Evaluation of Deception-Based Web Attacks Detection

Xiao Han
Orange Labs
Eurecom
xiao.han@orange.com

Nizar Kheir
Thales Group
nizar.kheir@thalesgroup.com

Davide Balzarotti
Eurecom
davide.balzarotti@eurecom.fr

ABSTRACT

A form of moving target defense that is rapidly increasing in popularity consists of enriching an application with a number of deceptive elements and raising an alert whenever an interaction with such elements takes place. The use of deception can reduce some of the advantages of an attacker, making the exploration of the target to discover vulnerabilities a difficult and risky task. Another popular argument in support of deception techniques is that they are very effective at detecting attackers while maintaining a low, or even zero, false positive rate. However, to the best of our knowledge, no experiments have been performed to evaluate the use of deception in web applications. In particular, the lack of precise measurements of false positive and false negative rates makes it very difficult to understand if, and to which extent, deception can be an effective defense solution and a replacement for other traditional detection techniques.

In this paper, we first implement a web deception framework that allows us to introduce deception in any web application. Using this framework, we conduct two experiments that measure respectively the number of false alarms in a production environment and the detection accuracy during a controlled red team experiment with 150 participants. The first experiment has been performed for a period of seven months with 258 regular users and no false alarms have been triggered. The second experiment shows instead that deception is indeed capable of detecting attackers even before they could find one of the numerous vulnerabilities in the target application. However, 36% of the attackers who successfully exploited at least one vulnerability did so without triggering any of our traps. While more experiments are needed to better understand this phenomenon, our preliminary study seems to suggest that deception is a valuable companion of other detection techniques but it may not be suitable as a single standalone protection mechanism.

MTD'17, October 30, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5176-8/17/10...$15.00
https://doi.org/10.1145/3140549.3140555

1 INTRODUCTION

It has been estimated that there are over one billion websites on the World Wide Web today [15], and this number is steadily increasing over time. In 2015 the global business-to-consumer (B2C) e-commerce turnover has increased by about 20 percent, attaining a value of 2.2 billion dollars [12] and even governments are increasingly transitioning to web services to enable savings on the budget [33].

Unfortunately, this popularity regularly attracts a large number of attackers and according to Symantec [32] three quarters of the websites they scanned in 2015 contained unpatched vulnerabilities.

A large number of techniques have been proposed to secure web applications on the server side. Li et al. [21] classified these techniques into three categories: 1) secure construction of new web applications, 2) security analysis and testing of legacy applications and 3) runtime protection of legacy applications. The first category of techniques usually requires the design of new languages or frameworks, whereas the challenge of security analysis and testing stems from finding the right balance between correctness and completeness. Runtime protection typically provides a scalable solution to secure legacy applications at the cost of certain performance overhead.

These traditional measures, although being essential in any modern security arsenal, cannot provide a comprehensive solution against Internet threats. Due to these limitations, complementary solutions have been recently investigated to help anticipating threats and possibly warn users against attacks in their very early stages. In particular, intrusion deception techniques have recently attracted a lot of interest among security researchers [1, 4].

Previous work in this area first focused on using honey URLs to detect web bots [8] or flash crowd attacks [13]. Recently, more work has been done to support the adoption of deception techniques for the purpose of web attack detection – for instance by inserting honey configuration files, invisible links, HTML comments that contain fake accounts information [35], decoy forms [18], and honey URL parameters [27]. All these solutions aim at attracting the attention of attackers who are trying to discover vulnerabilities in a web application, detect their attempts, and finally protect the target system.

These forms of deception are also an excellent example of moving target defense. In fact, a web application enriched with deceptive elements add uncertainty in the attacker steps, as it is unclear which elements (e.g., form fields) belong to the real application and which ones are instead just traps positioned to detect the attacker's moves. Moreover, since the actual number, nature, and position of each deception element is not fixed, but decided by the administrator, the resulting system is different from any other similar applications.

Therefore, adding decoys to a system is a way to "mutate" it – resulting in a large number of different instances. Moreover, the set of decoys deployed on a system needs to be constantly updated and modified over time, thus making the system evolving and appearing different from an attacker perspective.

When more and more companies are already promoting and selling deception-based protection systems, there is not yet a common consensus among the scientific community about the effectiveness of these techniques and their ability to detect cyber attacks. In particular, one of the main gaps in the current understanding of deception techniques is the lack of precise measurements of their false negative and false positive rates. Our experiments have shown that deception can detect attackers, even before they discover and exploit a vulnerability in the target system. However, evaluating the false negatives (i.e., the number of attackers that have *not* been detected) is much harder.

In this paper we present two separate experiments we conducted to evaluate these important aspects. To perform our tests we first implemented a web deception framework that allows to easily introduce different forms of deceptive elements to any web application without changing its source code or affecting the backend server. Our solution is deployed as a transparent reverse proxy, which injects deceptive elements in outgoing HTTP traffic based on a set of configurable rules, and remove them from the incoming HTTP requests before they are forwarded to their destination. When a deceptive element is triggered, our framework generates an alert and, if requested to do so, redirects the current HTTP session to another endpoint where the attacker can be monitored without causing any damage.

Using our framework, we perform two experiments to measure the false negative and false positive rates of these techniques. In the first experiment we installed our framework to protect a production content management system (CMS) for a period of seven months – from December 2016 to June 2017. We introduced examples of all known web deception techniques we could find in the literature, and added two new ones we developed specifically for this task. During our experiment, the CMS system was publicly accessible on the Internet, and has been routinely used by 258 authenticated users. No false alert has been reported by our system throughout this experiment.

We then performed a second experiment during a red team exercise where 150 participants were asked to find vulnerabilities within a custom developed e-commerce application. To be able to compare results, the framework was configured to re-use the same techniques we adopted in the previous CMS experiment. In this case, we found that deception was able to detect 64% of the participants who successfully exploited at least one vulnerability during the exercise.

However, 36% of the successful attackers were able to exploit the system without triggering any deception elements. This test seems to support the hypothesis that deception-based techniques are good candidates for the purpose of attack detection, but only if combined with other protection and detection techniques. In particular, the results of our experiments show that deception may enhance the ability to detect attacks. They also show that deception alone is insufficient to protect a web application, and more research is needed in this direction to better understand the limitations of this technique.

2 BACKGROUND

Deception relies on a "*planned set of actions taken to mislead attackers and to thereby cause them to take (or not to take) specific actions that aid computer-security defenses*" [37].

The use of deception techniques in computer security was first proposed over two decades ago under the form of a user account populated with fake documents to track the activity of a remote attacker [31]. The fictitious setting used to monitor the attacker is nowadays known by security researchers as a *honeypot* [20] and has been widely used to detect attacks and monitor many types of malicious activities [10, 28]. The planted documents deployed in a legitimate system have also gained popularity [19] as a way to detect insider attacks. Since then, other studies have adopted similar techniques, such as *honeyfiles* [36], and also *decoy* documents [7].

Another popular deception technique is *honeytoken* [26], which mainly consists of "*a digital or information system resource whose value lies in the unauthorized use of that resource*" [30]. The simple fact of using such honeytoken may indicate a malicious intention since normal users are not supposed to be aware of such information. Many variants of this technique can be found in follow-up contributions such as honey passwords [16], honey HTTP parameters [27], and honey data in a database [6, 9].

Analogous to conventional deceptive military tactics, deception can also reply to attacks with decoy actions, instead of indicating an access violation, but fake protocol messages [14], delayed response [17], or crafted error messages [23]. In this case, the deception elements are supposed to intrigue attackers and make them believe that they are about to compromise the target system.

Deception techniques, if used correctly, can place defenders a step ahead of attackers, by modifying the system with additional traps that increase the risk of being detected. For this reason, deception has been proposed as a good example of *moving target* defense. For instance, Crouse et al. [11] proposed probabilistic models that confirmed that the combination of deception and network address shuffle provides the largest impact to attacker success. Urias et al. [34] extended a deception environment to clone the entire subnet that has suspicious activities. Then they proposed to modify the cloned network view, host attributes and network files as a moving target defense.

3 METHODOLOGY

Our aim is to design experiments to evaluate both the ability of deception-based techniques to detect web attacks, and their rate of false positives. To achieve this goal, we first survey existing work and collect a catalog of deception techniques that had been previously proposed in the literature for web-based attack detection. We then implemented a framework that allowed us to quickly and transparently insert these elements in an existing web application. At the end of this Section we discuss the strategy that we use to deploy deception techniques, which finally allows us to perform the experiments described in Section 4.

3.1 Deceptive Elements

Most deception techniques for the purpose of web attack detection actually find their root in the concept of honeytoken [30]. Early examples of this technique mainly consisted of using honey hyperlinks to detect phishers [22], flash crowd attacks [13] and web bots [8]. Most recently, several works promoted the use of deception for web applications [18, 27, 35]. Moreover, the OWASP AppSensor project [25] that aims at detecting and responding to attacks from within the application, provides a detailed description of honey traps that may be used as detection points inside a web application. More precisely, it classifies these detection points into three categories: 1) alteration to honey trap data, which mainly refers to honeytoken in HTTP protocol such as fake hidden form fields, additional URL parameters and cookies, 2) honey trap resource requested, for instance, fake page and directory listed in the application's `robot.txt` file and 3) honey trap data collected and used by the attacker, where a typical example is the use of honey accounts information only visible in source HTML code.

In this paper, we design our system using the aforementioned deception techniques. Furthermore, we extend them by adding two additional deceptive elements. The first is a fake protected area (e.g., an administration console) that requires HTTP authentication. This is similar to a normal fake page, but it contains no content and the fact that it prompts the client for an authentication may attract both humans and automated tools that try to brute-force the password to gain access to the protected resource.

The second additional technique includes a set of fake vulnerabilities that, when tampered with by an attacker, return realistic error messages based on the pre-defined attack types. The focus of this particular type of deceptive element is to maintain the attackers busy trying to exploit some bug that does not exist.

For our experiment we supported two types of fake vulnerabilities: local file inclusion and SQL injection. For instance, if an attacker alters the honey trap by attempting a SQL injection as illustrated below, the fake vulnerability returns a classic error that exposes a valid-looking vulnerability.

```
Injection: ' or 1=1 --

Response: Invalid query: You have an error in your
SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to
use near '' at line 1
```

3.2 Deception Framework

We now describe the design requirements and the implementation details of our deception framework, which allows us to transparently add different deceptive elements without the need to modify the target application. While the framework is not the main contribution of our work, it is necessary to quickly test different approaches and conduct experiments on their actual effectiveness in a real-world deployment.

3.2.1 Design Requirements. To build a deception framework that enables to integrate deceptive elements into most web applications, our system needs to achieve the following design requirements:

- *Language/Framework Independent*
  In order to keep our system applicable to a large number of different web applications, our application should not be tied to any programming language nor any particular framework.
- *No Access to Source Code*
  One of the main goals of our system is to provide an additional layer of protection without touching or modifying the target web applications.
- *Non-Interference.*
  The system needs to support the insertion of any type of deceptive elements, including additional URL or form parameters, fake pages, or honey accounts. However, it is important that these elements do not interfere with the normal behavior of the target application.

To satisfy these requirements, our system was designed to work at the HTTP protocol level, modifying requests and responses on the fly by acting as a reverse proxy in front of the target application. To avoid any potential interference with the original application, the system makes sure that any direct sign of deceptive elements or any side-effect for their presence will be transparently removed from the incoming traffic and so it may not reach, nor to be processed by, the target application.

3.2.2 Implementation. Following the above design requirements, we decide to deploy our system in-front of the target web application in the form of a transparent reverse proxy, as illustrated in Figure 1. Our framework implements common mechanisms to modify the HTTP protocol and HTML content such as adding additional cookies, hidden input form fields, additional HTTP GET and POST parameters, fake protected areas, and fake vulnerabilities, which enables the injection of all deceptive elements described section in 3.1.

The deception framework is configured using simple regular expression rules that specify which request or response needs to be intercepted and modified, the type of deceptive element that needs to be inserted, and the fake data associated to that element. For each rule, the reverse proxy is responsible

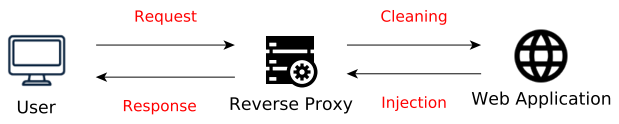


Figure 1: Deception Framework

for adding the specified deception technique to the outgoing responses, and removing them from the incoming requests, before it may relay them to the target web application.

The reverse proxy is implemented based on an open source HTTP hacking tool, Hoxy [29]. It enables the interception of HTTP request and responses, observing and altering all aspects of the requests and responses. Coupled with a HTML parsing library, it allows us to modify the HTML content on the fly during runtime.

Adding Deceptive Elements While Hoxy intercepts the HTTP response, it exposes the HTTP header and also the body in the form of JSON, string, jQuery or raw buffer. It is quite straightforward to add deceptive elements which are located in the HTTP header. For instance, to add a fake cookie, we simply add a `set-cookie` field with the desired data. Similarly, the fake protected area is implemented by adding a `WWW-Authenticate` field in the HTTP header. To implement a deceptive HTTP GET parameter, we modify the HTTP response status code to 302, which then redirects the original request towards the new URL that has been appended with the fake parameter.

There are also a few types of deceptive elements that require modifications of the HTTP body such as the fake hidden input field and fake data in JSON response. Hoxy supports by default the edit of HTML as a DOM object similar to jQuery. We implement hidden input field by searching the `form` field in the DOM object and further adding the hidden field inside it. It is also possible to convert the HTTP body to a JSON object to which we can easily add the fake data.

Finally, in order to implement the decoy vulnerabilities, we detect any modification on the deceptive elements and then apply regular expressions to determine whether the modification belongs to a known attack pattern. For this purpose we reuse the type of attacks defined in the Glastopf Web Honyepot [24]. For our experiments, we support two of the most popular types of attacks, which are local file inclusion and SQL injection. If the request matches one of the regular expressions associated to those attacks, our system forges and returns a realistic error message to deceive the attackers and make them believe that the system is indeed vulnerable.

Cleaning Deceptive Elements Our framework keeps and updates a record of the injected deceptive elements and their locations. For each incoming HTTP request whose URL is known to contain deceptive elements, our system cleans these elements before handling the request to the target web application. In this way, our framework does not interfere with the target application, and the latter only receives original requests where all signs of deception have been removed.

Reporting and Redirection When the proxy identifies that a user has interacted with one of the deceptive elements, our framework logs the incoming request including the source IP address, the deceptive element, and any user-supplied data that may contain attack information. The framework can also be configured to to redirect incoming requests whose URL matches a regular expression to another application by modifying the back-end server of the reverse proxy. This functionality, for example, allows defenders to redirect attackers from the target application to a similar but protected version of it so as to contain them and better observe their behaviors.

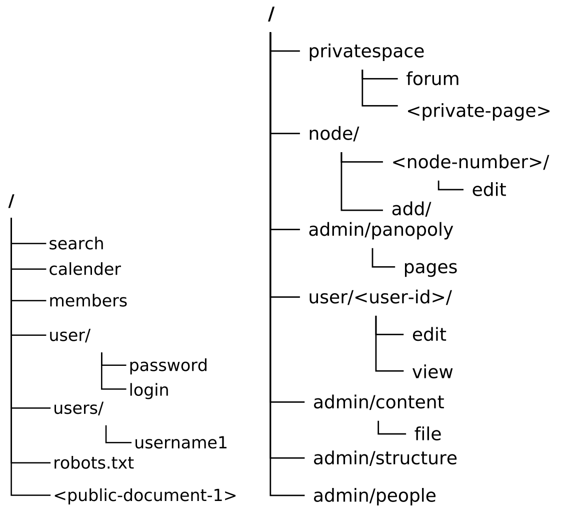
3.3 Deployment Strategy

The proper placement and integration of various deceptive elements into a target web application is still an open research question. For our purpose, we manually identified the position in which each technique was deployed by following the OWASP testing guide [25], which presents the best practices for web penetration testing. OWASP testing methodology is based on a black-box approach where testers have little information about the target application. In this situation, the testers play exactly the role of an external attacker. Moreover, this guide provides concrete examples of the techniques to use to discover vulnerabilities, which makes it a good starting point to select the placement of deceptive elements right where testers/attackers would most likely look for existing vulnerabilities.

As a result, we combined existing deception techniques with the methodology proposed by OWASP testing guide to devise deception at enticing locations under the perspective of an attacker. For example, at the information gathering stage, we place honey trap resources in `robot.txt` files, as the OWASP testing guide suggests to review them for information leakage. We then designed specific honey accounts disguised as a configuration file of the web application, which relates to the second step of the configuration and deployment management testing. Regarding the identity, authentication and session management testing, we deploy at both the login page and password reset page multiple deception techniques such as honey accounts only visible in source HTML code, hidden honey form fields and additional session-related cookies. Lastly, we placed additional `HTTP GET` and `POST` parameters both in the web pages and in the client JavaScript code that are subject to the input validation testing. The name of such parameters were carefully selected according to the content of the web application. On top of these parameters, we further deployed the two classes of fake vulnerabilities discussed above.

4 EXPERIMENT DESIGN

In this section, we present two experiments designed respectively to measure the number of generated false alarms due to the deployment of deception techniques in a production



(a) Public space **(b) Private space**

Figure 2: CMS application tree structure

Table 1: Deception in public CMS space

URL path	Deception technique	Quantity	FV	Protected
/robots.txt	Honey trap resource	3		✓
/search/*	Honey POST parameter	1	✓	
	Additional cookie	1	✓	
/usr/login	Honey GET parameter	1	✓	
	Honey account	1		
/usr/password	Hidden input field	1	✓	

Table 2: Deception in private CMS space

URL path	Deception technique	Quantity	FV	Protected
/node/add/*	Honey GET parameter	1	✓	
/node/0/*	Honey trap resource	1		✓
/node/*/edit	Honey GET parameter	1	✓	
/user/*/edit	Honey GET parameter	1	✓	
/user/*/view	Honey GET parameter	1	✓	

*FV: fake vulnerability enabled

*Protected: fake protected area requiring authentication

environment, and evaluate their ability to detect realistic web attacks.

This is a challenging task, as deception should be evaluated against human attackers that manually try to exploit an unknown application. In fact, automated scripts that target known vulnerabilities would not trigger any detection element, as they are programmed to generate exactly the request required to exploit the target, without visiting or interacting with anything else. For this reason, honeypots are not suitable to perform this test, and collecting data about hundreds of real attackers is a gigantic effort. We solved this problem by splitting the experiment in two parts. First, we deployed our solution on a real application, and monitored its users to detect any possible false alarm raised by our deception proxy. However, since this deployment cannot provide the data required to test the detection rate, we also performed a second experiment, this time deploying similar deceptive techniques in an application designed for a penetration testing experiment.

4.1 Use of Deception in a Real Content Management System

In the first experiment, we implemented deception inside a Content Management System (CMS) that is publicly accessible on the Internet. The testing of known deception techniques in such a production environment allows us to monitor real user interactions with the system, and to evaluate the false positives rate caused by these techniques.

4.1.1 CMS Application. The web application is based on Open Atrium [1], which is an extensible collaboration framework. It provides out-of-the-box functionalities such as dashboard, document sharing, forum, agenda, and user access control. The software was customized to allow research project members to access and manage publicly accessible project websites. These websites allow access to users (not members of the project), who are granted access only to the public content of each website. Collaborators of each project may also access to a private space that requires authentication by the service. Figures 2a and 2b illustrate respectively the structure of the public and private spaces. The public space consists of public documents, a search page, and a login page. Once authenticated, a user may view and edit private pages. The administrator can further add/remove users and manage user access.

4.1.2 Deception Placement. This experiment aims at evaluating the false positive rate of deception techniques in presence of legitimate users. While the two CMSes are different in structure and scope, we placed the deception elements to resemble as close as possible the deployment adopted in the CTF exercise. Table 1 and 2 present respectively the deception techniques that have been introduced in the public and in the private space.

4.2 Use of Deception in a Capture-The-Flag Competition

In the second experiment, we integrate deception techniques in a Capture The Flag (CTF) exercise, in which participants are presented with a specific environment where vulnerabilities are purposely planted. By successfully exploiting a vulnerability, the participant finds a flag which allows him to score points. The participant with the highest final score wins the exercise.

While CTF participants are not necessarily a perfect model of real attackers on the Internet, the red team exercise is designed to mimic what users would do to discover vulnerabilities in an unknown piece of software. Moreover, using a CTF competition to evaluate the accuracy of deception techniques at detecting attackers has the advantage of providing access

[1] https://www.drupal.org/project/openatrium

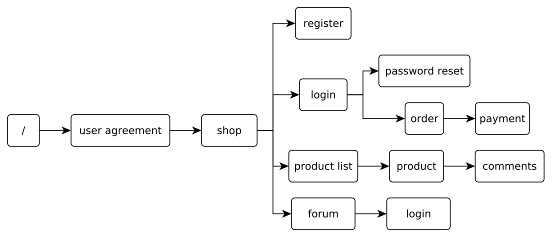


Figure 3: CTF Application Workflow

Table 3: Deception in CTF exercise

Page	Deception technique	Quantity	FV	Protected
User agreement	Honey POST parameter	1	✓	
	Additional cookie	1	✓	
Shop login	Honey POST parameter	1		✓
	Honey account	1		
Forum login	Honey account	1		
Account API	Honey GET parameter	1	✓	
Product comment	Honey POST parameter	1	✓	
/robots.txt	Honey trap resource	3		✓
/web.config	Honey trap resource	1		✓
/web.config	Honey account	1		

*FV: fake vulnerability enabled

*Protected: fake protected area requiring authentication

to hundreds of "attackers", something that would be very difficult by simply collecting data using a real application.

4.2.1 CTF Environment. The CTF exercise we used for our test was organized by Orange Labs and aimed at simulating a situation where participants audit the security of an e-commerce application in a black-box approach. The e-commerce application has been purposely developed for the experiment, following the workflow illustrated in Figure 3. Each visitor is first presented with a user agreement page that already presents the first challenge. This page contains an obfuscated JavaScript code that requires the user to read at least for one hour the user terms and conditions. By successfully hijacking the client side check, the user may visualize the list of products supplied by the platform, and the user comments on each product. Furthermore, users can create an account and use it to log in and checkout their orders, and finally perform online payment. In addition, there is also a form that allows authenticated users to post public messages.

Many classic vulnerabilities such as cross site scripting, local file inclusion, SQL inject, and remote code execution have been planted at different locations including the user profile, product comment page, order page and in particular the online payment page. In total, 15 flags have been inserted inside the CTF application.

4.2.2 Deception Placement. Our main goal is to evaluate the ability of deception techniques to detect web attacks in their early stage. Following the placement strategy described in Section 3, we manually inserted a number of deception tokens, as summarized in Table 3. Note that when the deception elements are properly designed, they cannot be identified by an attacker without first interacting with them.

We started by modifying `robots.txt` and `web.config` (a classic configuration file for Windows web server) to insert four honey trap resources and one honey account. We then added an additional `HTTP POST` parameter and a fake cookie in the user agreement page to see how deception affects the advancement of participants. We also added fake `HTTP GET` and `POST` parameters and honey accounts in most of the login-related pages and APIs. Lastly, we placed another fake `HTTP POST` parameter in the product comment page. Note that we did not place any deception on pages such as orders and payment where many real vulnerabilities (i.e. flags) have been planted. The main reason is that we focus our experiments on evaluating the ability of deception elements to detect attackers in the very early stages of their interaction with the target application. The second reason is that we were asked by the organizers to minimize the interference between deception techniques and real CTF vulnerabilities, so that we may not discourage the participants.

5 RESULTS

In this section we present and discuss the results that we obtained during our two experiments.

5.1 CMS Experiment

The experiment on the real CMS application has been performed during a period of seven months, from December 2016 to June 2017. Over this period, the application has been used to host 13 active projects, each of which had its own dedicated sub-domain name. In total, we observed 258 users that have successfully authenticated and interacted with the application. All users are the internal employee of the company where the experiment has been performed.

Public Space – Four alerts have been triggered from the honey trap resources placed inside the `robots.txt` file. The four subsequent connections were originating from the same remote IP address. According to the web server access logs, this IP address was actually performing a scan attempt to test the system and to check for known vulnerabilities. We also found similar abuse reports for the same IP address in the `AbuseIPDB` [2] service, which confirms that our system has indeed detected a malicious IP address.

Private Space – Over the total period of our experiments, no alerts have been triggered by the deception elements that were placed inside the private space. Therefore, and based on our experimental setup, deception techniques that have been deployed inside the CMS application have generated zero false positives. This seems to confirm that, since these deception techniques are implemented at the protocol or source HTML code level, they remain mostly invisible to normal (i.e. benign) users.

5.2 CTF Experiment

The CTF competition has been conducted in a local network for a period of 8 hours in September 2016. In total, the

[2] https://www.abuseipdb.com/

Table 4: Number of distinct IP addresses detected

Page	Traps	Detected Participants	Average N. of Attempts
Configuration files	5	14	8
User agreement	2	56	28
Account API	1	22	6
Product comment	1	16	446
Shop login	2	22	11
Forum login	1	7	2

*Configuration files: /robots.txt and /web.config

event counted 150 participants (each clearly identifiable by a different IP address), which included a mix of information security students and security professionals.

Overall, only 25 (16.7%) of them successfully discovered at least one flag, while 84 (56%) have triggered at least one of the 12 deception traps – e.g., by trying to tamper with a fake cookie or form parameter, by trying to use a honey credential to log in, or by visiting some of the honey URLs. These results seem to suggest that the deceptive elements were easier to trigger than the real vulnerabilities, which is exactly what a good deployment strategy should achieve. Interestingly, 64% of the participants that discovered a real vulnerability also triggered a deception trap, showing that while an attacker who spends enough time and resources trying to discover a real bug is also likely to raise an alert due to his interaction with the deceptive elements, this is not always the case.

5.2.1 Manual vs Automated Actions. A first interesting question we want to understand is whether our traps were triggered by manual interactions or by automated tools and scanners. To answer this question we inspected the `user agent` strings in the access log of the web server. In the context of a CTF exercise, we believe the user agent is a reliable source of information since there is no incentive for the participants to hide their actions or disguise their activities.

We found that five participants triggered the four honey trap resources (placed in the `robots.txt` and `/web.config` files) using popular scanner tools (Nikto and DirBuster). For the remaining honey traps, the trigger connection originated from popular browsers. We thus consider that those participants have discovered manually the traps. Interestingly, we also found evidence of users who run the `sqlmap` tool to try to exploit the fake SQL injection vulnerability, and others used the `hydra` tool to brute force the HTTP authentication of a fake URL. This supports the hypothesis that these two new form of deception we introduced in our framework are successful in slowing down attackers by tricking them into focusing their effort on fake parts of the application.

5.2.2 Impacts of Deception Placement. In order to measure the impacts of deception placed at different locations, we evaluate the number of participants that triggered each trap. Furthermore, we analyze the average number of manual attempts that have been performed on top of the deception element, which reflects the extent to which the participants are enticed by the given deception.

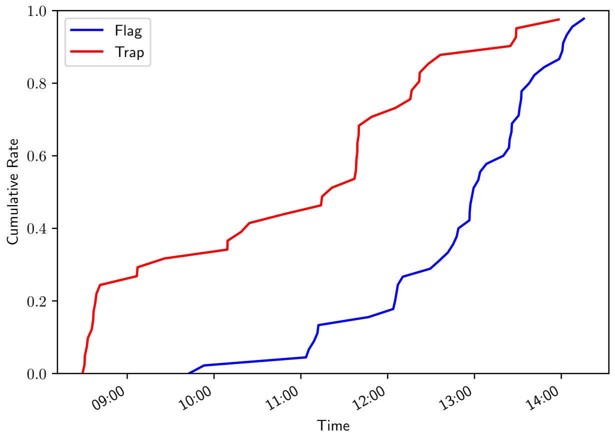


Figure 4: Cumulative distribution of detection and flag

As illustrated in Table 4, for each location, we present the number of traps implemented, the number of distinct IP addresses detected and the average number of manual attempts that modified the traps on that location. The user agreement page is the most effective location to detect the participants using deception. This page is actually situated just after the starting point of the application workflow, which seems to suggest that deception placed at the first pages of the workflow is quite effective to detect attackers since among the 84 detected IP addresses, 56 of them have been detected at the user agreement page.

On the other hand, deception at product comment page has received the largest average number of manual attempts per deception element, which indicates that this location is particularly interesting for participants. One possible explanation is that the product comment page accepts user-provided comments, so the participants spent more time playing with its parameters looking for a possible injection vulnerability. This kind of insight is quite useful and should be integrated while devising and deploying deceptions for a web application.

5.2.3 Effectiveness. In order to shed some light on whether deception techniques are effective in detecting attackers, we analyze the potential relationship between the triggered honey traps and the flags that were found by each participant. For each participant who found at least one flag, we collected the time at which each flag was discovered and each trap (if any) was triggered. Figure 4 presents the cumulative rate of honey traps triggered and flags found by the 25 participants. The horizontal axis describes the time during which the CTF has been played. The vertical axis presents the cumulative rate. The red and blue curves illustrate respectively the cumulative rates of the triggered honey traps and the discovered flags.

Almost 35% of honey traps have been triggered even before any flag could be found. This seems to indicate that honey traps may be used as an early warning system to detect attackers before they may identify true vulnerabilities and weaknesses in the target application. Further investigation shows that 16 of the 25 participants who have discovered at least one flag have triggered at least one honey trap. Moreover, 14 out of 16 have triggered a trap *before* finding a flag.

By the time 80% of the discovered traps had been triggered, only 30% of flags had been found. This seems to indicate that the more the attacker advances in his attack plan (by getting deeper in his attack path and finding new flags), the more he will get exposed and tricked by additional traps (80% of traps activated at this stage). We can deduce from those figures that the expected efficiency rapidly increases when the attacker advances further in his attack scenario, which may enhance the detection rates. This means that to be more effective, honey traps need to be composed and intertwined within the workflow of an application, and not only as a single layer or on the front page.

6 DISCUSSION

In this paper, we presented the design and implementation of a deception framework for web applications. We used our framework to conduct two experiments to evaluate respectively the effectiveness of deception techniques to detect web attacks and the false positive rate of the same techniques when deployed in a production environment. Even though the two experiments have been conducted in different conditions, we have implemented and deployed similar deception techniques in both of them.

Our test on a real application raised no false alarms over a period of seven months. This seems to confirm one of the main advantage of deception-based defenses: the ability to provide detection with zero false alarms.

The detection accuracy was obtained through a red teaming exercise, which we consider the most appropriate method available to perform such measurement in a controlled environment. While the results depend largely on the profile of the participants, our tests included a considerable number (150) of users with different skills and knowledge in web security. Nonetheless, a drawback of using a CTF scenario instead of observing real attackers is that CTF participants are more aggressive and have no incentives in being stealthy and in reducing their footprint on the target system. A real attacker may be more cautious, thus resulting in a different (probably lower) rate of deception elements triggered and vulnerability discovered. Hence, the results of this second experiment are positive but still far from perfect – with 36% of the attackers that were successfully able to find and exploit a vulnerability without interacting with any deceptive elements. We believe that this can be a consequence of the placement strategy we adopted, and maybe a more aggressive deployment would provide a higher detection rate. However, too many deceptive elements can tip of the attackers about the presence of this type of defense, therefore actually reducing the overall efficiency of the technique. More tests are required to better understand this phenomenon and the intricacies of an "optimal" solution to deploy deception on a web application.

7 RELATED WORK

The literature is brimming with approaches to protect web applications [21] against attacks. Nevertheless, current web attacks detection techniques fail to reliably and proactively detect attacks in their early stage. Due to these limitations, complementary solutions such as deception techniques have been recently investigated by the research community [1, 4]. In this paper, we focus mainly on those solutions that adopt a deception-based web application protection scheme. We further group existing approaches into two categories, one used to enhance attack detection and the other used for the purpose of attack mitigation.

7.1 Attack Detection

Brewer et al. [8] proposed a web application that integrates decoy links. These links are invisible to normal users, but are expected to be triggered by crawlers and web bots that connect to the application. Similarly, Gavrilis et al. [13] presented a deceptive method that detects denial of service attacks on web services by using fake links hidden in the web page. In the same way, McRae and Vaughn [22] submitted honey accounts which contained decoy URL to phishing sites to track phishers while they viewed the honey accounts.

Another approach to deceive web-based attacks consists of using fake information disguised as web server configuration errors. Only malicious users are expected to manipulate or exploit these errors, which expose them to detection by the system. In this scope, Virvilis et al. [35] introduced honey configuration files, such as `robots.txt`, including fake entries, hidden links, and HTML comments that indicate honey accounts, in order to detect potential attackers. Other studies proposed decoy forms [18] and honey URL parameters [27] that display fake configuration errors in an effort to mislead attackers and protect the target system.

In [2], Almeshekah proposed centralized deceptive server which enables the implementation of deception to protect target web servers. In each web server, the proposed system hooks incoming requests and further sends their metadata toward the centralized server where a decision is taken on whether the system should respond with deception. The author further analyzed the performance overhead introduced by the system.

7.2 Attack Mitigation

Julian [17] proposed to alter the response time of a web-based search engine by injecting random delays in reply to malicious requests in order to confuse an attacker. Anagnostakis et al. [3] introduced *"shadow honeypots"* that extend traditional honeypots with anomaly-based buffer overflow detection to protect web application. The shadow honeypot is a copy of the target application, with common context and application state. It is used to analyze anomalous traffic and to enhance the accuracy of anomaly detection.

Finally, Araujo et al. [5] presented a technique to transform traditional patches into *"honey-patches"*, designed to remove the vulnerability while at the same deceives the attackers into believing that the attacks have succeeded. On the detection of an attack targeting the known vulnerability of the web server, the system forwards the attacker to an unpatched but isolated instance of the same web server.

The work in this paper differs from previous work in that we focus on attack detection instead of attack mitigation. Unfortunately, most of the previous work in deception-based attack detection did not provide experiments and evaluations of the proposed techniques. Therefore, in this paper we present two experiments we conducted to evaluate the effectiveness and the false positive rate of deception-based web application protection.

8 CONCLUSION

In this paper, we present the design and implementation of two experiments we performed to evaluate the use of deception-based techniques to detect web attacks in their early stage. The first experiment measured the number of false alarms under a production environment. During a period of seven months, zero false alarms were raised during the interaction of 258 authenticated users. We also conducted a red teaming experiment with 150 participants. Our experiment setup was able to detect 64% of participants that successfully exploited at least one vulnerability. Our test shows interesting results which support the idea of deception as a low-FP detection approach, but also shows a false negative rate that is still quite high for a standalone solution. We hope to see more experiments in this area, in order to confirm the use of deception techniques for attack detection and better identify its limitations.

REFERENCES

[1] M. Almeshekah and E. Spafford. The case of using negative (deceiving) information in data protection. In *International Conference on Cyber Warfare and Security*, 2014.
[2] M. H. Almeshekah. *Using deception to enhance security: A Taxonomy, Model, and Novel Uses.* PhD thesis, 2015.
[3] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. P. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Usenix Security*, 2005.
[4] C. Anton. "Deception as Detection" or Give Deception a Chance?, 2016.
[5] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2014.
[6] M. Bercovitch, M. Renford, L. Hasson, A. Shabtai, L. Rokach, and Y. Elovici. HoneyGen: An automated honeytokens generator. In *IEEE International Conference on Intelligence and Security Informatics(ISI)*, 2011.
[7] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. *International Conference on Security and Privacy in Communication Systems*, 2009.
[8] D. Brewer, K. Li, L. Ramaswamy, and C. Pu. A link obfuscation service to detect webbots. *International Conference on Services Computing (SCC)*, 2010.
[9] A. Čenys, D. Rainys, L. Radvilavičius, and N. Goranin. Implementation of Honeytoken Module In DBMS Oracle 9ir2 Enterprise Edition for Internal Malicious Activity Detection. In *IEEE Computer Society's TC on Security and Privacy*, 2005.
[10] F. Cohen. A note on the role of deception in information protection. *Computers & Security*, 1998.
[11] M. Crouse, B. Prosser, and E. W. Fulp. Probabilistic performance analysis of moving target and deception reconnaissance defenses. In *Workshop on Moving Target Defense.* ACM, 2015.
[12] Ecommerce Foundation. Global b2c e-commerce report 2016. https://www.ecommercewiki.org/wikis/www.ecommercewiki.org/images/5/56/Global_B2C_Ecommerce_Report_2016.pdf, 2016.
[13] D. Gavrilis, I. Chatzis, and E. Dermatas. Flash crowd detection using decoy hyperlinks. *International Conference on Networking, Sensing and Control (ICNSC)*, 2007.
[14] H. C. Goh. Intrusion deception in defense of computer systems. Technical report, 2007.
[15] Internet Live Stats. Total number of websites. http://www.internetlivestats.com/total-number-of-websites/, 2017.
[16] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. In *ACM SIGSAC conference on Computer & communications security*, 2013.
[17] D. P. Julian. *Delaying Type Response for Use By Software Decoys.* PhD thesis, 2002.
[18] C. Katsinis, B. Kumar, S. Technology, and R. Systems. A Framework for Intrusion Deception on Web Servers. In *International Conference on Internet Computing, ICOMP'13*, 2013.
[19] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security.* ACM, 1994.
[20] S. Lance. The Value of Honeypots, Part One: Definitions and Values of Honeypots, 2001.
[21] X. Li and Y. Xue. A survey on server-side approaches to securing web applications. *ACM Comput. Surv.*, 2014.
[22] C. M. McRae and R. B. Vaughn. Phighting the phisher: Using Web bugs and honeytokens to investigate the source of phishing attacks. *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2007.
[23] B. Michael, M. Auguston, N. Rowe, and R. Riehle. Software Decoys: Intrusion Detection and Countermeasures. In *Proceedings of the IEEE Workshop on Information Assurance*, 2002.
[24] MushMush Foundation. Glastopf. https://github.com/mushorg/glastopf/blob/master/glastopf/requests.xml, 2017.
[25] OWASP. Appsensor project guide. https://www.owasp.org/index.php/File:Owasp-appsensor-guide-v2.pdf, 2015.
[26] A. Paes de Barros. RES: Protocol Anomaly Detection IDS - Honeypots, 2003.
[27] A. R. Petrunić. Honeytokens as active defense. *International Convention on Information and Communication Technology, Electronics and Microelectronics(MIPRO)*, 2015.
[28] N. Provos et al. A virtual honeypot framework. In *USENIX Security Symposium*, 2004.
[29] G. Reimer. Hoxy. http://greim.github.io/hoxy/, 2015.
[30] L. Spitzner. Honeytokens: The other honeypot, 2003.
[31] C. Stoll. *The cuckoo's egg: tracking a spy through the maze of computer espionage.* 1989.
[32] Symantec. Internet security threat report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf, 2016.
[33] UK Government Digital Service. How digital and technology transformation saved 1.7bn last year. https://gds.blog.gov.uk/2015/10/23/how-digital-and-technology-transformation-saved-1-7bn-last-year/, 2015.
[34] V. E. Urias, W. M. Stout, and C. Loverro. Computer network deception as a moving target defense. In *International Carnahan Conference on Security Technology (ICCST).* IEEE, 2015.
[35] N. Virvilis, B. Vanautgaerden, and O. S. Serrano. Changing the game: The art of deceiving sophisticated attackers. *International Conference on Cyber Conflict, CYCON*, 2014.
[36] J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, 2004.
[37] J. J. Yuill. *Defensive Computer-security Deception Operations: Processes, Principles and Techniques.* PhD thesis, 2006.