Furthermore, the relation between this overhead and the traffic levels was not linear: the biggest overhead was registered for moderate traffic, 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 inconsistencies in the web server. The results obtained for the high level of traffic should also be indicative of how our solution affects fairly active production environments. Therefore, we concluded that our solution doesn't introduce a noticeable overhead into existing web applications, even for high levels of traffic.

5. Conclusions

To conclude we will now summarize some of the most important merits of our solution:

- It is easy to set up and it should also be compatible with any existing web application;
- Using our solution doesn't require any modifications to existing web applications and it can be entirely done by operation/security teams;
- It can automatically integrate honeypots into existing web applications while still being able to create versatile honeypots, unlike the previous solutions that we found;
- Setting up new honeypots is as easy as editing the MockingPot configuration file and creating optional HTML pages (not required for text honeypots);
- 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 significant 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 camouflaging 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;
- mTANNER mitigated multiple problems which we found in TANNER regarding our goals, as we discussed in Section 3.4. Well-configured honeypots should be hard to detect and should be able to keep attackers interested for long periods of time, as shown through the study described in Section 4.2.

References

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

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

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

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

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

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

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

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

[9] W. Bulajoul, A. James, and M. Pannu. Network Intrusion Detection Systems in High-Speed Traffic in Computer Networks. In *2013 IEEE 10th International Conference on e-Business Engineering*, pages 168–175, 2013.

[10] 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.

[11] 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.*

[12] 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.

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

[14] C. Pohl, A. Zugenmaier, M. Meier, and H.-J. Hof. B.Hive: A Zero Configuration Forms Honeypot for Productive Web Applications. volume 455, 05 2015.

[15] 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.

[16] H. Sanghvi and M. Dahiya. Cyber Reconnaissance: An Alarm before Cyber Attack. *International Journal of Computer Applications*, 63:36–38, 02 2013.

[17] D. Thomas-Reynolds and S. Butakov. *Factors Affecting the Performance of Web Application Firewall.*

[18] 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.